

## ✓ Numpy

### Agenda

1. What is Numpy?
2. Why Numpy is used?
3. Importing NumPy in Python
4. Generating an array in numpy
  - 1D,2D and 3D array
  - Create NumPy Array with List and Tuple
  - Create Array of Fixed Size
  - Create Using Function
  - Reshaping Array using Reshape Method
  - Flatten Array
  - Numpy sorting Arrays
5. NumPy array Indexing
  - Basic Indexing
  - Slicing
  - Integer array indexing
  - Boolean array indexing
  - Modifying Elements
  - Indexing in Multi-Dimensional Arrays
  - Combining Indexing Methods
6. NumPy Arithmetic Operations
  - Addition, Subtraction, Multiplication, Division.
  - np.mean(), np.std(), np.min(), np.max().
7. Manipulating Array in Numpy
  - Reshaping Array
  - Flattening Arrays
  - Transpose Operations
  - Concatenating Arrays
  - Stacking Arrays
  - Splitting Arrays

- Adding / Removing Dimensions

## 8. Vector and Matrix Operations in NumPy

## 9. Advanced Topic

- Broadcasting
- Structured Arrays

## ✓ What is Numpy?



- NumPy, short for Numerical Python, is a powerful open-source library in Python used for numerical computing. It provides support for multi-dimensional arrays and matrices, along with a collection of mathematical functions to operate on these arrays efficiently. NumPy is the foundation for many other scientific libraries in Python, including Pandas, SciPy, and Matplotlib.

## ✓ Why Numpy is used?

- NumPy is a cornerstone of scientific computing in Python, and its use is essential in data science, machine learning, finance, engineering, and many other fields that require efficient numerical processing.
- Here are some key reasons why NumPy is widely used:
  - Performance: Fast and memory-efficient numerical computations.
  - Comprehensive Functionality: Extensive mathematical, statistical, and linear algebra operations.
  - Ease of Use: Intuitive API and strong integration with other Python libraries.
  - Advanced Features: Broadcasting, vectorization, and complex array manipulations.

## ✓ Importing NumPy in Python

- import numpy as np: This imports the NumPy library and gives it the alias np. The alias is a common convention in the Python community and is used to make the code more concise.
- After importing NumPy, you can access its functions and methods using the np prefix.

```
import numpy as np
```

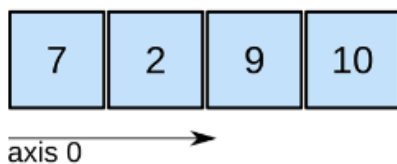
```
# Example
```

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

## ✓ Generating an array in numpy

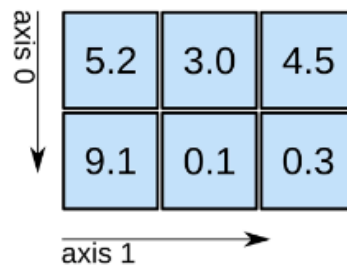
### ✓ 1-D, 2-D and 3-D Array

#### 1D array



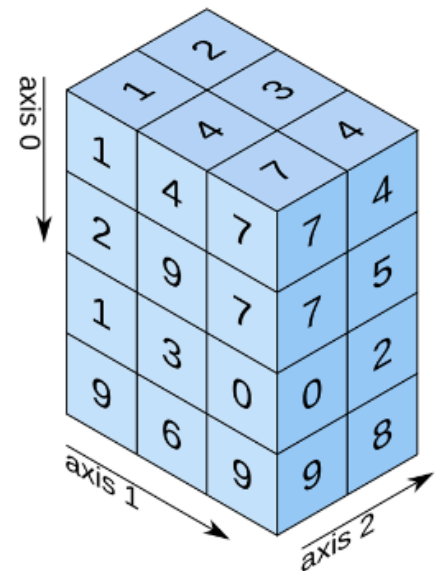
shape: (4,)

#### 2D array



shape: (2, 3)

#### 3D array




shape: (4, 3, 2)

- 1-D Array
  - The One-dimensional array contains elements only in one dimension. In other words, the shape of the NumPy array should contain only one value in the tuple.

```
# Creating a 1D array
```

```
array_1d = np.array([1, 2, 3, 4, 5])
```

```
print(array_1d)
```


 [1 2 3 4 5]

- 2-D Array

- A 2D Array in Python is a two-dimensional data structure kept linearly in memory. It has two dimensions, which are the rows and columns, and hence symbolizes a matrix.

#2D Array

```
import numpy as np
array_2d = np.array([[1, 2, 3],
                     [4, 5, 6]])
print(array_2d)
```



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


- 3-D Array

- Multidimensional arrays are an extension of 2-D matrices and use additional subscripts for indexing. A 3-D array, for example, uses three subscripts. The first two are just like a matrix, but the third dimension represents pages or sheets of elements.

