What is Numpy?

NumPy is a core Python library for scientific computing, offering powerful tools for working with numerical data.

- It provides a high-performance multidimensional array object, along with specialized objects such as masked arrays and matrices. NumPy also includes a wide range of optimized functions for performing fast operations on arrays—covering areas like mathematical and logical computations, shape manipulation, sorting, indexing, input/output, discrete Fourier transforms, basic linear algebra, statistical analysis, random number generation, and more. At the heart of NumPy is the ndarray object, which represents n-dimensional arrays containing elements of the same data type, enabling efficient storage and processing of large datasets.

In [605…
```
import numpy as np # Importing the NumPy library
```

In [606…
```
np.array([2,4,56,422,32,1]) # Creating a NumPy array
```

Out[606…
```
array([  2,   4,  56, 422,  32,   1])
```

In [607…
```
a = np.array([2,4,56,422,32,1]) #Vector
print(a)
```

```
[  2   4  56 422  32   1]
```

In [608…
```
type(a) # Checking the type of the array
```

Out[608…
```
numpy.ndarray
```

In [609…
```
new = np.array([[45,34,22,2],[24,55,3,22]]) # Creating a 2D NumPy array Matrix
print(new)
```

```
[[45 34 22  2]
 [24 55  3 22]]
```

In [610…
```
np.array ([[2,3,33,4,45],[23,45,56,66,2],[357,523,32,24,2],[32,32,44,33,234]]) # 3D Array ( Matrix)
```

Out[610…
```
array([[  2,   3,  33,   4,  45],
       [ 23,  45,  56,  66,   2],
       [357, 523,  32,  24,   2],
       [ 32,  32,  44,  33, 234]])
```

dtype

Specifies the desired data type for the array. If not provided, NumPy automatically determines the smallest data type capable of holding all the elements in the given sequence.

In [611…
```
np.array([11,23,44] , dtype =float) # Creating a NumPy array with float data type
```

Out[611…
```
array([11., 23., 44.])
```

In [612…
```
np.array([11,23,44] , dtype =bool) # Here True because python treats Non-zero values as True
```

Out[612…
```
array([ True,  True,  True])
```

In [613…
```
np.array([11,23,44] , dtype =complex) # Creating a NumPy array with complex data type
```

Out[613…
```
array([11.+0.j, 23.+0.j, 44.+0.j])
```

NumPy Arrays vs. Python Sequences

NumPy arrays have a fixed size once created, unlike Python lists, which can grow or shrink dynamically. If you change the size of an ndarray, a new array is created and the original is discarded.

All elements in a NumPy array must share the same data type, ensuring consistent memory size for each element. This uniformity, combined with NumPy's optimized C-based implementation, allows for advanced mathematical and scientific operations to be performed on large datasets efficiently—often with less code than using Python's built-in sequences.

Many scientific and mathematical Python libraries are built around NumPy arrays. While these libraries may accept Python sequences as input, they typically convert them to NumPy arrays internally and often return results as NumPy arrays.

arange

The arange function can be called with a flexible number of positional arguments, allowing you to specify just the stop value, or both start and stop values, and optionally the step size.

In [614…
```
np.arange(1,25) # Creating an array with a range of values from 1 to 25, 1 inclusive and 25 exclusive
```

Out[614…
```
array([ 1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16, 17,
       18, 19, 20, 21, 22, 23, 24])
```

In [615…
```
np.arange(1,25,2) # Creating an array with a range of values from 1 to 25, with a step size of 2
```

Out[615…
```
array([ 1,  3,  5,  7,  9, 11, 13, 15, 17, 19, 21, 23])
```

reshape

When reshaping an array, the product of the new dimensions must be equal to the total number of elements in the original array.

In [616…
```
np.arange(1,11).reshape(5,2) # Creating a 2D array with values from 1 to 10, reshaped into a 5x2 matrix
```

Out[616…
```
array([[ 1,  2],
       [ 3,  4],
       [ 5,  6],
       [ 7,  8],
       [ 9, 10]])
```

In [617…
```
np.arange(1,11).reshape(2,5) # Creating a 2D array with values from 1 to 10, reshaped into a 2x5 matrix
```

Out[617…
```
array([[ 1,  2,  3,  4,  5],
       [ 6,  7,  8,  9, 10]])
```

In [618…
```
np.arange(1,13).reshape(3,4) # Creating a 2D array with values from 1 to 12, reshaped into a 3x4 matrix
```

Out[618…
```
array([[ 1,  2,  3,  4],
       [ 5,  6,  7,  8],
       [ 9, 10, 11, 12]])
```

ones & zeros

You can create arrays filled entirely with 1s using ones() and arrays filled with 0s using zeros(). These functions are useful for initializing values—for example, setting up weight matrices of a specific shape in deep learning.

In [619…
```
np.ones((3,4)) # we have to mention inside tuple, which is the shape of the array we want to create. Here, it creates a 3x4 array filled with ones.
```

Out[619…
```
array([[1., 1., 1., 1.],
       [1., 1., 1., 1.],
       [1., 1., 1., 1.]])
```

In [620…
```
np.zeros((3,4)) # Creating a 3x4 array filled with zeros
```

Out[620…
```
array([[0., 0., 0., 0.],
       [0., 0., 0., 0.],
       [0., 0., 0., 0.]])
```

random()

The random() function in NumPy (from numpy.random) is used to generate arrays of random numbers.

In [621…
```
np.random.random((4,3))
```

Out[621…
```
array([[0.25602764, 0.91719383, 0.39000084],
       [0.04804188, 0.51257528, 0.46034875],
       [0.96903467, 0.50407805, 0.76435476],
       [0.0801154 , 0.55760182, 0.19633784]])
```

linspace

linspace (short for linearly spaced) generates a specified number of evenly spaced points within a given range. It is commonly used when you need values at equal intervals, such as for plotting graphs or creating test datasets.

In [622…
```
np.linspace(-10,10,10) # Creating an array with 10 evenly spaced values between -10 and 10, Lower range -10 and upper range 10
```

Out[622…
```
array([-10.        ,  -7.77777778,  -5.55555556,  -3.33333333,
        -1.11111111,   1.11111111,   3.33333333,   5.55555556,
         7.77777778,  10.        ])
```

In [623…
```
np.linspace(-2,12,6) # Creating an array with 6 evenly spaced values between -2 and 12, Lower range -2 and upper range 12
```

Out[623…
```
array([-2. ,  0.8,  3.6,  6.4,  9.2, 12. ])
```

identity

An identity matrix is a square matrix in which all the elements on the main diagonal are 1s, and all other elements are 0s. In NumPy, you can create one using np.identity() by specifying the size of the matrix.

In [624…
```
np.identity(3) # Creating a 3x3 identity matrix, where the diagonal elements are 1 and all other elements are 0
```

Out[624…
```
array([[1., 0., 0.],
       [0., 1., 0.],
       [0., 0., 1.]])
```

In [625…
```
np.identity(6) # Creating a 6x6 identity matrix, where the diagonal elements are 1 and all other elements are 0
```

Out[625…
```
array([[1., 0., 0., 0., 0., 0.],
       [0., 1., 0., 0., 0., 0.],
       [0., 0., 1., 0., 0., 0.],
       [0., 0., 0., 1., 0., 0.],
       [0., 0., 0., 0., 1., 0.],
       [0., 0., 0., 0., 0., 1.]])
```

Array Attributes

NumPy arrays (ndarray objects) have several built-in attributes that provide information about the array's structure and contents. Some commonly used attributes include:

- ndim - Number of dimensions (axes) in the array.
- shape - A tuple representing the size of the array in each dimension.
- size - Total number of elements in the array.
- dtype - Data type of the elements stored in the array.
- itemsize - Size in bytes of each element.
- nbytes - Total memory consumed by the array (in bytes).
- T - Transpose of the array.

In [626…
```
a1 = np.arange(10) # 1D array with values from 0 to 9
a1
```

Out[626…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [627…
```
a2 =np.arange(12, dtype =float).reshape(3,4) # Matrix
a2
```

Out[627…
```
array([[ 0.,  1.,  2.,  3.],
       [ 4.,  5.,  6.,  7.],
       [ 8.,  9., 10., 11.]])
```

In [628…
```
a3 = np.arange(8).reshape(2,2,2) # 3D array
a3
```

Out[628…
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

ndim

In [629…
```
print(a1.ndim)
print(a2.ndim)
print(a3.ndim)
```

```
1
2
3
```

shape

In [630…
```
print(a1.shape) # 1D array has 10 Items
print(a2.shape) # 2D array has 3 rows and 4 columns
print(a3.shape) # 3D array has 2 blocks, each containing 2 rows and 2 columns
```

```
(10,)
(3, 4)
(2, 2, 2)
```

size

In [631…
```
a3
```

Out[631…
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

In [632…
```
a3.size # Total number of elements in the 3D array
```

Out[632… 8

In [633…
```
print(a2)
print(a2.size) # Total number of elements in the 2D array
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
12
```

item size

In [634…
```
print(a1.itemsize) # Size of each element in bytes (1D array)
print(a2.itemsize) # Size of each element in bytes (2D array)
print(a3.itemsize) # Size of each element in bytes (3D array)
```

```
8
8
8
```

In [635…
```
a1
```

Out[635… array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])

In [636…
```
a1.itemsize
```

Out[636… 8

In [637…
```
a1 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=np.int32)
print(a1.itemsize)  # 4 bytes

a1 = np.array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9], dtype=np.int64)
print(a1.itemsize)  # 8 bytes
```

```
4
8
```

The default integer type in NumPy depends on your system architecture:

32-bit systems → default is int32 (4 bytes)

64-bit systems → default is int64 (8 bytes)

dtype

In [638…
```
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int64
float64
int64
```

Changing Data Type

In [639…
```
#astype

x = np.array([33, 22, 2.5]) # Implicitly dtype is float64
x
```

Out[639…
```
array([33. , 22. ,  2.5])
```

In [640…
```
print(x.dtype) # Checking the data type of the array
```

```
float64
```

In [641…
```
x.astype(int)
```

Out[641…
```
array([33, 22,  2])
```

transpose

In [642…
```
arr = np.array([[1, 2, 3],
                [4, 5, 6]])

print("Original:\n", arr)
print("Transposed:\n", arr.T)
```

```
Original:
 [[1 2 3]
 [4 5 6]]
Transposed:
 [[1 4]
 [2 5]
 [3 6]]
```

Array Operations

In [643…
```
z1 = np.arange(12).reshape(3,4)
z2 = np.arange(12,24).reshape(3,4)
```

In [644…
```
z1
```

Out[644…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [645…
```
z2
```

Out[645…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

Scalar Operations

Scalar operations in NumPy involve performing arithmetic (addition, subtraction, multiplication, division, etc.) between a single constant value (scalar) and every element of a NumPy array. This is done using broadcasting, meaning the scalar is applied to each element individually without the need for explicit loops.

In [646…
```
# arithmetic
z1 + 2
```

Out[646…
```
array([[ 2,  3,  4,  5],
       [ 6,  7,  8,  9],
       [10, 11, 12, 13]])
```

In [647…
```
# Subtraction
z1 - 2
```

Out[647…
```
array([[-2, -1,  0,  1],
       [ 2,  3,  4,  5],
       [ 6,  7,  8,  9]])
```

In [648…
```
# Multiplication
z1 * 2
```

Out[648…
```
array([[ 0,  2,  4,  6],
       [ 8, 10, 12, 14],
       [16, 18, 20, 22]])
```

In [649…
```
# power
z1 ** 2
```

Out[649…
```
array([[  0,   1,   4,   9],
       [ 16,  25,  36,  49],
       [ 64,  81, 100, 121]])
```

In [650…
```
## Modulo
z1 % 2
```

Out[650…
```
array([[0, 1, 0, 1],
       [0, 1, 0, 1],
       [0, 1, 0, 1]])
```

Relational Operators

Relational operators, also called comparison operators, are used to compare values. In NumPy, they perform element-wise comparisons on arrays and return a Boolean array (True or False) based on the condition applied to each element.

In [651…
```
z2
```

Out[651…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [652…
```
z2 > 2 # if 2 is greater than everything gives True
```

Out[652…
```
array([[ True,  True,  True,  True],
       [ True,  True,  True,  True],
       [ True,  True,  True,  True]])
```

In [653…
```
z2 > 20 # if 20 is greater than everything gives False
```

Out[653…
```
array([[False, False, False, False],
       [False, False, False, False],
       [False,  True,  True,  True]])
```

Vector Operations

Vector operations in NumPy involve performing arithmetic or mathematical operations on two NumPy arrays (vectors) of the same shape. These operations are carried out element-wise, meaning each element in one array is combined with the corresponding element in the other array.

In [654…
```
print(z1)

print(z2)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[12 13 14 15]
 [16 17 18 19]
 [20 21 22 23]]
```

In [655…
```
# Arithmetic
z1 + z2 # both numpy array Shape is same , we can add item wise
```

Out[655…
```
array([[12, 14, 16, 18],
       [20, 22, 24, 26],
       [28, 30, 32, 34]])
```

In [656…
```
z1 * z2 # both numpy array Shape is same , we can multiply item wise
```

Out[656…
```
array([[  0,  13,  28,  45],
       [ 64,  85, 108, 133],
       [160, 189, 220, 253]])
```

In [657…
```
z1 - z2 # both numpy array Shape is same , we can subtract item wise
```

Out[657…
```
array([[-12, -12, -12, -12],
       [-12, -12, -12, -12],
       [-12, -12, -12, -12]])
```

In [658…
```
z1 / z2 # both numpy array Shape is same , we can divide item wise
```

Out[658…
```
array([[0.        , 0.07692308, 0.14285714, 0.2       ],
       [0.25      , 0.29411765, 0.33333333, 0.36842105],
       [0.4       , 0.42857143, 0.45454545, 0.47826087]])
```

Array Functions

In [659…
```
k1 = np.random.random((3,3))
k1 = np.round(k1*100)
k1
```

Out[659…
```
array([[65., 90., 84.],
       [78., 24., 72.],
       [67., 14., 46.]])
```

In [660…
```
# Max
np.max(k1) # Finding the maximum value in the array
```

Out[660…
```
np.float64(90.0)
```

In [661…
```
# min
np.min(k1)
```

Out[661…
```
np.float64(14.0)
```

In [662…
```
# sum
np.sum(k1) # Finding the sum of all elements in the array
```

Out[662… np.float64(540.0)

In [663…
```
# prod ----> Multiplication
np.prod(k1) # Finding the product of all elements in the array
```

Out[663… np.float64(2857815339724800.0)

In NumPy

0 = Columns, 1 = Rows

In [664…
```
# if we want maximum of every row

np.max(k1, axis = 1)
```

Out[664… array([90., 78., 67.])

In [665…
```
# maximum of every column

np.max(k1, axis = 0)
```

Out[665… array([78., 90., 84.])

In [666…
```
# product of every column

np.prod(k1, axis = 0)
```

Out[666… array([339690., 30240., 278208.])

Statistics related functions

In [667…
```
# mean
k1 # Finding the mean of every row
```

Out[667…
```
array([[65., 90., 84.],
       [78., 24., 72.],
       [67., 14., 46.]])
```

In [668…
```
np.mean(k1) # Finding the mean of all elements in the array
```

Out[668… np.float64(60.0)

In [669…
```
# mean of every column

k1.mean(axis=0) # Finding the mean of every column
```

Out[669… array([70. , 42.66666667, 67.33333333])

In [670…
```
# median

np.median(k1) #Finding the median of all elements in the array
```

Out[670… np.float64(67.0)

In [671…
```
np.median(k1, axis = 1) # Finding the median of every row
```

Out[671…
```
array([84., 72., 46.])
```

In [672…
```
# Standard deviation

np.std(k1) # Standard deviation of all elements in the array
```

Out[672…
```
np.float64(25.002222123465568)
```

In [673…
```
np.std(k1, axis =0) # Standard deviation of every column
```

Out[673…
```
array([ 5.71547607, 33.71778298, 15.860503  ])
```

In [674…
```
# variance
np.var(k1) # Finding the variance of all elements in the array
```

Out[674…
```
np.float64(625.1111111111111)
```

Trignometry Functions

In [675…
```
np.sin(k1) # Finding the sin of all elements in the array
```

Out[675…
```
array([[ 0.82682868,  0.89399666,  0.73319032],
       [ 0.51397846, -0.90557836,  0.25382336],
       [-0.85551998,  0.99060736,  0.90178835]])
```

In [676…
```
np.cos(k1) # Finding the cos of all elements in the array
```

Out[676…
```
array([[-0.56245385, -0.44807362, -0.6800235 ],
       [-0.85780309,  0.42417901, -0.96725059],
       [-0.5177698 ,  0.13673722, -0.43217794]])
```

In [677…
```
np.tan(k1) # Finding the tan of all elements in the array
```

Out[677…
```
array([[-1.47003826, -1.99520041, -1.07818381],
       [-0.59918   , -2.1348967 , -0.26241738],
       [ 1.65231726,  7.24460662, -2.08661353]])
```

Dot Product

In NumPy, the dot() function is used to compute the dot product of two arrays. The meaning of the dot product depends on the dimensions of the inputs:

- 1D arrays (vectors): Returns the sum of the products of corresponding elements (scalar result).
- 2D arrays (matrices): Performs matrix multiplication.
- 1D and 2D combination: Performs multiplication of a vector by a matrix or vice versa.

In [678…
```
s2 = np.arange(12).reshape(3,4)
s3 = np.arange(12,24).reshape(4,3)
```

In [679…
```
s2
```

Out[679…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [680…
```
s3
```

Out[680…
```
array([[12, 13, 14],
       [15, 16, 17],
       [18, 19, 20],
       [21, 22, 23]])
```

In [681…
```
np.dot(s2,s3) # Performing dot product between s2 and s3
```

Out[681…
```
array([[114, 120, 126],
       [378, 400, 422],
       [642, 680, 718]])
```

Log and Exponents

In NumPy, you can compute the logarithm and exponentiation of arrays using the log() and exp() functions, respectively.

In [682…
```
np.exp(s2) # Finding the exponential of all elements in the array
```

Out[682…
```
array([[1.00000000e+00, 2.71828183e+00, 7.38905610e+00, 2.00855369e+01],
       [5.45981500e+01, 1.48413159e+02, 4.03428793e+02, 1.09663316e+03],
       [2.98095799e+03, 8.10308393e+03, 2.20264658e+04, 5.98741417e+04]])
```

round() / floor() / ceil()

- round() Rounds values to the nearest integer (or to a specified number of decimals).
- floor() Rounds values down to the nearest integer.
- ceil() Rounds values up to the nearest integer.

In [683…
```
# Round to the nearest integer
arr = np.array([1.2, 2.7, 3.5, 4.9])
rounded_arr = np.round(arr) # Rounds values to the nearest integer
print(rounded_arr)
```

```
[1. 3. 4. 5.]
```

In [684…
```
# Round to two decimals
arr = np.array([1.234, 2.567, 3.891])
rounded_arr = np.round(arr, decimals=2)  # Rounds values to two decimal places
print(rounded_arr)
```

```
[1.23 2.57 3.89]
```

In [685…
```
#randomly
np.round(np.random.random((2,3))*100) # Rounds random values to nearest integer
```

Out[685…
```
array([[13., 17., 61.],
       [44., 28., 22.]])
```

floor()

In [686…
```
# Floor operation
arr = np.array([1.2, 2.7, 3.5, 4.9])
```

```
floored_arr = np.floor(arr) # Rounds values down to the nearest integer
print(floored_arr)
```

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

In [687…
```
np.floor(np.random.random((2,3))*100) # Rounds random values down to the nearest integer
```

Out[687…
```
array([[54., 28., 32.],
       [67., 84., 41.]])
```

ceil()

In [688…
```
arr = np.array([1.2, 2.7, 3.5, 4.9])
ceiled_arr = np.ceil(arr) # Rounds values up to the nearest integer
print(ceiled_arr)
```

```
[2. 3. 4. 5.]
```

In [689…
```
np.ceil(np.random.random((2,3))*100) # gives highest integer ex : 7.8 = 8
```

Out[689…
```
array([[75., 54., 43.],
       [ 2., 31., 21.]])
```

Indexing and Slicing

In [690…
```
p1 = np.arange(10)
p2 = np.arange(12).reshape(3,4)
p3 = np.arange(8).reshape(2,2,2)
```

In [691…
```
p1
```

Out[691…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [692…
```
p2
```

Out[692…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [693…
```
p3
```

Out[693…
```
array([[[0, 1],
        [2, 3]],

       [[4, 5],
        [6, 7]]])
```

Indexing on 1D array

In [694…
```
p1
```

Out[694…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [695…
```
p1[-1] # Accessing the last element
```

Out[695…
```
np.int64(9)
```

In [696…
```
p1[0] # Accessing the first element
```

Out[696… np.int64(0)

Indexing on 2D array

In [697…
```
p2
```

Out[697…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [698…
```
p2[1,2] # Accessing the element at row 1, column 2 in p2
```

Out[698… np.int64(6)

In [699…
```
p2[2,3] # Accessing the element at row 2, column 3 in p2
```

Out[699… np.int64(11)

In [700…
```
p2[1,0] # Accessing the element at row 1, column 0 in p2
```

Out[700… np.int64(4)

indexing on 3D (Tensors)

In [701…
```
p3[1, 0, 1] # Accessing the element at depth 1, row 0, column 1 in p3
```

Out[701… np.int64(5)

EXPLANATION :Here 3D is consists of 2 ,2D array , so Firstly we take 1 because our desired is 5 is in second matrix which is 1 .and 1 row so 0 and second column so 1

In [702…
```
p3[0,1,0] # Accessing the element at depth 0, row 1, column 0 in p3
```

Out[702… np.int64(2)

EXPLANATION :Here firstly we take 0 because our desired is 2, is in first matrix which is 0 . and 2 row so 1 and first column so 0

In [703…
```
p3[0,0,0] # Accessing the element at depth 0, row 0, column 0 in p3
```

Out[703… np.int64(0)

Here first we take 0 because our desired is 0, is in first matrix which is 0 . and 1 row so 0 and first column so 0

In [704…
```
p3[1,1,0] # Accessing the element at depth 1, row 1, column 0 in p3
```

Out[704… np.int64(6)

EXPLANATION : Here first we take because our desired is 6, is in second matrix which is 1 . and second row so 1 and first column so 0

Slicing

- Fetching Multiple items

Slicing on 1D

In [705…
```
p1
```

Out[705…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [706…
```
p1[2:5] # Slicing the array from index 2 to 4 (5 is exclusive)
```

Out[706…
```
array([2, 3, 4])
```

EXPLANATION :Here First we take , whatever we need first item ,2 and up last(4) + 1 which 5 .because last element is not included

In [707…
```
p1[2:5:2] # Slicing the array from index 2 to 4 with a step size of 2
```

Out[707…
```
array([2, 4])
```

Slicing on 2D

In [708…
```
p2
```

Out[708…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [709…
```
p2[0, :] # Accessing all columns of the first row in p2
```

Out[709…
```
array([0, 1, 2, 3])
```

EXPLANATION :Here 0 represents first row and (:) represnts Total column

In [710…
```
p2[:,2] # Accessing all rows of the second column in p2
```

Out[710…
```
array([ 2,  6, 10])
```

EXPLANATION :Here we want all rows so (:) , and we want 3rd column so 2

In [711…
```
p2 # Accessing all rows of the second column in the second matrix of p2
```

Out[711…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [712…
```
p2[1:3] # Slicing the array from row 1 to row 2 (3 is exclusive)
```

Out[712…
```
array([[ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [713…
```
p2[1:3 ,1:3] # Slicing the array from row 1 to row 2 and column 1 to column 2 (3 is exclusive)
```

Out[713…
```
array([[ 5,  6],
       [ 9, 10]])
```

EXPLANATION :Here first [1:3] we slice 2 second row is to third row is not existed which is 2 and Secondly , we take [1:3] which is same as first:we slice 2 second row is to third row is not included which is 3

In [714…
```
p2
```

Out[714…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [715…
```
p2[::2, ::3] # Slicing the array to access every second row and every third column
```

Out[715…
```
array([[ 0,  3],
       [ 8, 11]])
```

EXPLANATION : Here we take (:) because we want all rows , second(:2) for alternate value, and (:) for all columns and (:3) jump for two steps

In [716…
```
p2
```

Out[716…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [717…
```
p2[::2] # Slicing the array to access every second row
```

Out[717…
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11]])
```

In [718…
```
p2[::2 ,1::2] # Slicing the array to access every second row and every second column starting from the first column
```

Out[718…
```
array([[ 1,  3],
       [ 9, 11]])
```

EXPLANATION : Here we take (:) because we want all rows , second(:2) for alternate value, and (1) for we want from second column and (:2) jump for two steps and ignore middle one

In [719…
```
p2
```

Out[719…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [720…
```
p2[1]
```

Out[720…
```
array([4, 5, 6, 7])
```

In [721…
```
p2[1,::3]
```

Out[721…
```
array([4, 7])
```

EXPLANATION : Here we take (1) because we want second row , second(:) for total column, (:3) jump for two steps and ignore middle ones

In [722…
```
p2
```

Out[722…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [723…
```
p2[0:2] # Slicing the array to access rows 0 and 1
```

Out[723…
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [724…
```
p2[0:2 ,1: ] # Slicing the array to access rows 0 and 1 and all columns starting from the first column
```

Out[724…
```
array([[1, 2, 3],
       [5, 6, 7]])
```

In [725…
```
p2
```

Out[725…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [726…
```
p2[0:2] # Slicing the array to access rows 0 and 1
```

Out[726…
```
array([[0, 1, 2, 3],
       [4, 5, 6, 7]])
```

In [727…
```
p2[0:2 ,1::2] # Slicing the array to access rows 0 and 1 and every second column starting from the first column
```

Out[727…
```
array([[1, 3],
       [5, 7]])
```

EXPLANATION : 0:2 selects the rows from index 0 (inclusive) to index 2 (exclusive), which means it will select the first and second rows of the array. , is used to separate row and column selections. 1::2 selects the columns starting from index 1 and selects every second column. So it will select the second and fourth columns of the array.