#3D Array

```
import numpy as np

array_3d = np.array([
    [[1, 2, 3], [4, 5, 6], [7, 8, 9]],      # First layer
    [[10, 11, 12], [13, 14, 15], [16, 17, 18]], # Second layer
    [[19, 20, 21], [22, 23, 24], [25, 26, 27]] # Third layer
])
print(array_3d)
```



```
[[[ 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 27]]]
```

## ✓ Create NumPy Array with List and Tuple

- Create a NumPy array from a list:-
  - The simplest way to create a Python list to a NumPy array is by using the `numpy.array()` function. This function takes a Python list as input and returns a NumPy array.

```
#Creating Numpy array from list
import numpy as np

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

 [1 2 3 4 5]

- Create a NumPy Array from a Tuple:-
  - A tuple is similar to a list but is immutable which means it can't be changed and it is in enclosed parentheses `()`.

```
#Creating Numpy Array from Tuple
import numpy as np


tuple_arr = np.array((1, 2, 3, 4, 5))
print(tuple_arr)
```

 [1 2 3 4 5]

## ✓ Create Array of Fixed Size

- Creating an array of a specific size in Python is pretty simple. Let's break it down. First up, you can initialize an array of a specific size with a default value (like 0 or any other value) using the following code: `arr = [0]*size`.

```
size = 5
fixed = [0] * size
print(fixed)
```

 [0, 0, 0, 0, 0]

## ✓ Create Using Function

- You can create arrays using various functions provided by the NumPy library in Python. These functions offer more control and flexibility compared to creating arrays with `np.array`.

- Here are some commonly used functions:

```
#np.zeros():- It creates an array which is filled with zeros.
zeros = np.zeros((3, 3))
print(zeros)
```

```
→ [[0. 0. 0.]
    [0. 0. 0.]
    [0. 0. 0.]]
```

```
#np.ones():- It creates an array which is filled with ones.
ones = np.ones((2,2))
print(ones)
```

```
→ [[1. 1.]
    [1. 1.]]
```

```
#np.full():- Creates an array which is filled with a specific value.
random = np.full((2,2),7)
print(random)
```

```
→ [[7 7]
    [7 7]]
```

```
#np.arange():- The np.arange() function is used to generate an array with evenly spaced values.
sequence_array = np.arange(5,15)
print(sequence_array)
```

```
→ [ 5  6  7  8  9 10 11 12 13 14]
```

```
#np.linspace():- Return evenly spaced numbers over a specified interval.
```

```
# Create an array with 5 values spaced between 0 and 1
line = np.linspace(0, 1, 10)
print(line)
```

```
→ [0.          0.11111111 0.22222222 0.33333333 0.44444444 0.55555556
    0.66666667 0.77777778 0.88888889 1.          ]
```

## ✓ Reshaping Array using Reshape Method

- The shape of an array is the number of elements in each dimension. By reshaping we can add or remove dimensions or change number of elements in each dimension.
- Reshape From 1-D to 2-D

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(4, 3)

print(newarr)
```

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

- Reshape From 1-D to 3-D

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12])

newarr = arr.reshape(2, 3, 2)

print(newarr)
```

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

   [[ 7  8]
    [ 9 10]
    [11 12]]]
```

- Reshaping N-D to 1-D array

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

reshaped = array.reshape((9))

print(reshaped)
```

```
⇒ [1 2 3 4 5 6 7 8 9]
```

- Reshaping using unknown dimension

```
array = np.array([1, 2, 3, 4, 5, 6, 7, 8, 9, 10, 11, 12, 13, 14, 15, 16])

reshaped1 = array.reshape((2, 2, -1))

# converting it to 2-D array
reshaped2 = array.reshape((4, -1))
```

```
print(reshaped2)
```

```
→ [[ 1  2  3  4]
    [ 5  6  7  8]
    [ 9 10 11 12]
    [13 14 15 16]]
```

## ✓ Flatten Array

- The `numpy.ndarray.flatten()` function in Python is a method provided by the NumPy library, which is widely used for numerical and array operations. This function is specifically designed for NumPy arrays (ndarrays) and serves the purpose of returning a flattened copy of the input array.

```
#FLATTEN aARRAY
array_2d = np.array([[1, 2, 3], [4, 5, 6]])
```

```
# Flatten the array
flattened_array = array_2d.flatten()
```

```
print(array_2d)
```

```
print(flattened_array)
```

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

## ✓ Numpy sorting Arrays

- Sorting means putting elements in an ordered sequence.
- Ordered sequence is any sequence that has an order corresponding to elements, like numeric or alphabetical, ascending or descending.
- The NumPy ndarray object has a function called `sort()`, that will sort a specified array.

```
#Sorting the Array
arr = np.array([3, 2, 0, 1])
```

```
print(np.sort(arr))
```

```
→ [0 1 2 3]
```



```
#Sort the array alphabetically
arr = np.array(['banana', 'cherry', 'apple'])

print(np.sort(arr))
```

```
↔ ['apple' 'banana' 'cherry']
```

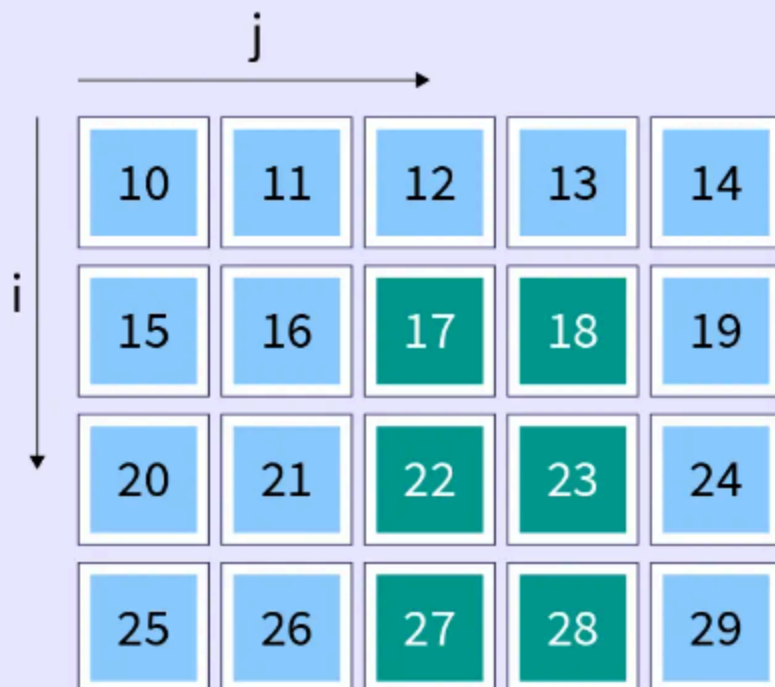
```
#Sort a boolean array
arr = np.array([True, False, True])

print(np.sort(arr))
```

```
↔ [False True True]
```

## ✓ NumPy array Indexing

- NumPy array indexing allows you to access and modify elements in an array. It is powerful and flexible, supporting various types of indexing including basic indexing, slicing, advanced indexing, and boolean indexing.



**Basic Indexing:** Access elements using single or multiple indices.

**Slicing:** Access a range of elements using ``start:stop:step``.

**Advanced Indexing:** Use arrays or lists of indices to access multiple elements at once.

**Boolean Indexing:** Use conditions to filter and access elements.

**Modifying Elements:** Use indexing to modify specific elements or ranges in an array.

**Multi-Dimensional Arrays:** Indexing extends to 2D, 3D, or higher-dimensional arrays.

## ▼ Basic Indexing

- Basic indexing works similarly to Python lists. You use square brackets `[]` to access elements by their index.

```
# Creating a 1D array
array = np.array([10, 20, 30, 40, 50])
```

```
# Accessing individual elements
print(array[0])
print(array[3])
print(array[-1])
```

```
⇒ 10
   40
   50
```

## ▼ Slicing

- Slicing allows you to access a range of elements. The syntax is `array[start:stop:step]`.

```
sub_array = array[1:4]
```

```
# Slicing with a step# Output: [10 30 50] (every second element)
```

```
# Slicing a 2D array
sub_array_2d = array_2d[0:2, 1:3]
```

## Integer array indexing

- By integer array indexing, you can select multiple elements from an array using their specific indices.

## ✓ Boolean Array Indexing

- You can use a boolean array to select elements where the condition is True.

```
# Boolean array indexing
bool_array = array > 30
print(array[bool_array])

# Direct boolean condition
print(array[array % 20 == 0])
```

```
⇒ [40 50]
   [20 40]
```

## ✓ Modifying Elements

- You can modify elements in an array using indexing.

```
# Modify single element
array[0] = 100
print(array)

# Modify multiple elements using slicing
array[1:3] = [200, 300]
print(array)
```

```
⇒ [100 200 300 40 50]
   [100 200 300 40 50]
```

## ✓ Indexing in Multi-Dimensional Arrays

- You can combine basic, advanced, and boolean indexing in multi-dimensional arrays.