Slicing in 3D

In [728…
```
p3 = np.arange(27).reshape(3,3,3)
p3
```

Out[728…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [729…
```
p3[1] # Accessing the second 2D array (slice) from the 3D array
```

Out[729…
```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [730…
```
p3[::2] # Slicing the array to access every second 2D array (slice) from the 3D array
```

Out[730…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

EXPLANATION : Along the first axis, (::2) selects every second element. This means it will select the subarrays at indices 0 and 2

In [731…
```
p3
```

Out[731…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [732…
```
p3[0] # Accessing the first 2D array (slice) from the 3D array
```

Out[732…
```
array([[0, 1, 2],
       [3, 4, 5],
       [6, 7, 8]])
```

In [733…
```
p3[0,1,:] # Accessing the second row of the first 2D array (slice) from the 3D array
```

Out[733…
```
array([3, 4, 5])
```

EXPLANATION : 0 represnts first matrix , 1 represents second row , (:) means total columns in that row

In [734…
```
p3
```

Out[734…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [735…
```
p3[1]
```

Out[735…
```
array([[ 9, 10, 11],
       [12, 13, 14],
       [15, 16, 17]])
```

In [736…
```
p3[1,:,1] # Accessing the second row of the second 2D array (slice) from the 3D array
```

Out[736…
```
array([10, 13, 16])
```

EXPLANATION : 1 represents middle column , (:) all columns , 1 represents middle column

In [737…
```
p3
```

Out[737…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [738…
```
p3[2] # Accessing the third 2D array (slice) from the 3D array
```

Out[738…
```
array([[18, 19, 20],
       [21, 22, 23],
       [24, 25, 26]])
```

In [739…
```
p3[2, 1: ]# Accessing the second row and all columns of the third 2D array (slice) from the 3D array
```

Out[739…
```
array([[21, 22, 23],
       [24, 25, 26]])
```

In [740…
```
p3[2, 1: ,1:] # Accessing the second row and all columns starting from the second column of the third 2D array (slice) from the 3D array
```

Out[740…
```
array([[22, 23],
       [25, 26]])
```

EXPLANATION : Here we go through 3 stages , where 2 for last array , and (1:) from second row to total rows , and (1:) is for second column to total columns

In [741…
```
p3
```

Out[741…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [742…
```
p3[0::2] # Slicing the array to access every second 2D array (slice) starting from the first one
```

Out[742…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [743…
```
p3[0::2 , 0] # Accessing the first column of every second 2D array (slice) starting from the first one in the 3D array
```

Out[743…
```
array([[ 0,  1,  2],
       [18, 19, 20]])
```

In [744…
```
p3[0::2 , 0 , ::2] # Accessing the first column of every second 2D array (slice) starting from the first one in the 3D array and all rows starting from the first one
```

Out[744…
```
array([[ 0,  2],
       [18, 20]])
```

EXPLANATION : Here we take (0::2) first and last column , so we did jump using this, and we took (0) for first row , and we (::2) ignored middle column

Iterating

In [745…
```
p1
```

Out[745…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [746…
```
for i in p1: # Accessing each element in the 1D array
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
```

In [747…
```
p2
```

Out[747…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [748…
```
for i in p2: # Accessing each row in the 2D array
    print(i) # prints rows
```

```
[0 1 2 3]
[4 5 6 7]
[ 8  9 10 11]
```

In [749…
```
p3
```

Out[749…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [750…
```
for i in p3: # Accessing each element in the 3D array
    print(i)
```

```
[[0 1 2]
 [3 4 5]
 [6 7 8]]
[[ 9 10 11]
 [12 13 14]
 [15 16 17]]
[[18 19 20]
 [21 22 23]
 [24 25 26]]
```

In [751…
```
for i in np.nditer(p3): # Iterating through each element in the 3D array
    print(i)
```

```
0
1
2
3
4
5
6
7
8
9
10
11
12
13
14
15
16
17
18
19
20
21
22
23
24
25
26
```

Reshaping

- **Transpose** - Converts rows in to columns and columns into rows

In [752…
```
p2
```

Out[752…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [753…
```
np.transpose(p2) # Transposing the 2D array p2
```

Out[753…
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [754…
```
# Another method
```

```
p2.T # Transposing the 2D array p2
```

Out[754…
```
array([[ 0,  4,  8],
       [ 1,  5,  9],
       [ 2,  6, 10],
       [ 3,  7, 11]])
```

In [755…
```
p3
```

Out[755…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [756…
```
p3.T # Transposing the 3D array p3
```

Out[756…
```
array([[[ 0,  9, 18],
        [ 3, 12, 21],
        [ 6, 15, 24]],

       [[ 1, 10, 19],
        [ 4, 13, 22],
        [ 7, 16, 25]],

       [[ 2, 11, 20],
        [ 5, 14, 23],
        [ 8, 17, 26]]])
```

- ravel() is a method that converts an array of any shape (any number of dimensions) into a 1-dimensional array.

In [757…
```
p2
```

Out[757…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [758…
```
p2.ravel() # Flattening the 2D array p2
```

Out[758…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11])
```

In [759…
```
p3
```

Out[759…
```
array([[[ 0,  1,  2],
        [ 3,  4,  5],
        [ 6,  7,  8]],

       [[ 9, 10, 11],
        [12, 13, 14],
        [15, 16, 17]],

       [[18, 19, 20],
        [21, 22, 23],
        [24, 25, 26]]])
```

In [760…
```
p3.ravel() # Flattening the 3D array p3
```

Out[760…
```
array([ 0,  1,  2,  3,  4,  5,  6,  7,  8,  9, 10, 11, 12, 13, 14, 15, 16,
       17, 18, 19, 20, 21, 22, 23, 24, 25, 26])
```

stacking means joining two or more arrays together along a new axis. The arrays you stack must have the same shape (except in the dimension where stacking happens).

- **Horizontal stacking** – joins arrays side-by-side (hstack())
- **Vertical stacking** – joins arrays top-to-bottom (vstack())

In [761…
```
# Horizontal stacking

w1 = np.arange(12).reshape(3,4)
w2 = np.arange(12,24).reshape(3,4)
```

In [762…
```
w1
```

Out[762…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [763…
```
w2
```

Out[763…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [764…
```
np.hstack((w1,w2)) # Horizontal stacking of w1 and w2
```

Out[764…
```
array([[ 0,  1,  2,  3, 12, 13, 14, 15],
       [ 4,  5,  6,  7, 16, 17, 18, 19],
       [ 8,  9, 10, 11, 20, 21, 22, 23]])
```

In [765…
```
w1
```

Out[765…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [766…
```
w2
```

Out[766…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [767…
```
np.vstack((w1,w2)) # Vertical stacking of w1 and w2
```

Out[767…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

splitting is basically the opposite of stacking. It means breaking one array into multiple smaller arrays along a given axis.

- splitting is basically the opposite of stacking.
- **split()** – split into equal parts (or specified indices)
- **hsplit()** – horizontal split (columns)
- **vsplit()** – vertical split (rows)

In [768…
```
w1
```

Out[768…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11]])
```

In [769…
```
np.hsplit(w1,2) # Splitting w1 into 2 equal parts horizontally
```

Out[769…
```
[array([[0, 1],
        [4, 5],
        [8, 9]]),
 array([[ 2,  3],
        [ 6,  7],
        [10, 11]])]
```

In [770…
```
np.hsplit(w1,4) # Splitting w1 into 4 equal parts horizontally
```

Out[770…
```
[array([[0],
        [4],
        [8]]),
 array([[1],
        [5],
        [9]]),
 array([[ 2],
        [ 6],
        [10]]),
 array([[ 3],
        [ 7],
        [11]])]
```

In [771…
```
w2
```

Out[771…
```
array([[12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [772…
```
np.vsplit(w2,3) # Splitting w2 into 3 equal parts vertically
```

Out[772…
```
[array([[12, 13, 14, 15]]),
 array([[16, 17, 18, 19]]),
 array([[20, 21, 22, 23]])]
```

NumPy Arrays vs Python Sequences (like lists)

- **Fixed size:** Once you create a NumPy array, its size can't change. Python lists can grow or shrink, but resizing a NumPy array makes a new array and deletes the old one.
- **Same data type:** All elements in a NumPy array must have the same type (e.g., all integers, all floats), so they take up the same amount of memory.
- **Fast math operations:** NumPy arrays are built for handling large amounts of data quickly, especially for math and scientific work. They can do more with less code compared to Python lists.
- **Used by many libraries:** Most scientific and math libraries in Python use NumPy arrays. Even if they accept lists, they often convert them into NumPy arrays before doing calculations, and usually give results back as NumPy arrays.

Speed of List Vs Numpy

- **List**

In [773…
```python
a = [i for i in range(10000000)]     # Create list 'a' with numbers from 0 to 9,999,999
b = [i for i in range(10000000, 20000000)]  # Create list 'b' with numbers from 10,000,000 to 19,999,999

c = []  # Empty list to store the sum of elements from 'a' and 'b'

import time  # Import time module to measure execution time

start = time.time()  # Record start time
for i in range(len(a)):
    c.append(a[i] + b[i])  # Add corresponding elements from 'a' and 'b' and store in 'c'

print(time.time() - start)  # Print the total time taken for the loop
```

```
2.8916499614715576
```

- **NumPy**

In [774…
```python
import numpy as np       # Import NumPy library for fast numerical operations
import time              # Import time module to measure execution time

a = np.arange(10000000)              # Create NumPy array with values 0 to 9,999,999
b = np.arange(10000000, 20000000)     # Create NumPy array with values 10,000,000 to 19,999,999

start = time.time()       # Record start time
c = a + b                 # Vectorized addition of arrays 'a' and 'b'
print(time.time() - start)  # Print the time taken for the operation
```

```
0.19583630561828613
```

In [775…
```python
2.7065064907073975 / 0.02248692512512207
```

Out[775…
```
120.35911871666826
```

- So, NumPy is faster than normal Python list operations, as we can see in the above example, because NumPy uses optimized C-based arrays and vectorized operations.

Memory Used for List Vs NumPy

- **List**

In [776…
```python
P = [i for i in range(10000000)]  # Create a list 'P' containing numbers from 0 to 9,999,999
import sys  # Import sys module to access system-specific parameters and functions
sys.getsizeof(P)  # Get the size of list 'P' in bytes (only the list object, not the size of all elements)
```

Out[776…
```
89095160
```

- **NumPy**

In [777…
```
R = np.arange(10000000)   # Create a NumPy array with values from 0 to 9,999,999
sys.getsizeof(R)          # Get the memory size (in bytes) used by the NumPy array object
```

Out[777… 80000112

In [778…
```
# We can reduce memory usage further by using NumPy with a smaller data type (int16 here)
R = np.arange(10000000, dtype=np.int16)  # Create a NumPy array from 0 to 9,999,999 using 16-bit integers
sys.getsizeof(R)  # Get the memory size of the array in bytes
```

Out[778… 20000112

Advance Indexing and Slicing

In [779…
```
# Normal Indexing and Slicing Example

w = np.arange(12).reshape(4, 3)  # Create a NumPy array with values from 0 to 11 and reshape it into 4 rows and 3 columns
w  # Display the array
```

Out[779…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [780…
```
w[1,2] # Fetching 5 from array
```

Out[780… np.int64(5)

In [781…
```
w[1:3] # Fetching 4,5,7,8
```

Out[781…
```
array([[3, 4, 5],
       [6, 7, 8]])
```

In [782…
```
w[1:3 , 1:3] #
```

Out[782…
```
array([[4, 5],
       [7, 8]])
```

Fancy Indexing

- Fancy indexing lets you pick or change specific elements in an array using lists of indices or complex conditions.
- It's a powerful NumPy feature for working with and modifying array data in flexible ways.