```
# Creating a 3D array
array_3d = np.array([[[1, 2, 3], [4, 5, 6]],
                    [[7, 8, 9], [10, 11, 12]]])

# Accessing specific elements
print(array_3d[0, 1, 2])
# Slicing in 3D array
sub_array_3d = array_3d[:, 1, :2]

# Boolean indexing in 3D
```

```
bool_idx_3d = array_3d > 5  
print(array_3d[bool_idx_3d])
```

```
↵ 6  
[ 6  7  8  9 10 11 12]
```

## ✓ Combining Indexing Methods

- You can combine different indexing methods for more complex operations.

```
# Combining slicing and integer indexing  
combined = array_2d[1:, [0, 2]]  
print(combined)
```

```
# Combining boolean indexing with modification  
array[array > 50] = 0  
print(array)
```

```
↵ [[4 6]]  
[ 0  0  0 40 50]
```

## ✓ NumPy Arithmetic Operations

- NumPy is an open-source Python library for performing array computing (matrix operations). It is a wrapper around the library implemented in C and used for performing several trigonometric, algebraic, and statistical operations. NumPy objects can be easily converted to other types of objects like the Pandas data frame and the tensorflow tensor.

### ARITHMETIC OPERATIONS



## ✓ Addition

- NumPy's `numpy.add()` is a function that performs element-wise addition on NumPy arrays. This means it adds the corresponding elements between two arrays, element by element, instead of treating them as single values.
- `numpy.add()` function is used when we want to compute the addition of two arrays.

```
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])
```

```
# Performing addition using arithmetic operator
add_ans = a+b
print(add_ans)
```

```
# Performing addition using numpy function
add_ans = np.add(a, b)
print(add_ans)
```

```
# this would work
c = np.array([1, 2, 3, 4])
add_ans = a+b+c
print(add_ans)
```

```
# but here NumPy only considers the first two arrays (a and b) and ignores the third one (c)
add_ans = np.add(a, b, c)
print(add_ans)
```

```
⇒ [ 7  77  23 130]
   [ 7  77  23 130]
   [ 8  79  26 134]
   [ 7  77  23 130]
```

## ✓ Subtraction

- `numpy.subtract()` function is used when we want to compute the difference of two array. It returns the difference of `arr1` and `arr2`, element-wise.

```
# Defining both the matrices
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])
```

```
# Performing subtraction using arithmetic operator
sub_ans = a-b
print(sub_ans)
```

```
# Performing subtraction using numpy function
```

```
sub_ans = np.subtract(a, b)
print(sub_ans)
```

```
↩ [ 3 67  3 70]
   [ 3 67  3 70]
```

## ✓ Multiplication

- `numpy.multiply()` function is used when we want to compute the multiplication of two array. It returns the product of `arr1` and `arr2`, element-wise.

```
# Defining both the matrices
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])
```

```
# Performing multiplication using arithmetic operator
mul_ans = a*b
print(mul_ans)
```

```
# Performing multiplication using numpy function
mul_ans = np.multiply(a, b)
print(mul_ans)
```

```
↩ [ 10 360 130 3000]
   [ 10 360 130 3000]
```

## ✓ Division

- `numpy.divide(arr1, arr2, out = None, where = True, casting = 'same_kind', order = 'K', dtype = None)` : Array element from first array is divided by elements from second element (all happens element-wise). Both `arr1` and `arr2` must have same shape and element in `arr2` must not be zero; otherwise it will raise an error.

```
# Defining both the matrices
a = np.array([5, 72, 13, 100])
b = np.array([2, 5, 10, 30])
```

```
# Performing division using arithmetic operators
div_ans = a/b
print(div_ans)
```

```
# Performing division using numpy functions
div_ans = np.divide(a, b)
print(div_ans)
```

```

↳ [ 2.5      14.4      1.3      3.33333333]
   [ 2.5      14.4      1.3      3.33333333]

```

### ✓ np.mean()

- Compute the arithmetic mean (average) of the given data (array elements) along the specified axis.

```

# 1D array
arr = [20, 2, 7, 1, 34]

print("arr : ", arr)
print("mean of arr : ", np.mean(arr))

```

```

↳ arr : [20, 2, 7, 1, 34]
   mean of arr : 12.8

```

### ✓ numpy.minimum()

- This function is used to find the element-wise minimum of array elements.
- It compare two arrays and returns a new array containing the element-wise minima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned.

```

in_num1 = 10
in_num2 = 21

print ("Input  number1 : ", in_num1)
print ("Input  number2 : ", in_num2)

out_num = np.minimum(in_num1, in_num2)
print (out_num)

```

```

↳ Input  number1 : 10
   Input  number2 : 21
   10

```

### ✓ numpy.maximum()

- This function is used to find the element-wise maximum of array elements.
- It compares two arrays and returns a new array containing the element-wise maxima. If one of the elements being compared is a NaN, then that element is returned. If both elements are NaNs then the first is returned.