In [783…
```
w # Create a NumPy array with values from 0 to 11 and reshape it into 4 rows and 3 columns
```

Out[783…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [784…
```
# Fetch 1,3,4 row

w[[0,2,3]] # Fetch 1,3,4 row
```

Out[784…
```
array([[ 0,  1,  2],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [785…
```
# New array

z = np.arange(24).reshape(6,4)
z
```

Out[785…
```
array([[ 0,  1,  2,  3],
       [ 4,  5,  6,  7],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [16, 17, 18, 19],
       [20, 21, 22, 23]])
```

In [786…
```
# Fetch 1, 3, ,4, 6 rows

z[[0,2,3,5]]
```

Out[786…
```
array([[ 0,  1,  2,  3],
       [ 8,  9, 10, 11],
       [12, 13, 14, 15],
       [20, 21, 22, 23]])
```

In [787…
```
# Fetch 1,3,4 columns

z[:,[0,2,3]]
```

Out[787…
```
array([[ 0,  2,  3],
       [ 4,  6,  7],
       [ 8, 10, 11],
       [12, 14, 15],
       [16, 18, 19],
       [20, 22, 23]])
```

Boolean Indexing

- Boolean indexing lets you pick elements from an array based on True/False conditions.
- It helps you filter and extract only the elements that meet a specific condition, so you can easily work with a selected part of your data.

In [788…
```
G = np.random.randint(1,100,24).reshape(6,4) # Create a 6x4 array with random integers between 1 and 100
```

In [789…
```
# find all numbers greater than 50
G > 50 # Boolean Indexing
```

Out[789…
```
array([[ True,  True, False, False],
       [False,  True,  True,  True],
       [ True, False,  True, False],
       [False,  True,  True, False],
       [ True,  True, False, False],
       [False, False, False,  True]])
```

In [790…
```
# Where is True , it gives result , everything other that removed.we got value
G[G > 50] # Boolean Indexing
```

Out[790…
```
array([53, 85, 60, 58, 95, 89, 58, 73, 86, 60, 53, 84], dtype=int32)
```

It is the best technique to filter data based on given conditions.

In [791…
```
# find out even numbers
G % 2 == 0 #True/False
```

Out[791…
```
array([[False, False, False, False],
       [ True,  True,  True, False],
       [False, False,  True, False],
       [ True, False,  True,  True],
       [ True, False, False,  True],
       [ True, False, False,  True]])
```

In [792…
```
# Gives only the even numbers
# G = np.random.randint(1,100,24).reshape(6,4) # Create a 6x4 array with random integers between 1 and 100
G [ G % 2 == 0] #False are ignored, Gives only the even numbers
```

Out[792…
```
array([46, 60, 58, 58, 28, 86, 46, 60, 44, 32, 84], dtype=int32)
```

In [793…
```
# find all numbers greater than 50 and are even
(G > 50 ) & (G % 2 == 0) #Boolean Indexing , True/False
```

Out[793…
```
array([[False, False, False, False],
       [False,  True,  True, False],
       [False, False,  True, False],
       [False, False,  True, False],
       [ True, False, False, False],
       [False, False, False,  True]])
```

Here we used (&) bitwise Not logical (and), because we are working with boolean values

In [794…
```
# find all numbers greater than 50 and are even
G [(G > 50 ) & (G % 2 == 0)] #Boolean Indexing
```

Out[794…
```
array([60, 58, 58, 86, 60, 84], dtype=int32)
```

In [795…
```
# find all numbers not divisible by 7
G % 7 == 0 # False are ignored , gives only the numbers divisible by 7
```

Out[795…
```
array([[False, False,  True, False],
       [False, False, False, False],
       [False, False, False, False],
       [ True, False, False, False],
       [False, False,  True, False],
       [False, False, False,  True]])
```

In [796…
```
# find all numbers not divisible by 7
G[~(G % 7 == 0)] # (~) = Not operator
```

Out[796…
```
array([53, 85, 27, 46, 60, 58, 95, 89, 39, 58, 17, 73, 86, 46, 60, 53, 44,
       32, 19, 45], dtype=int32)
```

Broadcasting in NumPy

(Used in Vectorization)

Broadcasting is a technique that allows **NumPy** to work with arrays of different shapes when performing arithmetic operations.

- If the arrays have **different shapes**, NumPy automatically **stretches the smaller array** to match the larger one.
- This makes operations **faster** and **more memory-efficient** because we avoid writing loops.

In [797…
```
# same shape
a = np.arange(6).reshape(2,3) # 2x3 array
b = np.arange(6,12).reshape(2,3) # 2x3 array

print(a)
print(b)

print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
[[ 6  8 10]
 [12 14 16]]
```

In [798…
```
# diff shape
a = np.arange(6).reshape(2,3) # 2x3
b = np.arange(3).reshape(1,3) # 1x3

print(a)
print(b)

print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[0 1 2]]
[[0 2 4]
 [3 5 7]]
```

Broadcasting Rules in NumPy

Broadcasting allows **arrays of different shapes** to be used together in arithmetic operations.
Here are the simple rules:

1 Make the arrays have the same number of dimensions

- If the arrays have different numbers of dimensions, **add new dimensions of size 1** to the **front** of the smaller array.

Examples:

- `(3, 2)` → 2D, `(3,)` → 1D → Convert `(3,)` to `(1, 3)`
- `(3, 3, 3)` → 3D, `(3,)` → 1D → Convert `(3,)` to `(1, 1, 3)`

2 Make each dimension match in size

- If the size of a dimension does not match, **stretch dimensions with size 1** to match the other array.

Example:

- `(3, 3)` → 2D, `(3,)` → 1D → Convert `(3,)` to `(1, 3)` → Stretch to `(3, 3)`

- If a dimension has **size not equal to 1** in either array and they don't match, broadcasting **fails** and an **error** is raised.

Broadcasting allows NumPy to perform operations **without manually reshaping arrays** every time.

In [799…
```
# More examples
a = np.arange(12).reshape(4,3) # 2 D
b = np.arange(3) # 1 D
print(a)
print(b)
print(a+b) # 2 D
```

```
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
[0 1 2]
[[ 0  2  4]
 [ 3  5  7]
 [ 6  8 10]
 [ 9 11 13]]
```

EXPLANATION : Arthematic Operation possible because , Here a = (4,3) is 2D and b =(3) is 1D so did converted (3) to (1,3) and streched to (4,3)

In [800…
```
# Could not Broadcast

a = np.arange(12).reshape(3,4) # 2 D
b = np.arange(3) # 1 D

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[0 1 2]
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[800], line 9
      6 print(a)
      7 print(b)
----> 9 print(a+b)

ValueError: operands could not be broadcast together with shapes (3,4) (3,)
```

EXPLANATION : Arthematic Operation not possible because , Here a = (3,4) is 2D and b =(3) is 1D so did converted (3) to (1,3) and streched to (3,3) but , a is not equals to b . so it got failed

In [801…
```
a = np.arange(3).reshape(1,3) #
b = np.arange(3).reshape(3,1)

print(a)
print(b)

print(a+b)
```

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

EXPLANATION : Arthematic Operation possible because , Here a = (1,3) is 2D and b =(3,1) is 2D so did converted (1,3) to (3,3) and b(3,1) convert (1)to 3 than (3,3) . finally it equally.

In [802…
```
a = np.arange(3).reshape(1,3)
b = np.arange(4).reshape(4,1)

print(a)
print(b)

print(a + b)
```

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

EXPLANATION : Same as before

In [803…
```
a = np.array([1])
# shape -> (1,1) streched to 2,2 # 1 -> 2 , 1 -> 2
b = np.arange(4).reshape(2,2) # 4 -> 2 , 4 -> 2
# shape -> (2,2) # 2 -> 2 , 2 -> 2

print(a)
print(b)

print(a+b)
```

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

In [804…
```
# doesnt work

a = np.arange(12).reshape(3,4)
b = np.arange(12).reshape(4,3)

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
[[ 0  1  2]
 [ 3  4  5]
 [ 6  7  8]
 [ 9 10 11]]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[804], line 9
      6 print(a)
      7 print(b)
----> 9 print(a+b)

ValueError: operands could not be broadcast together with shapes (3,4) (4,3)
```

EXPLANATION : there is no 1 to convert ,so got failed

In [805…
```
# Not Work
a = np.arange(16).reshape(4,4)
b = np.arange(4).reshape(2,2)

print(a)
print(b)

print(a+b)
```

```
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]
 [12 13 14 15]]
[[0 1]
 [2 3]]
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
Cell In[805], line 8
      5 print(a)
      6 print(b)
----> 8 print(a+b)

ValueError: operands could not be broadcast together with shapes (4,4) (2,2)
```

EXPLANATION : there is no 1 to convert ,so got failed

Working with Mathematical Formulas

In [806…
```
k = np.arange(10) # 1 D
```

In [807…
```
k
```

Out[807…
```
array([0, 1, 2, 3, 4, 5, 6, 7, 8, 9])
```

In [808…
```
np.sum(k) # 1 D # Sum of all elements
```

Out[808…
```
np.int64(45)
```

In [809…
```
np.sin(k) #sin of all elements
```

Out[809…
```
array([ 0.        ,  0.84147098,  0.90929743,  0.14112001, -0.7568025 ,
       -0.95892427, -0.2794155 ,  0.6569866 ,  0.98935825,  0.41211849])
```

Sigmoid Function

The **Sigmoid function** is a mathematical function that maps any real number to a value between **0 and 1**.
It is commonly used in **machine learning**, especially for **binary classification**.

🔹 Formula

[sigma(x) = frac{1}/{1 + e^{-x}}]

Where:

- (x) = input value
- (e) = Euler's number (~2.718)

In [810…
```
def sigmoid(array):  # Define a function that takes an array as input
 return 1/(1+np.exp(-(array)))  # Calculate the sigmoid of the array
k = np.arange(10)  # Create an array of 10 elements
sigmoid(k)  # Call the sigmoid function with the array
```

Out[810…
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661])
```

In [811…
```
k = np.arange(100) # Create an array of 100 elements
sigmoid(k) # Apply the sigmoid function to the array
```

Out[811…
```
array([0.5       , 0.73105858, 0.88079708, 0.95257413, 0.98201379,
       0.99330715, 0.99752738, 0.99908895, 0.99966465, 0.99987661,
       0.9999546 , 0.9999833 , 0.99999386, 0.99999774, 0.99999917,
       0.99999969, 0.99999989, 0.99999996, 0.99999998, 0.99999999,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ,
       1.        , 1.        , 1.        , 1.        , 1.        ])
```

Mean Squared Error (MSE)

Mean Squared Error (MSE) is a way to measure how far off our predictions are from the actual values.

It is calculated as:

[{MSE} = frac{1}{n} sum_{i=1}^{n} (y_i - hat{y}_i)^2]

Where:

- (y_i) = Actual value
- (hat{y}_i) = Predicted value
- (n) = Total number of data points

Key points:

- MSE is always **non-negative**.
- Smaller MSE → predictions are **closer** to actual values.
- Squaring the errors ensures that **positive and negative differences don't cancel out**.

In [812…
```
actual = np.random.randint(1,50,25) # Actual values
predicted = np.random.randint(1,50,25) # Predicted values
```

In [813…
```
actual  # Actual values
```

Out[813…
```
array([36, 34, 15, 36, 27, 30, 32, 32, 37, 42, 36, 27, 38, 11, 26, 37,  1,
       29, 32, 11, 25,  3, 48, 13, 19], dtype=int32)
```

In [814…
```
predicted # Predicted values
```

Out[814…
```
array([18, 36, 26, 15, 47, 32,  7,  7,  4, 47,  5, 43, 22, 36, 40, 38, 21,
       33, 38, 45, 25, 17, 29, 20, 37], dtype=int32)
```

In [815…
```
actual = np.random.randint(1,50,25) # Actual values
predicted = np.random.randint(1,50,25) # Predicted values

def mse(actual,predicted): # Define a function that takes actual and predicted values as input
    return np.mean((actual-predicted)**2) # Calculate the mean squared error
print(mse(actual,predicted)) # Call the function with actual and predicted values
```

```
315.24
```

In [816…
```
# detailed
actual-predicted
```

Out[816…
```
array([-27,   1,  -1, -22,   4, -13, -18,  13,  20, -10,  13,  -1,  -2,
       -14, -23,   8,  10,  13, -22,  -9,  31, -31, -28, -27,  16],
      dtype=int32)
```

In [817…
```
(actual-predicted)**2 # Square the difference
```

Out[817…
```
array([729,   1,   1, 484,  16, 169, 324, 169, 400, 100, 169,   1,   4,
       196, 529,  64, 100, 169, 484,  81, 961, 961, 784, 729, 256],
      dtype=int32)
```

In [818…
```
np.mean((actual-predicted)**2) # Calculate the mean squared error
```

Out[818…
```
np.float64(315.24)
```

Working with Missing Values

In [819…
```
# Working with missing values -> np.nan

S = np.array([1,2,3,4,np.nan,6]) # Create an array with missing values
S
```

Out[819…
```
array([ 1.,  2.,  3.,  4., nan,  6.])
```

In [820…
```
np.isnan(S) # Check for missing values
```

Out[820…
```
array([False, False, False, False,  True, False])
```

In [821…
```
S[np.isnan(S)] # Nan values
```

Out[821…
```
array([nan])
```

In [822…
```
S[~np.isnan(S)] # Not Nan Values
```

Out[822…
```
array([1., 2., 3., 4., 6.])
```

Plotting Graphs

In [823…
```
# plotting a 2D plot #
# x = y
x = np.linspace(-10, 10, 100)  # Create an array of 100 evenly spaced numbers from -10 to 10
x
```