```

in_num1 = 10
in_num2 = 21

print ("Input  number1 : ", in_num1)
print ("Input  number2 : ", in_num2)

out_num = np.maximum(in_num1, in_num2)
print ("maximum of 10 and 21 : ", out_num)

```

```

⇒ Input  number1 :  10
   Input  number2 :  21
   maximum of 10 and 21 :  21

```

### ✓ numpy.std(arr, axis = None)

- Compute the standard deviation of the given data (array elements) along the specified axis(if any)..
- Standard Deviation (SD) is measured as the spread of data distribution in the given data set.

```

# 1D array
arr = [20, 2, 7, 1, 34]

print("arr : ", arr)
print("std of arr : ", np.std(arr))

print ("\nMore precision with float32")
print("std of arr : ", np.std(arr, dtype = np.float32))

print ("\nMore accuracy with float64")
print("std of arr : ", np.std(arr, dtype = np.float64))

```

```

⇒ arr :  [20, 2, 7, 1, 34]
   std of arr :  12.576167937809991

   More precision with float32
   std of arr :  12.576168

   More accuracy with float64
   std of arr :  12.576167937809991

```

### ✓ Manipulating Array in Numpy

- Manipulating arrays in NumPy is essential for tasks like reshaping, stacking, splitting, and performing mathematical operations on arrays.



- **Reshaping and Flattening:** Change the structure of an array.
- **Transposing:** Rotate the axes of an array.
- **Concatenating and Stacking:** Combine arrays.
- **Splitting:** Divide arrays into multiple sub-arrays.
- **Dimension Manipulation:** Add or remove dimensions.
- **Broadcasting:** Perform operations on arrays of different shapes.
- **Mathematical Operations:** Perform element-wise arithmetic operations.
- **Aggregating Functions:** Compute summary statistics.

## ✓ Reshaping Arrays

- Reshaping allows you to change the dimensions of an array without changing its data.

```
# Creating a 1D array
array = np.array([1, 2, 3, 4, 5, 6])

# Reshaping into a 2x3 array
reshaped_array = np.reshape(array, (2, 3))
print("Reshaped Array:\n", reshaped_array)
```

➞ Reshaped Array:

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

## ✓ Flattening Arrays

- Flattening converts a multi-dimensional array into a 1D array.

```
# Flattening the 2D array back to 1D
flattened_array = reshaped_array.flatten()
print("Flattened Array:", flattened_array)
```

➞ Flattened Array: [1 2 3 4 5 6]

## ✓ Transposing Arrays

- Transposing swaps the axes of an array, effectively rotating it.

```
# Transposing a 2D array (rows become columns)
transposed_array = reshaped_array.T
print("Transposed Array:\n", transposed_array)
```

⇒ Transposed Array:

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

## ✓ Concatenating Arrays

You can concatenate (join) arrays along different axes.

```
array_1 = np.array([[1, 2], [3, 4]])
array_2 = np.array([[5, 6], [7, 8]])

# Concatenating along rows (axis 0)
concatenated_array_0 = np.concatenate((array_1, array_2), axis=0)
print("Concatenated along rows:\n", concatenated_array_0)

# Concatenating along columns (axis 1)
concatenated_array_1 = np.concatenate((array_1, array_2), axis=1)
print("Concatenated along columns:\n", concatenated_array_1)
```

⇒ Concatenated along rows:

```
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
```

Concatenated along columns:

```
[[1 2 5 6]
 [3 4 7 8]]
```

## ✓ Stacking Arrays

- Stacking is similar to concatenation, but it adds an extra dimension.
- Vertical Stacking (vstack): Stack arrays vertically (row-wise).
- Horizontal Stacking (hstack): Stack arrays horizontally (column-wise).

```
# Vertical stacking
vstacked_array = np.vstack((array_1, array_2))
print("Vertically Stacked Array:\n", vstacked_array)

# Horizontal stacking
hstacked_array = np.hstack((array_1, array_2))
print("Horizontally Stacked Array:\n", hstacked_array)
```

```
⇒ Vertically Stacked Array:
[[1 2]
 [3 4]
 [5 6]
 [7 8]]
Horizontally Stacked Array:
[[1 2 5 6]
 [3 4 7 8]]
```

## ✓ Splitting Arrays

- Splitting divides an array into multiple sub-arrays.
  - `np.split()`: Split an array into multiple sub-arrays.
  - `np.hsplit()`: Split horizontally along the columns.
  - `np.vsplit()`: Split vertically along the rows.