Out[823…
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
       -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
       -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
       -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
       -6.76767677,  -6.56565657,  -6.36363636,  -6.16161616,
       -5.95959596,  -5.75757576,  -5.55555556,  -5.35353535,
       -5.15151515,  -4.94949495,  -4.74747475,  -4.54545455,
       -4.34343434,  -4.14141414,  -3.93939394,  -3.73737374,
       -3.53535354,  -3.33333333,  -3.13131313,  -2.92929293,
       -2.72727273,  -2.52525253,  -2.32323232,  -2.12121212,
       -1.91919192,  -1.71717172,  -1.51515152,  -1.31313131,
       -1.11111111,  -0.90909091,  -0.70707071,  -0.50505051,
       -0.3030303 ,  -0.1010101 ,   0.1010101 ,   0.3030303 ,
        0.50505051,   0.70707071,   0.90909091,   1.11111111,
        1.31313131,   1.51515152,   1.71717172,   1.91919192,
        2.12121212,   2.32323232,   2.52525253,   2.72727273,
        2.92929293,   3.13131313,   3.33333333,   3.53535354,
        3.73737374,   3.93939394,   4.14141414,   4.34343434,
        4.54545455,   4.74747475,   4.94949495,   5.15151515,
        5.35353535,   5.55555556,   5.75757576,   5.95959596,
        6.16161616,   6.36363636,   6.56565657,   6.76767677,
        6.96969697,   7.17171717,   7.37373737,   7.57575758,
        7.77777778,   7.97979798,   8.18181818,   8.38383838,
        8.58585859,   8.78787879,   8.98989899,   9.19191919,
        9.39393939,   9.5959596 ,   9.7979798 ,  10.        ])
```

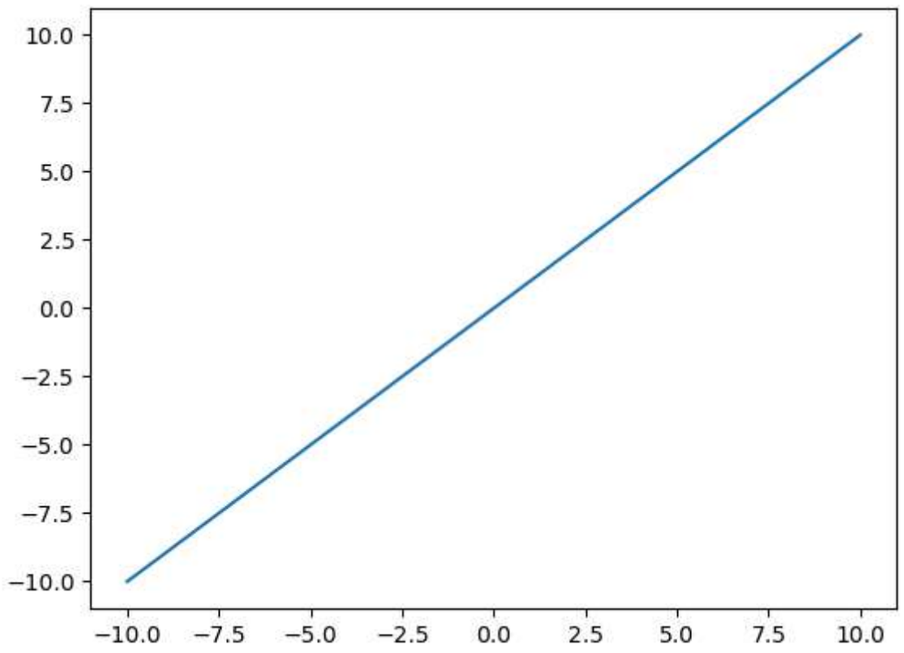
In [824…
```
y = x  # Assigns the values of x to y (y now references the same data as x)
```

In [825…
```
y
```

Out[825…
```
array([-10.        ,  -9.7979798 ,  -9.5959596 ,  -9.39393939,
        -9.19191919,  -8.98989899,  -8.78787879,  -8.58585859,
        -8.38383838,  -8.18181818,  -7.97979798,  -7.77777778,
        -7.57575758,  -7.37373737,  -7.17171717,  -6.96969697,
        -6.76767677,  -6.56565657,  -6.36363636,  -6.16161616,
        -5.95959596,  -5.75757576,  -5.55555556,  -5.35353535,
        -5.15151515,  -4.94949495,  -4.74747475,  -4.54545455,
        -4.34343434,  -4.14141414,  -3.93939394,  -3.73737374,
        -3.53535354,  -3.33333333,  -3.13131313,  -2.92929293,
        -2.72727273,  -2.52525253,  -2.32323232,  -2.12121212,
        -1.91919192,  -1.71717172,  -1.51515152,  -1.31313131,
        -1.11111111,  -0.90909091,  -0.70707071,  -0.50505051,
        -0.3030303 ,  -0.1010101 ,   0.1010101 ,   0.3030303 ,
         0.50505051,   0.70707071,   0.90909091,   1.11111111,
         1.31313131,   1.51515152,   1.71717172,   1.91919192,
         2.12121212,   2.32323232,   2.52525253,   2.72727273,
         2.92929293,   3.13131313,   3.33333333,   3.53535354,
         3.73737374,   3.93939394,   4.14141414,   4.34343434,
         4.54545455,   4.74747475,   4.94949495,   5.15151515,
         5.35353535,   5.55555556,   5.75757576,   5.95959596,
         6.16161616,   6.36363636,   6.56565657,   6.76767677,
         6.96969697,   7.17171717,   7.37373737,   7.57575758,
         7.77777778,   7.97979798,   8.18181818,   8.38383838,
         8.58585859,   8.78787879,   8.98989899,   9.19191919,
         9.39393939,   9.5959596 ,   9.7979798 ,  10.        ])
```

In [826…
```python
import matplotlib.pyplot as plt  # Import the Matplotlib library for plotting
plt.plot(x, y)  # Create a simple line plot of y versus x
```

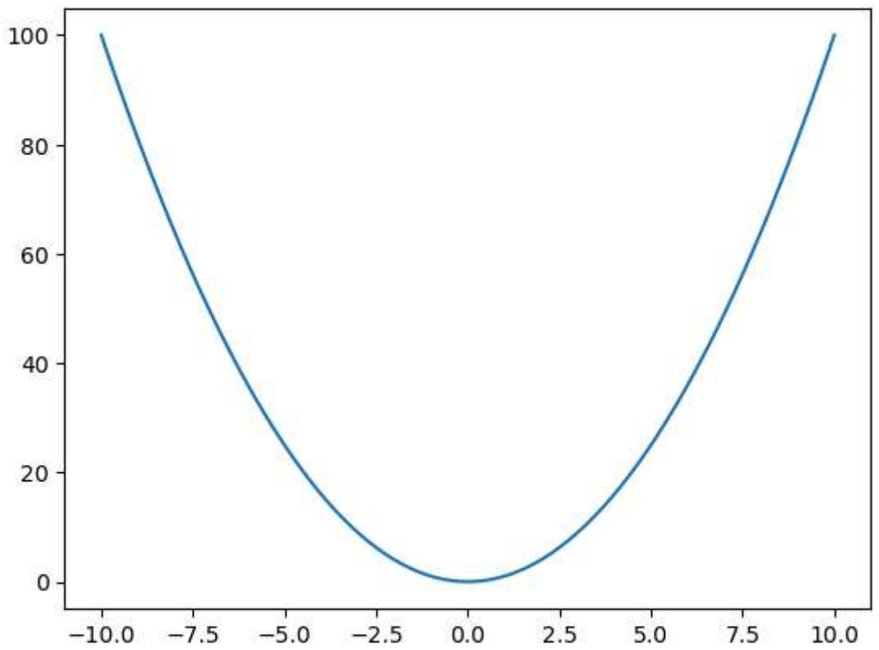
Out[826…
```
[<matplotlib.lines.Line2D at 0x22d01ac0f50>]
```



In [827…
```python
# Generate 100 evenly spaced values from -10 to 10
x = np.linspace(-10, 10, 100)

# Compute y as the square of each x value
```

```
y = x**2

# Plot y versus x as a line graph
plt.plot(x, y)
```

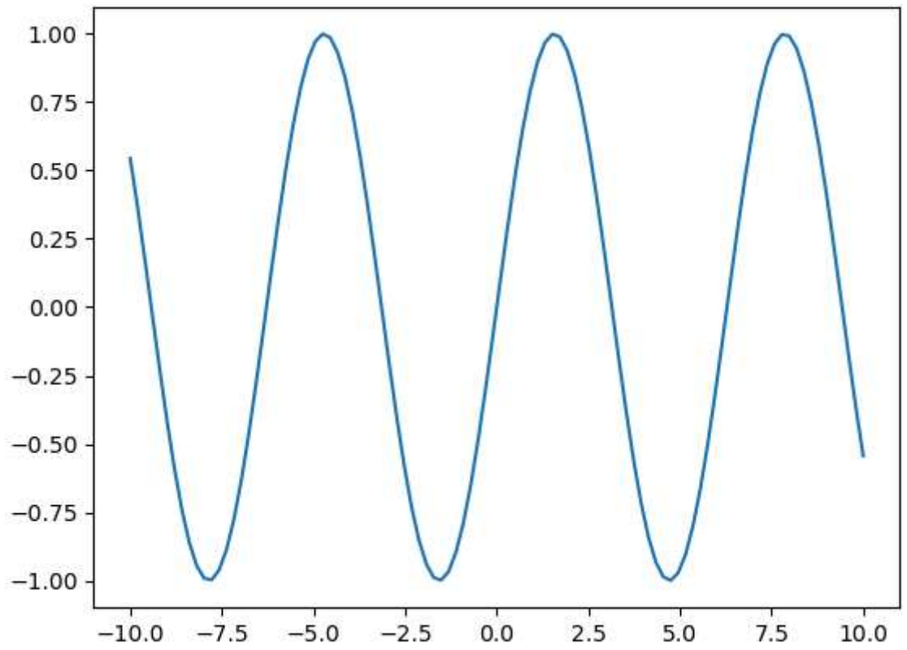
Out[827… [<matplotlib.lines.Line2D at 0x22d023afb10>]



In [828…

```
# Create 100 points between -10 and 10
x = np.linspace(-10, 10, 100)

# Compute the sine of each x value
y = np.sin(x)

# Plot y versus x
plt.plot(x, y)
```

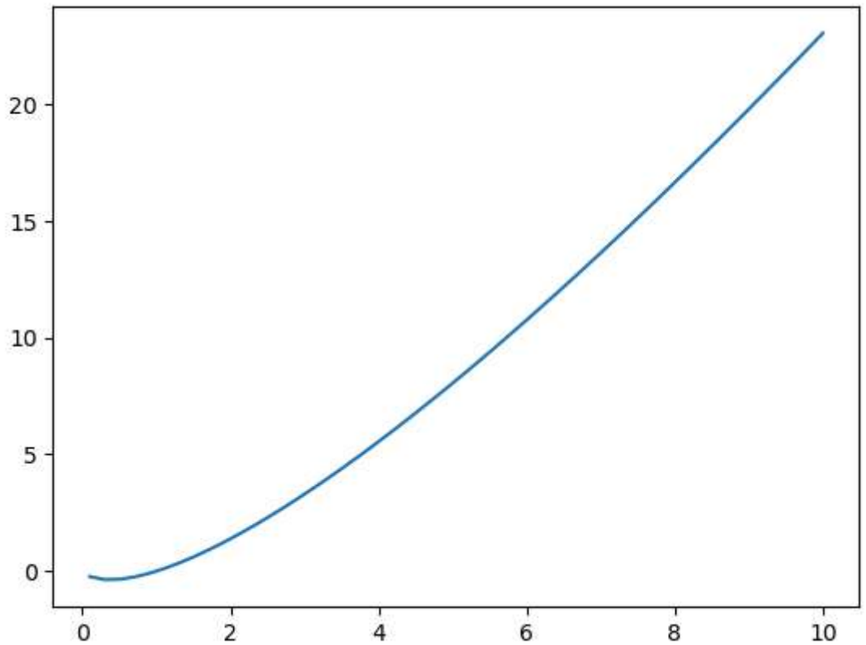
Out[828… [<matplotlib.lines.Line2D at 0x22d024339d0>]



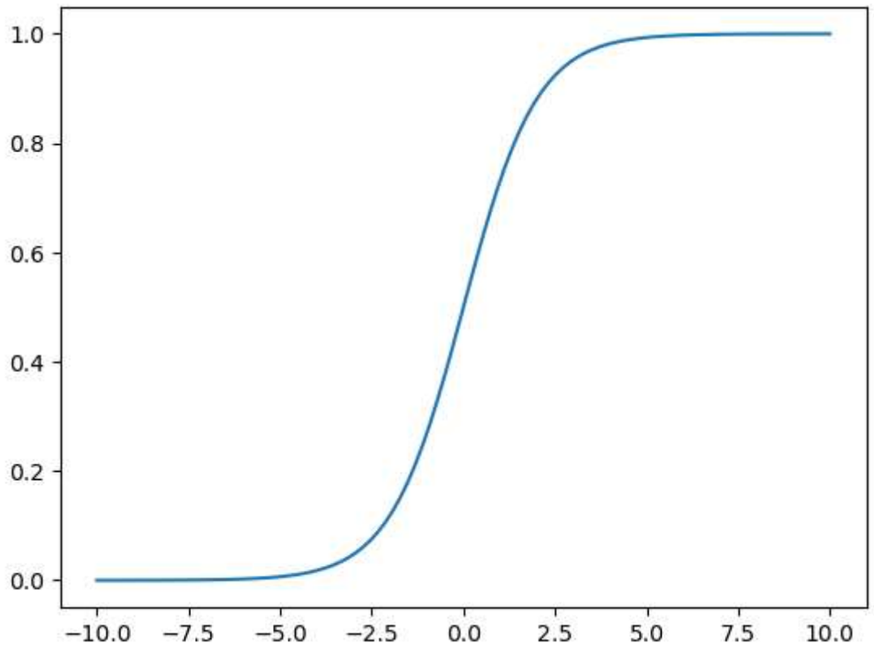
In [829…
```
x = np.linspace(-10, 10, 100)  # Create 100 evenly spaced points between -10 and 10
y = x * np.log(x)               # Compute y = x * log(x) for each point in x
plt.plot(x, y)                  # Plot y versus x using a line plot
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_4296\1584350857.py:2: RuntimeWarning: invalid value encountered in log
  y = x * np.log(x)               # Compute y = x * log(x) for each point in x
```

Out[829…
```
[<matplotlib.lines.Line2D at 0x22d024fc550>]
```



In [830…
```
# Create 100 evenly spaced points between -10 and 10
x = np.linspace(-10, 10, 100)