```
# Splitting a 1D array into 3 parts
split_array = np.split(array, 3)
print("Split Array:", split_array)
```

```
# Splitting a 2D array into 2 parts horizontally
hsplit_array = np.hsplit(concatenated_array_1, 2)
print("Horizontally Split Array:\n", hsplit_array)
```

```
⇒ Split Array: [array([1, 2]), array([3, 4]), array([5, 6])]
Horizontally Split Array:
[array([[1, 2],
        [3, 4]]), array([[5, 6],
        [7, 8]])]
```

## ✓ Adding/Removing Dimensions

- You can add or remove dimensions using `np.expand_dims()` and `np.squeeze()`.

```
# Adding a new axis
expanded_array = np.expand_dims(array, axis=0) # Convert to 2D
print("Expanded Array Shape:", expanded_array.shape)
```

```
# Removing single-dimensional entries
squeezed_array = np.squeeze(expanded_array)
print("Squeezed Array Shape:", squeezed_array.shape)
```

⇒ Expanded Array Shape: (1, 6)  
Squeezed Array Shape: (6,)

## ✓ Vector and Matrix Operations in NumPy

- Vector and matrix operations are fundamental in NumPy, as it is designed for efficient numerical computing. NumPy provides a variety of functions to perform these operations easily and efficiently.

### ✓ Vector Operations

- Creating Vectors
  - Vectors are essentially 1D arrays in NumPy.

```
# Creating a vector
vector = np.array([1, 2, 3, 4])
```

```
# Vector operations
print("Vector:", vector)
```

⇒ Vector: [1 2 3 4]

- Vector Addition and Subtraction
  - You can add or subtract vectors element-wise.

```
# Creating two vectors
vector_a = np.array([1, 2, 3])
vector_b = np.array([4, 5, 6])
```

```
# Addition
vector_add = vector_a + vector_b
print("Vector Addition:", vector_add)
```

```
# Subtraction
vector_sub = vector_a - vector_b
print("Vector Subtraction:", vector_sub)
```

➞ Vector Addition: [5 7 9]  
Vector Subtraction: [-3 -3 -3]

- Scalar Multiplication
  - Multiply each element of the vector by a scalar.

```
# Scalar multiplication
scalar = 2
vector_scaled = scalar * vector_a
print("Scalar Multiplication:", vector_scaled)
```

➞ Scalar Multiplication: [2 4 6]

- Dot Product
  - The dot product of two vectors is the sum of the products of their corresponding elements.

```
# Dot product
dot_product = np.dot(vector_a, vector_b)
print("Dot Product:", dot_product)
```

➞ Dot Product: 32

## ✓ Matix Operations

- Creating Matrices
  - Matrices in NumPy are 2D arrays.

```
# Creating a matrix
matrix = np.array([[1, 2], [3, 4]])

print("Matrix:\n", matrix)
```

➞ Matrix:  
[[1 2]  
[3 4]]

- Matrix Addition and Subtraction
  - Matrices must have the same shape for addition and subtraction.

```
# Creating two matrices
matrix_a = np.array([[1, 2], [3, 4]])
matrix_b = np.array([[5, 6], [7, 8]])

# Addition
matrix_add = matrix_a + matrix_b
print("Matrix Addition:\n", matrix_add)

# Subtraction
matrix_sub = matrix_a - matrix_b
print("Matrix Subtraction:\n", matrix_sub)
```

```
⇒ Matrix Addition:
[[ 6  8]
 [10 12]]
Matrix Subtraction:
[[-4 -4]
 [-4 -4]]
```

- Matrix Multiplication

- Matrix multiplication involves the dot product of rows and columns.

```
# Matrix multiplication
matrix_mul = np.dot(matrix_a, matrix_b)
print("Matrix Multiplication:\n", matrix_mul)
```

```
⇒ Matrix Multiplication:
[[19 22]
 [43 50]]
```

# Alternatively, you can use the @ operator for matrix multiplication in Python 3.5+:

```
matrix_mul_alt = matrix_a @ matrix_b
print("Matrix Multiplication (using @):\n", matrix_mul_alt)
```

```
⇒ Matrix Multiplication (using @):
[[19 22]
 [43 50]]
```

- Matrix Transposition

- Transpose a matrix by swapping its rows and columns.

```
# Transposing a matrix
matrix_transpose = matrix_a.T
print("Matrix Transposition:\n", matrix_transpose)
```



Matrix Transposition:

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

- Matrix Determinant

- The determinant of a matrix is a scalar value that can be computed for square matrices.

```
# Determinant of a matrix
matrix_det = np.linalg.det(matrix_a)
print("Matrix Determinant:", matrix_det)
```



Matrix Determinant: -2.0000000000000004

## ✓ Broadcasting

- The term broadcasting refers to the ability of NumPy to treat arrays with different dimensions during arithmetic operations. This process involves certain rules that allow the smaller array to be 'broadcast' across the larger one, ensuring that they have compatible shapes for these operations.
- Broadcasting is not limited to two arrays; it can be applied over multiple arrays as well.
- Just to have a clear understanding, let's count calories in foods using a macro-nutrient breakdown. Roughly put, the caloric parts of food are made of fats (9 calories per gram), protein (4 CPG), and carbs (4 CPG).
- So if we list some foods (our data), and for each food list its macro-nutrient breakdown (parameters), we can then multiply each nutrient by its caloric value (apply scaling) to compute the caloric breakdown of every food item.
- Rules for Broadcasting:-
  - If the arrays don't have the same rank then prepend the shape of the lower rank array with 1s until both shapes have the same length.
  - The two arrays are compatible in a dimension if they have the same size in the dimension or if one of the arrays has size 1 in that dimension.
  - The arrays can be broadcast together if they are compatible with all dimensions.
  - After broadcasting, each array behaves as if it had a shape equal to the element-wise maximum of shapes of the two input arrays.
  - In any dimension where one array had size 1 and the other array had size greater than 1, the first array behaves as if it were copied along that dimension.

```
# Array Broadcasting of Single-Dimensional Array
a = np.array([17, 11, 19]) # 1x3 Dimension array
print(a)
b = 3
print(b)
```

```
# Broadcasting happened because of
# miss match in array Dimension.
c = a + b
print(c)
```

```
⇒ [17 11 19]
   3
   [20 14 22]
```

```
# Array Broadcasting of Two-Dimensional Array
A = np.array([[11, 22, 33], [10, 20, 30]])
print(A)
```

```
b = 4
print(b)
```

```
C = A + b
print(C)
```

```
⇒ [[11 22 33]
   [10 20 30]]
   4
   [[15 26 37]
   [14 24 34]]
```

- In this example, the code showcases NumPy broadcasting operations:
  - Computing the outer product of vectors
  - Broadcasting a vector to a matrix,
  - Broadcasting a vector to the transposed matrix,
  - Reshaping and broadcasting a vector to a matrix, and
  - Performing scalar multiplication on a matrix.

```
v = np.array([1, 2, 3])
w = np.array([4, 5])
```

```
# Outer product of vectors v and w
print(np.reshape(v, (3, 1)) * w)
```

```
x = np.array([[1, 2, 3], [4, 5, 6]])
```

```
# Broadcasting vector v to matrix x
print(x + v)
```



```
# Broadcasting vector w to the transposed matrix x
print((x.T + w).T)