# Compute the sigmoid function for each x value
y = 1 / (1 + np.exp(-x))

# Plot the sigmoid curve
plt.plot(x, y)
plt.show()
```



In [831…
```
import numpy as np      # Import NumPy for numerical operations and working with arrays
import matplotlib.pyplot as plt  # Import Matplotlib's pyplot module for creating plots and visualizations
```

Meshgrid

A **meshgrid** is a way to create a grid of coordinates from 1D arrays.
It is very useful when you want to **plot 2D functions** or **combine multiple sets of numbers**.

In NumPy, we use `np.meshgrid` to generate these coordinate grids:

- Creating and plotting 2D functions f(x, y)
- Generating all combinations of two or more numbers

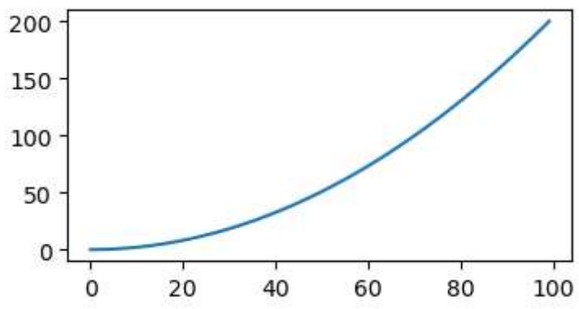
In [832…
```
x = np.linspace(0, 10, 100)  # Create 100 evenly spaced points from 0 to 10 for x-axis
y = np.linspace(0, 10, 100)  # Create 100 evenly spaced points from 0 to 10 for y-axis
```

Try to create 2D function

In [833…
```
f = x**2 + y**2  # Calculate f(x, y) = x² + y² for each point on the grid
```

Plot

In [834…
```
# Create a figure of size 4x2 inches, plot the data in 'f', and display the plot
plt.figure(figsize=(4,2))
plt.plot(f)
plt.show()
```



In [835…
```
x = np.arange(3)  # Create a 1D array [0, 1, 2] representing x-coordinates
y = np.arange(3)  # Create a 1D array [0, 1, 2] representing y-coordinates
```

In [836…
```
x
```

Out[836… array([0, 1, 2])

In [837…
```
y
```

Out[837… array([0, 1, 2])

- Generate a coordinate grid (meshgrid) from 1D arrays x and y
- X and Y will contain the x and y coordinates of all points in the grid

In [838…
```
xv, yv = np.meshgrid(x, y)  # Create 2D coordinate grids from 1D arrays x and y
```

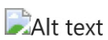
In [839…
```
xv
```

Out[839…
```
array([[0, 1, 2],
       [0, 1, 2],
       [0, 1, 2]])
```

In [840…
```
yv
```

Out[840…
```
array([[0, 0, 0],
       [1, 1, 1],
       [2, 2, 2]])
```

In [841…
```
# Create evenly spaced numbers using np.linspace
# P: 9 numbers from -4 to 4
# V: 11 numbers from -5 to 5
P = np.linspace(-4, 4, 9)
V = np.linspace(-5, 5, 11)

# Print the generated arrays
print(P)
print(V)
```



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

```
In [842… P_1, V_1 = np.meshgrid(P, V)  # Create 2D coordinate grids from 1D arrays P and V
```

```
In [843… print(P_1)
```

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

```
In [844… print(V_1)
```

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

NumPy Meshgrid: Creating Coordinates for a Grid System

Alt text

Using `xv` and `yv` Arrays on a 2D Grid

The arrays `xv` and `yv` represent the **x and y coordinates** on a 2D grid.

- `xv` → all the x-coordinates
- `yv` → all the y-coordinates

You can perform **normal NumPy operations** on these arrays, just like with any other NumPy array.

```
In [845… xv**2 + yv**2  # Compute the sum of squares of xv and yv (e.g., calculating f(x, y) = x^2 + y^2 for a grid)
```

```
Out[845… array([[0, 1, 4],
                [1, 2, 5],
                [4, 5, 8]])
```

Surface Plots of 2D Functions

We can also create **surface plots** to visualize functions of two variables.

For example, consider the function: f(x, y) = e^{-(x^2 + y^2)}

We can plot this function over the ranges:

- (x) from -2 to 2
- (y) from -1 to 1

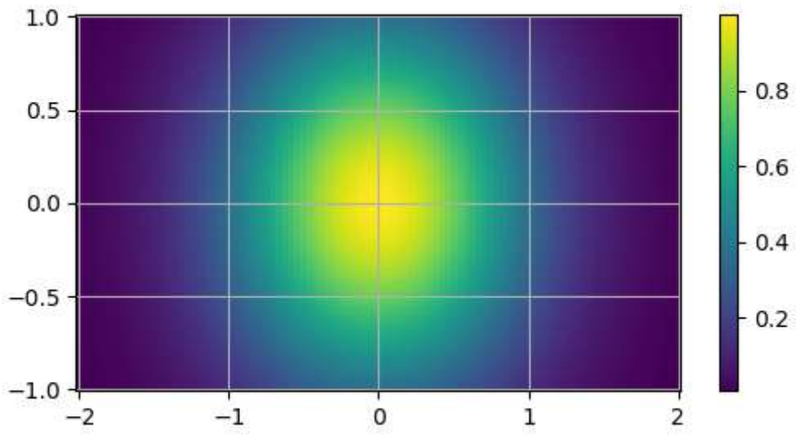
In [846…
```
# Create a 2D Gaussian function using meshgrid
# 1. x and y are 1D arrays from -2 to 2 and -1 to 1 respectively, each with 100 points
# 2. xv, yv are 2D coordinate grids generated by np.meshgrid
# 3. f calculates the function f(x, y) = e^(-(x^2 + y^2)) for each point on the grid
x = np.linspace(-2,2,100)
y = np.linspace(-1,1,100)
xv, yv = np.meshgrid(x, y)
f = np.exp(-xv**2-yv**2)
```

Note on 2D Plots

Note: pcolormesh is typically the preferable function for 2D plotting, as opposed to imshow or pcolor, which take longer. For **2D plotting** in Matplotlib, it is usually better to use:

`plt.pcolormesh()`

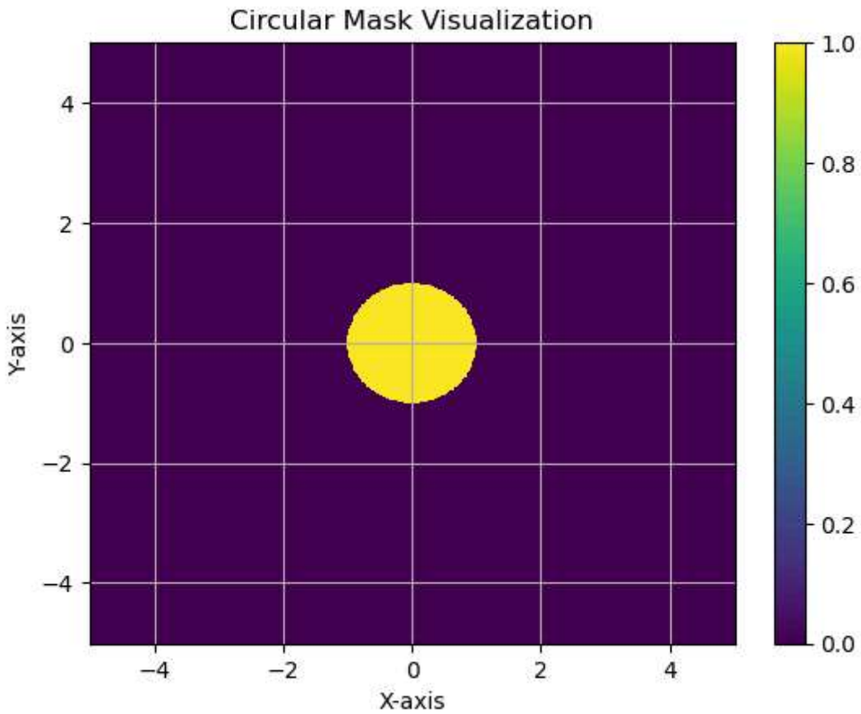
In [847…
```
# Create a figure of size 6x3 inches
# Plot a 2D color mesh using the coordinate grids xv and yv and function values f
# Use 'auto' shading to fill the cells smoothly
# Add a colorbar to indicate the mapping of colors to values
# Add a grid for better readability
# Display the plot
plt.figure(figsize=(6, 3))
plt.pcolormesh(xv, yv, f, shading='auto')
plt.colorbar()
plt.grid()
plt.show()
```



f(x,y) = 1 & x^2+y^2 < 1 \ 0 & x^2+y^2

In [848…
```
import numpy as np
import matplotlib.pyplot as plt

# Define a function that returns 1 inside a circle (radius 1) and 0 outside
def f(x, y):
    return np.where((x**2 + y**2 < 1), 1.0, 0.0)
```

```
# Create x and y values
x = np.linspace(-5, 5, 500)
y = np.linspace(-5, 5, 500)

# Create a 2D grid of coordinates
xv, yv = np.meshgrid(x, y)

# Apply the function to generate the mask
circular_mask = f(xv, yv)

# Plot the masked region
plt.pcolormesh(xv, yv, circular_mask, shading='auto')
plt.colorbar()  # Show color scale
plt.grid()      # Show grid
plt.title("Circular Mask Visualization")
plt.xlabel("X-axis")
plt.ylabel("Y-axis")
plt.show()
```



In [849…
```
# Create a NumPy array 'x' with 9 evenly spaced values
# starting from -4 to 4 (both -4 and 4 are included)
x = np.linspace(-4, 4, 9)
```

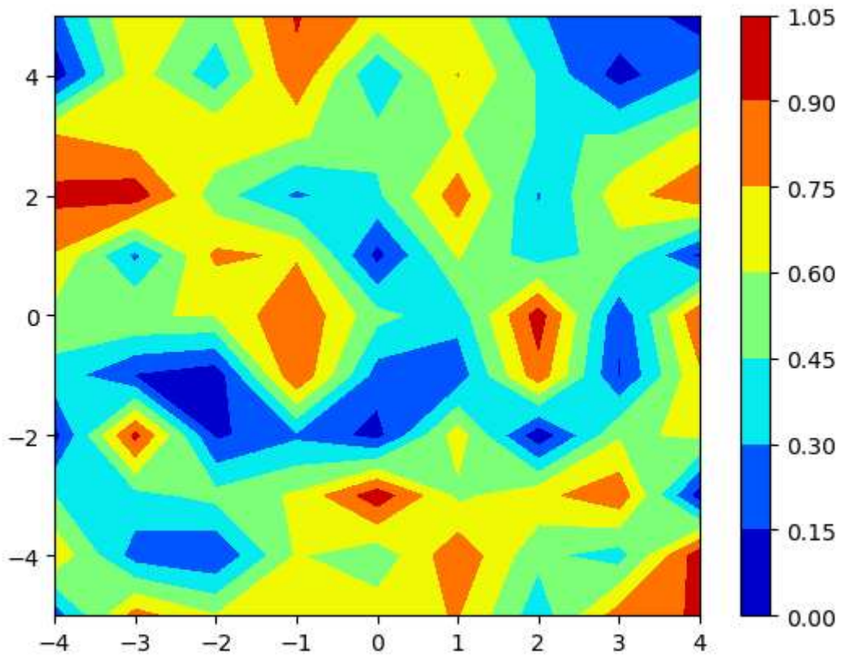
In [850…
```
# numpy.linspace creates an array of
# 9 linearly placed elements between
# -4 and 4, both inclusive
```

In [851…
```
y = np.linspace(-5, 5, 11)  # Creates an array of 11 evenly spaced numbers from -5 to 5
```

In [852…
```
x_1, y_1 = np.meshgrid(x, y)  # Create a 2D coordinate grid from 1D arrays x and y
```

In [853…

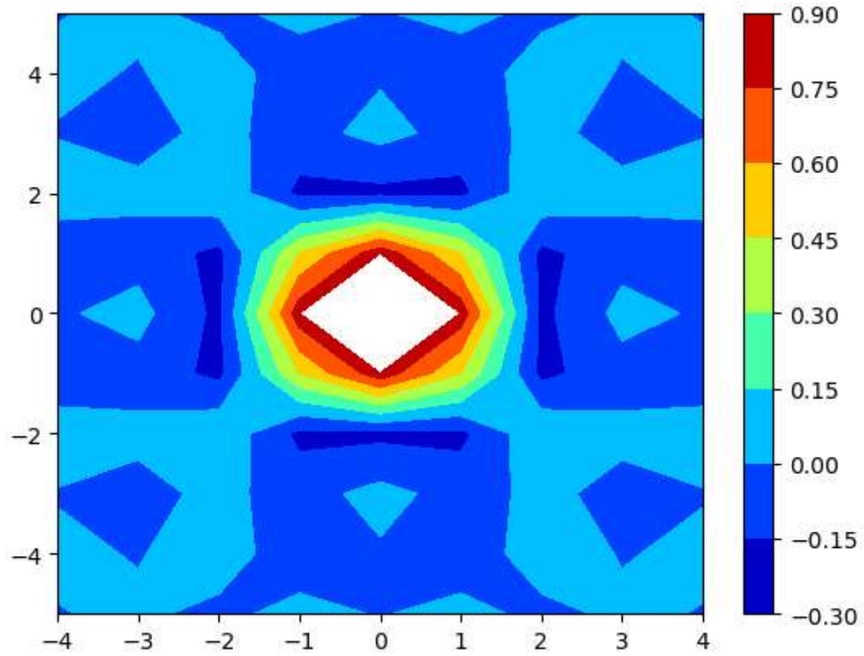
```
# Generate a 2D array of random numbers with shape (11, 9)
random_data = np.random.random((11, 9))