# Reshaping vector w and broadcasting to matrix x
print(x + np.reshape(w, (2, 1)))

# Broadcasting scalar multiplication to matrix x
print(x * 2)
```

```
⇒ [[ 4  5]
   [ 8 10]
   [12 15]]
   [[2 4 6]
   [5 7 9]]
   [[ 5  6  7]
   [ 9 10 11]]
   [[ 5  6  7]
   [ 9 10 11]]
   [[ 2  4  6]
   [ 8 10 12]]
```

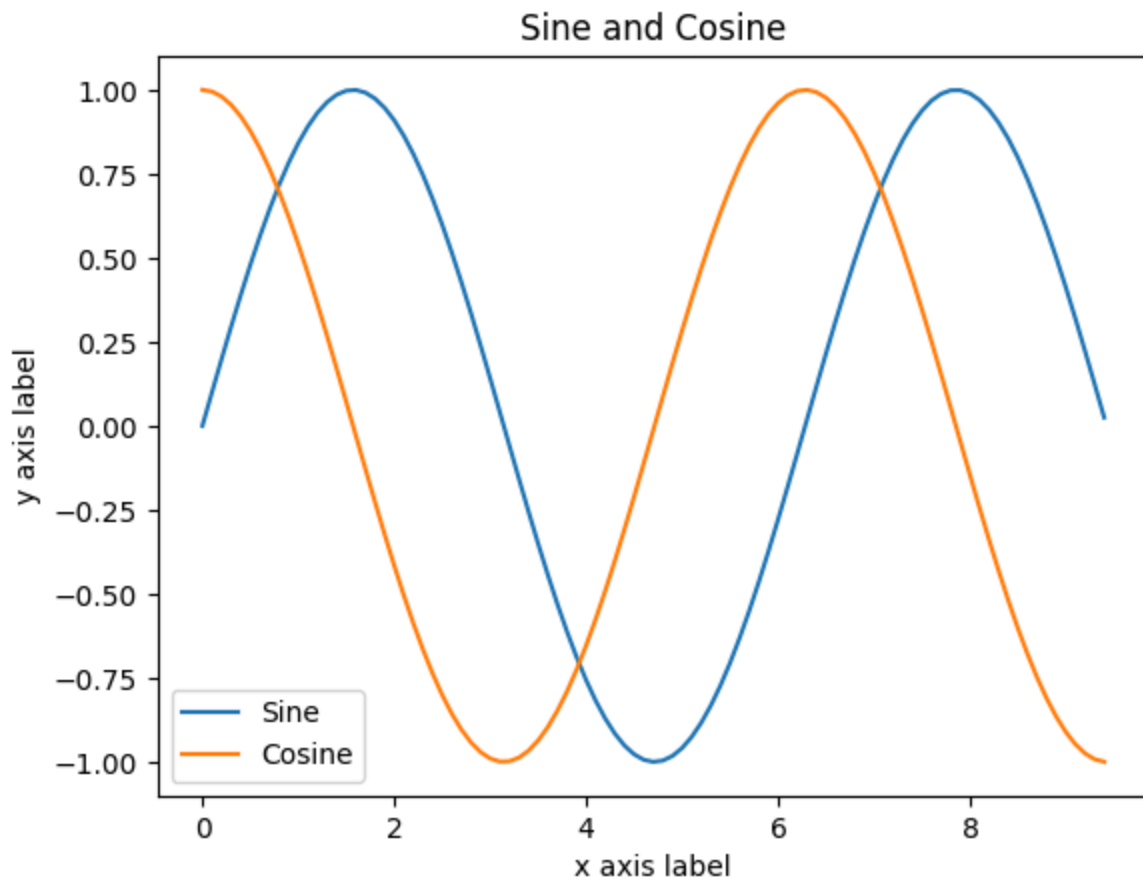
- Broadcasting is also frequently used in displaying images based on two-dimensional functions. If we want to define a function  $z=f(x, y)$ .
- In this example, we are utilizing NumPy and Matplotlib to generate and plot the sine and cosine curves. It first creates arrays for x coordinates ranging from 0 to  $3\pi$  with a step of 0.1. Then, it computes the corresponding y values for sine and cosine functions.

```
import matplotlib.pyplot as plt

# Computes x and y coordinates for
# points on sine and cosine curves
x = np.arange(0, 3 * np.pi, 0.1)
y_sin = np.sin(x)
y_cos = np.cos(x)

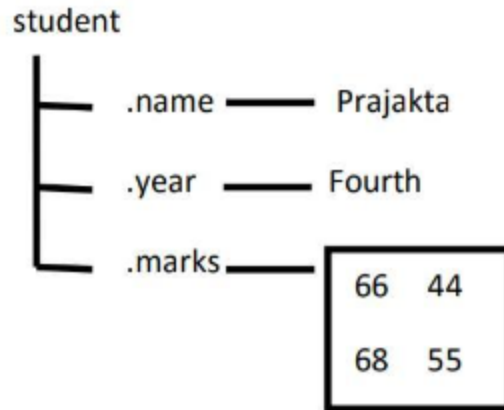
# Plot the points using matplotlib
plt.plot(x, y_sin)
plt.plot(x, y_cos)
plt.xlabel('x axis label')
plt.ylabel('y axis label')
plt.title('Sine and Cosine')
plt.legend(['Sine', 'Cosine'])

plt.show()
```



## ✓ Structured Arrays

- Numpy's Structured Array is similar to the Struct in C. It is used for grouping data of different data types and sizes.
- Structured array uses data containers called fields. Each data field can contain data of any data type and size.
- Properties of Structured Array:-
  - All structs in the array have the same number of fields.
  - All structs have the same field names.
  - For example, consider a structured array of students which has different fields like name, year, and marks. image.png



- Each record in the array `student` has a structure of class `Struct`. The array of a structure is referred to as a struct as adding any new fields for a new struct in the array contains the empty array.
- Why use Structured Array?
  - Structured arrays in NumPy allow us to work with arrays that contain elements of different data types. They are very useful in the case of tabular and structured data.
  - Structured arrays are a very useful and convenient way of storing data. They allow us to store data with different data types, making them very useful for data science projects.

```
# Creating Structured Array in NumPy
dt= np.dtype([('name', (np.str_, 10)), ('age', np.int32), ('weight', np.float64)])
a = np.array([('Sana', 2, 21.0), ('Mansi', 7, 29.0)],
              dtype=dt)
```

```
print(a)
```

```
↗ [ ('Sana', 2, 21.) ('Mansi', 7, 29.) ]
```

```
#Sorting Structured Array
a = np.array([('Sana', 2, 21.0), ('Mansi', 7, 