# Create a filled contour plot using the random data
# x_1 and y_1 are coordinate grids, 'jet' colormap is applied
plt.contourf(x_1, y_1, random_data, cmap='jet')

# Add a color bar to show the mapping of values to colors
plt.colorbar()

# Display the plot
plt.show()
```



In [854…

```
# Calculate the 2D sine function values for the meshgrid points
# The function is sin(x^2 + y^2) / (x^2 + y^2)
sine = (np.sin(x_1**2 + y_1**2)) / (x_1**2 + y_1**2)

# Create a filled contour plot of the sine function using the 'jet' colormap
plt.contourf(x_1, y_1, sine, cmap='jet')

# Add a colorbar to show the mapping of values to colors
plt.colorbar()

# Display the plot
plt.show()
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_4296\1343069530.py:3: RuntimeWarning: invalid value encountered in divide
  sine = (np.sin(x_1**2 + y_1**2)) / (x_1**2 + y_1**2)
```



We observe that x_1 is a row repeated matrix whereas y_1 is a column repeated matrix. One row of x_1 and one column of y_1 is enough to determine the positions of all the points as the other values will get repeated over and over.

In [855…
```
# Create a sparse meshgrid from 1D arrays x and y
# 'sparse=True' generates smaller memory-efficient arrays, useful for broadcasting in calculations
x_1, y_1 = np.meshgrid(x, y, sparse=True)
```

In [856…
```
x_1
```

Out[856…
```
array([[-4., -3., -2., -1.,  0.,  1.,  2.,  3.,  4.]])
```

In [857…
```
y_1
```

Out[857…
```
array([[-5.],
       [-4.],
       [-3.],
       [-2.],
       [-1.],
       [ 0.],
       [ 1.],
       [ 2.],
       [ 3.],
       [ 4.],
       [ 5.]])
```

The shape of x_1 changed from (11, 9) to (1, 9) and that of y_1 changed from (11, 9) to (11, 1) The indexing of Matrix is however different. Actually, it is the exact opposite of Cartesian indexing

`np.sort` — Sorting Arrays in NumPy

The `np.sort()` function **returns a sorted copy of a NumPy array** without changing the original array.

In [858…
```
a = np.random.randint(1, 100, 15)  # Generate a 1D NumPy array of 15 random integers between 1 and 99
a
```

Out[858…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [859…
```
b = np.random.randint(1, 100, 24).reshape(6, 4)  # Create a 6x4 array of random integers between 1 and 99
b
```

Out[859…
```
array([[91, 44, 61,  3],
       [64, 31, 71,  2],
       [88, 67, 36, 67],
       [ 4, 47, 11, 10],
       [38, 73, 86, 86],
       [55, 23, 78, 14]], dtype=int32)
```

In [860…
```
np.sort(a) # Default= Ascending
```

Out[860…
```
array([ 6, 13, 18, 26, 41, 48, 55, 61, 61, 65, 66, 72, 81, 96, 96],
      dtype=int32)
```

In [861…
```
np.sort(a)[::-1] # Descending order
```

Out[861…
```
array([96, 96, 81, 72, 66, 65, 61, 61, 55, 48, 41, 26, 18, 13,  6],
      dtype=int32)
```

In [862…
```
np.sort(b) # row rise sorting
```

Out[862…
```
array([[ 3, 44, 61, 91],
       [ 2, 31, 64, 71],
       [36, 67, 67, 88],
       [ 4, 10, 11, 47],
       [38, 73, 86, 86],
       [14, 23, 55, 78]], dtype=int32)
```

In [863…
```
np.sort(b,axis = 0) # column rise sorting
```

Out[863…
```
array([[ 4, 23, 11,  2],
       [38, 31, 36,  3],
       [55, 44, 61, 10],
       [64, 47, 71, 14],
       [88, 67, 78, 67],
       [91, 73, 86, 86]], dtype=int32)
```

np.append()

The `numpy.append()` function is used to **add values to the end of an array**.

In [864…
```
a
```

Out[864…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [865…
```
np.append(a, 200)  # Adds the value 200 to the end of the array 'a' and returns a new array
```

Out[865…
```
array([ 61,  81,  65,  96,  18,  72,  26,  66,   6,  96,  55,  41,  13,
        61,  48, 200])
```

In [866…
```
b # on 2D
```

Out[866…
```
array([[91, 44, 61,  3],
       [64, 31, 71,  2],
       [88, 67, 36, 67],
       [ 4, 47, 11, 10],
       [38, 73, 86, 86],
       [55, 23, 78, 14]], dtype=int32)
```

In [867…
```
# Append a column of ones to the array `b` along the last axis
# np.ones((b.shape[0], 1)) creates a column with the same number of rows as `b`
# np.append(..., axis=1) adds it as a new column
np.append(b, np.ones((b.shape[0], 1)), axis=1)
```

Out[867…
```
array([[91., 44., 61.,  3.,  1.],
       [64., 31., 71.,  2.,  1.],
       [88., 67., 36., 67.,  1.],
       [ 4., 47., 11., 10.,  1.],
       [38., 73., 86., 86.,  1.],
       [55., 23., 78., 14.,  1.]])
```

In [868…
```
# Adding Extra column :1

np.append(b,np.ones((b.shape[0],1)))
```

Out[868…
```
array([91., 44., 61.,  3., 64., 31., 71.,  2., 88., 67., 36., 67.,  4.,
       47., 11., 10., 38., 73., 86., 86., 55., 23., 78., 14.,  1.,  1.,
        1.,  1.,  1.,  1.])
```

In [869…
```
# Append a new column of random numbers to the existing array `b` along axis 1
np.append(b, np.random.random((b.shape[0], 1)), axis=1)
```

Out[869…
```
array([[9.10000000e+01, 4.40000000e+01, 6.10000000e+01, 3.00000000e+00,
        2.64417214e-01],
       [6.40000000e+01, 3.10000000e+01, 7.10000000e+01, 2.00000000e+00,
        7.65144895e-01],
       [8.80000000e+01, 6.70000000e+01, 3.60000000e+01, 6.70000000e+01,
        7.70145312e-02],
       [4.00000000e+00, 4.70000000e+01, 1.10000000e+01, 1.00000000e+01,
        8.20468269e-01],
       [3.80000000e+01, 7.30000000e+01, 8.60000000e+01, 8.60000000e+01,
        3.61956574e-01],
       [5.50000000e+01, 2.30000000e+01, 7.80000000e+01, 1.40000000e+01,
        6.73319682e-01]])
```

np.concatenate()

The `numpy.concatenate()` function is used to **combine multiple arrays** along an existing axis.

In [870…
```
c = np.arange(6).reshape(2,3)  # Create a 2x3 array with values from 0 to 5
d = np.arange(6,12).reshape(2,3)  # Create a 2x3 array with values from 6 to 11
```

In [871…
```
c
```

Out[871…
```
array([[0, 1, 2],
       [3, 4, 5]])
```

In [872…
```
d
```

Out[872…
```
array([[ 6,  7,  8],
       [ 9, 10, 11]])
```

Using `np.concatenate()` Instead of `vstack` and `hstack`

In NumPy, we often use `vstack` (vertical stack) and `hstack` (horizontal stack) to combine arrays.
But we can also use **`np.concatenate()`**, which is more flexible.

In [873…
```
np.concatenate((c,d)) # Row wise
```

Out[873…
```
array([[ 0,  1,  2],
       [ 3,  4,  5],
       [ 6,  7,  8],
       [ 9, 10, 11]])
```

In [874…
```
np.concatenate((c,d),axis =1 ) # column wise
```

Out[874…
```
array([[ 0,  1,  2,  6,  7,  8],
       [ 3,  4,  5,  9, 10, 11]])
```

np.unique()

The `np.unique()` function in **NumPy** helps us find all the **unique values** in an array.

In [875…
```
# Create a NumPy array with repeated integers from 1 to 6
e = np.array([1,1,2,2,3,3,4,4,5,5,6,6])
```

In [876…
```
e
```

Out[876…
```
array([1, 1, 2, 2, 3, 3, 4, 4, 5, 5, 6, 6])
```

In [877…
```
np.unique(e)  # Returns the sorted unique elements of the array 'e'
```

Out[877…
```
array([1, 2, 3, 4, 5, 6])
```

`np.expand_dims`

The `numpy.expand_dims()` function is used to **add a new dimension** to an existing array.
This is useful when you want to **reshape an array** for operations like matrix multiplication or preparing data for machine learning models.

In [878…
```
a
```

Out[878…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [879…
```
a.shape # 1 D
```

Out[879…
```
(15,)
```

In [880…
```
np.expand_dims(a, axis=0)  # Adds a new dimension at axis 0, converting a 1D array into a 2D row vector
```

Out[880…
```
array([[61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48]],
      dtype=int32)
```

In [881…
```
np.expand_dims(a, axis=0).shape  # Adds a new axis at position 0, converting 1D array 'a' into a 2D array
```

Out[881… (1, 15)

In [882…
```
np.expand_dims(a, axis=1)  # Adds a new axis at position 1, converting 1D array 'a' into a 2D column vector
```

Out[882…
```
array([[61],
       [81],
       [65],
       [96],
       [18],
       [72],
       [26],
       [66],
       [ 6],
       [96],
       [55],
       [41],
       [13],
       [61],
       [48]], dtype=int32)
```

Using `expand_dims()` with Vectors

We can work with both **row vectors** and **column vectors** in Python.

The function `np.expand_dims()` is used to **add an extra dimension** to an existing array (or tensor).

In [883…
```
np.expand_dims(a, axis=1).shape  # Adds a new axis (dimension) at position 1, converting 'a' from 1D to 2D
```

Out[883… (15, 1)

`np.where()` — Find Elements by Condition

The **`numpy.where()`** function helps us **find the positions (indices) of elements** in an array that meet a certain condition.

In [884…
```
a
```

Out[884…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [885…
```
np.where(a > 50)  # Returns the indices of array 'a' where the elements are greater than 50
```

Out[885… (array([0, 1, 2, 3, 5, 7, 9, 10, 13]),)

In [886…
```
#np.where(condition, value_if_true, value_if_false)
```

In [887…
```
np.where(a > 50, 0, a)  # Replace all elements in array 'a' greater than 50 with 0; keep others unchanged
```

Out[887…
```
array([ 0,  0,  0,  0, 18,  0, 26,  0,  6,  0,  0, 41, 13,  0, 48],
      dtype=int32)
```

np.argmax()

The **numpy.argmax()** function is used to find the **index of the maximum value** in an array.

- arg stands for **argument**, which means the **position** of the maximum value.

- You can specify the **axis** along which to search for the maximum.

In [888…
```
a
```

Out[888…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [889…
```
np.argmax(a)  # Returns the index of the **largest value** in array `a`
```

Out[889…
```
np.int64(3)
```

In [890…
```
b # on 2D
```

Out[890…
```
array([[91, 44, 61,  3],
       [64, 31, 71,  2],
       [88, 67, 36, 67],
       [ 4, 47, 11, 10],
       [38, 73, 86, 86],
       [55, 23, 78, 14]], dtype=int32)
```

In [891…
```
np.argmax(b, axis=1)  # Returns the index of the largest value in each row of array 'b'
```

Out[891…
```
array([0, 2, 0, 1, 2, 2])
```

In [892…
```
np.argmax(b, axis=0)  # Returns the indices of the maximum values **column-wise** in array b
```

Out[892…
```
array([0, 4, 4, 4])
```

In [893…
```
# np.argmin returns the index of the minimum value in the array 'a'
```

In [894…
```
a
```

Out[894…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [895…
```
np.argmin(a)
```

Out[895…
```
np.int64(8)
```

Statistics in NumPy: `np.cumsum`

The `numpy.cumsum()` function is used to calculate the **cumulative sum** of elements in an array.

- **Cumulative sum** means each element is the sum of **all previous elements plus the current one**.
- You can also specify an **axis** to calculate sums along rows or columns in 2D arrays.

In [896…
```
a
```

Out[896…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [897…
```
np.cumsum(a)  # Compute the cumulative sum of the elements in array 'a'
```

Out[897…
```
array([ 61, 142, 207, 303, 321, 393, 419, 485, 491, 587, 642, 683, 696,
       757, 805])
```

In [898…
```
b
```

Out[898…
```
array([[91, 44, 61,  3],
       [64, 31, 71,  2],
       [88, 67, 36, 67],
       [ 4, 47, 11, 10],
       [38, 73, 86, 86],
       [55, 23, 78, 14]], dtype=int32)
```

In [899…
```
np.cumsum(b, axis=1)  # Compute the cumulative sum of array 'b' along each row (axis=1)
```

Out[899…
```
array([[ 91, 135, 196, 199],
       [ 64,  95, 166, 168],
       [ 88, 155, 191, 258],
       [  4,  51,  62,  72],
       [ 38, 111, 197, 283],
       [ 55,  78, 156, 170]])
```

In [900…
```
np.cumsum(b, axis=0)  # Computes the cumulative sum of array 'b' along rows (axis 0)
```

Out[900…
```
array([[ 91,  44,  61,   3],
       [155,  75, 132,   5],
       [243, 142, 168,  72],
       [247, 189, 179,  82],
       [285, 262, 265, 168],
       [340, 285, 343, 182]])
```

In [901…
```
# np.cumprod(a) --> Returns the cumulative product of elements in array 'a'
# Each element in the output is the product of all previous elements up to that position
```

In [902…
```
np.cumprod(a)  # Computes the cumulative product of elements in array 'a'.
```

Out[902…
```
array([                61,              4941,            321165,
                30831840,         554973120,       39958064640,
           1038909680640,       68568038922240,     411408233533440,
          39495190419210240,  2172235473056563200, -3172065973228666880,
       -4343369504553566208, -6691122745833816064, -7579242546960793600])
```

np.percentile — Nth Percentile in NumPy

The `numpy.percentile()` function is used to **find the nth percentile** of data in an array.

In [903…
```
np.percentile(a, 100)  # Returns the maximum value in the array 'a'
```

Out[903…
```
np.float64(96.0)
```

In [904…
```
np.percentile(a, 0)  # Returns the 0th percentile (minimum value) of array 'a'
```

Out[904…
```
np.float64(6.0)
```

In [905…
```
np.percentile(a, 50)  # Calculate the 50th percentile (median) of the array 'a'
```

Out[905…
```
np.float64(61.0)
```

In [906…
```
np.median(a)  # Computes the median (middle value) of all elements in array 'a'
```

Out[906…
```
np.float64(61.0)
```

np.histogram

NumPy provides a built-in function called `numpy.histogram()` that helps us **analyze the distribution of data**.

It **calculates the frequency of values** in a dataset and can be used to create histograms, which show how often each value or range of values occurs.

In [907…
```
a
```

Out[907…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [908…
```
# Create a histogram of array 'a' with custom bin edges at 10, 20, 30, ..., 100
np.histogram(a, bins=[10,20,30,40,50,60,70,80,90,100])
```

Out[908…
```
(array([2, 1, 0, 2, 1, 4, 1, 1, 2]),
 array([ 10,  20,  30,  40,  50,  60,  70,  80,  90, 100]))
```

In [909…
```
np.histogram(a, bins=[0, 50, 100])  # Computes the histogram of array 'a' using two bins: 0-50 and 50-100
```

Out[909…
```
(array([6, 9]), array([  0,  50, 100]))
```

`np.corrcoef`

The function `np.corrcoef()` is used to find the **correlation** between two or more datasets.

- It returns the **Pearson correlation coefficient**, which tells how strongly two variables are related.
  - Value **1** → Perfect positive correlation
  - Value **0** → No correlation
  - Value **-1** → Perfect negative correlation

In [910…
```
# Creating NumPy arrays:
# 'salary' stores the salaries of 5 employees
# 'experience' stores the corresponding years of work experience
salary = np.array([20000, 40000, 25000, 35000, 60000])
experience = np.array([1, 3, 2, 4, 2])
```

In [911…
```
salary
```

Out[911…
```
array([20000, 40000, 25000, 35000, 60000])
```

In [912…
```
experience
```

Out[912…
```
array([1, 3, 2, 4, 2])
```

In [913…
```
np.corrcoef(salary, experience)  # Calculate the correlation matrix between salary and experience
```

Out[913…
```
array([[1.        , 0.25344572],
       [0.25344572, 1.        ]])
```

Utility Functions: `np.isin()`

The **`np.isin()`** function in NumPy helps us **check if elements of one array exist in another array**.

In [914…
```
a
```

Out[914…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [915…
```
items = [10,20,30,40,50,60,70,80,90,100]

# Check which elements in array 'a' are present in the list 'items'
# np.isin() returns a boolean array of the same shape as 'a'
# True if the element of 'a' is in 'items', False otherwise
np.isin(a, items)
```

Out[915…
```
array([False, False, False, False, False, False, False, False, False,
       False, False, False, False, False, False])
```

In [916…
```
# Selects all elements in array 'a' that are also present in 'items'
a[np.isin(a, items)]
```

Out[916…
```
array([], dtype=int32)
```

`np.flip` — Reverse Array Elements

The **`numpy.flip()`** function **reverses the order** of elements in an array along a specified axis, **without changing the array's shape**.

In [917…
```
a
```

Out[917…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [918…
```
np.flip(a)  # Reverses the order of elements in array 'a' along all axes
```

Out[918…
```
array([48, 61, 13, 41, 55, 96,  6, 66, 26, 72, 18, 96, 65, 81, 61],
      dtype=int32)
```

In [919…
```
b
```

Out[919…
```
array([[91, 44, 61,  3],
       [64, 31, 71,  2],
       [88, 67, 36, 67],
       [ 4, 47, 11, 10],
       [38, 73, 86, 86],
       [55, 23, 78, 14]], dtype=int32)
```

In [920…
```
np.flip(b) # Reverses the order of elements in array 'b along all axes
```

Out[920…
```
array([[14, 78, 23, 55],
       [86, 86, 73, 38],
       [10, 11, 47,  4],
       [67, 36, 67, 88],
       [ 2, 71, 31, 64],
       [ 3, 61, 44, 91]], dtype=int32)
```

In [921…
```
np.flip(b, axis=1)  # Flip the array 'b' horizontally (reverse the order of columns)
```

Out[921…
```
array([[ 3, 61, 44, 91],
       [ 2, 71, 31, 64],
       [67, 36, 67, 88],
       [10, 11, 47,  4],
       [86, 86, 73, 38],
       [14, 78, 23, 55]], dtype=int32)
```

In [922…
```
np.flip(b, axis=0)  # Flip the array 'b' upside down (reverse the order along the rows / first axis)
```

Out[922…
```
array([[55, 23, 78, 14],
       [38, 73, 86, 86],
       [ 4, 47, 11, 10],
       [88, 67, 36, 67],
       [64, 31, 71,  2],
       [91, 44, 61,  3]], dtype=int32)
```

`np.put()` — Replace Elements in a NumPy Array

The `numpy.put()` function allows you to **replace specific elements** in an array with new values.

Key points:

- Works on **flattened arrays** (the array is treated as 1D internally).
- You specify **indices** of elements you want to replace and the **new values**.

In [923…
```
a
```

Out[923…
```
array([61, 81, 65, 96, 18, 72, 26, 66,  6, 96, 55, 41, 13, 61, 48],
      dtype=int32)
```

In [924…
```
np.put(a, [0, 1], [110, 530])  # Replace elements at indices 0 and 1 of array 'a' with 110 and 530 respectively
```

In [925…
```
a
```

Out[925…
```
array([110, 530,  65,  96,  18,  72,  26,  66,   6,  96,  55,  41,  13,
        61,  48], dtype=int32)
```

`np.delete()`

The **`numpy.delete()`** function is used to **remove elements or sub-arrays** from a NumPy array.

- It **returns a new array** with the specified elements removed.
- You can delete elements along a specific **axis** (rows or columns) if needed.

In [926…
```
a
```

Out[926…
```
array([110, 530,  65,  96,  18,  72,  26,  66,   6,  96,  55,  41,  13,
        61,  48], dtype=int32)
```

In [927…
```
np.delete(a, 0)  # Deletes the element at index 0 from array 'a' and returns a new array
```

Out[927…
```
array([530,  65,  96,  18,  72,  26,  66,   6,  96,  55,  41,  13,  61,
        48], dtype=int32)
```

In [928…
```
np.delete(a, [0, 2, 4])  # Deletes the elements at index 0, 2, and 4 from array 'a' and returns a new array
```

Out[928…
```
array([530,  96,  72,  26,  66,   6,  96,  55,  41,  13,  61,  48],
      dtype=int32)
```

NumPy Set Functions

NumPy provides several **set operations** to work with arrays. These functions are useful when you want to **compare or combine arrays**.

1. `np.union1d(arr1, arr2)`
   Returns **all unique elements** from both arrays (like a union in math).

2. `np.intersect1d(arr1, arr2)`
   Returns only the elements **common to both arrays** (like intersection).

3. `np.setdiff1d(arr1, arr2)`
   Returns elements in `arr1` that are **not in** `arr2` (like difference).

4. `np.setxor1d(arr1, arr2)`
   Returns elements that are **in either** `arr1` **or** `arr2` **but not in both** (like symmetric difference).

5. `np.in1d(arr1, arr2)`
   Returns a **boolean array** showing whether each element of `arr1` is in `arr2`.

In [929…
```
m = np.array([1,2,3,4,5])
n = np.array([3,4,5,6,7])
```

In [930…
```
np.union1d(m, n)  # Returns the sorted union of the unique elements from arrays m and n
```

Out[930…
```
array([1, 2, 3, 4, 5, 6, 7])
```

In [931…
```
np.intersect1d(m, n)  # Returns the sorted, unique values that are present in both arrays m and n
```

Out[931…
```
array([3, 4, 5])
```

In [932…
```
# Returns the sorted values in array `m` that are not present in array `n`
np.setdiff1d(m, n)
```

Out[932…
```
array([1, 2])
```

In [933…
```
# Returns the sorted values in array `n` that are NOT in array `m`
np.setdiff1d(n, m)
```

Out[933…
```
array([6, 7])
```

In [934…
```
np.setxor1d(m, n)  # Returns the sorted **symmetric difference** of arrays m and n (elements in m or n but not in both)
```

Out[934…
```
array([1, 2, 6, 7])
```

In [935…
```
# Check if the value 1 is present in the array 'm'
# np.in1d returns a boolean array indicating membership for each element in 'm'
np.in1d(m, 1)
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_4296\3571827762.py:3: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  np.in1d(m, 1)
```

Out[935… array([True, False, False, False, False])

In [936…
```
m[np.in1d(m, 1)]  # Select all elements in array 'm' that are equal to 1
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_4296\2748030463.py:1: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  m[np.in1d(m, 1)]  # Select all elements in array 'm' that are equal to 1
```

Out[936… array([1])

In [937…
```
np.in1d(m, 10)  # Checks which elements of array `m` are equal to 10; returns a boolean array
```

```
C:\Users\DELL\AppData\Local\Temp\ipykernel_4296\2348059225.py:1: DeprecationWarning: `in1d` is deprecated. Use `np.isin` instead.
  np.in1d(m, 10)  # Checks which elements of array `m` are equal to 10; returns a boolean array
```

Out[937… array([False, False, False, False, False])

`np.clip` — Limiting Array Values

The `numpy.clip()` function is used to **limit the values in an array**.

- Any value **smaller than a minimum** is set to the minimum.
- Any value **larger than a maximum** is set to the maximum.
- Values **within the range** stay unchanged.

In [938…
```
a
```

Out[938… array([110, 530, 65, 96, 18, 72, 26, 66, 6, 96, 55, 41, 13,
61, 48], dtype=int32)

In [940…
```
np.clip(a, a_min=15, a_max=50)  # Limits the values in array 'a' so that all values below 15 become 15 and all values above 50 become 50

# it clips the minimum data to 15 and replaces everything below data to 15 and maximum to 50
```

Out[940… array([50, 50, 50, 50, 18, 50, 26, 50, 15, 50, 50, 41, 15, 50, 48],
dtype=int32)

Data Clipping and Axis Swapping in NumPy

1. Clipping Data

The `np.clip()` function **limits values in an array** to a specified minimum and maximu

2. Swapping Axes

The `np.swapaxes()` function switches two axes of an array:

In [941…
```
arr = np.array([[1, 2, 3], [4, 5, 6]])
# Swap the axes of the array: rows become columns and columns become rows
swapped_arr = np.swapaxes(arr, 0, 1)
```

In [942…
```
arr
```

Out[942… array([[1, 2, 3],
[4, 5, 6]])

In [943…
```
print("Original array:")
print(arr)
```

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

In [944…
```
print("Swapped array:")
print(swapped_arr)
```

```
Swapped array:
[[1 4]
 [2 5]
 [3 6]]
```

In [ ]: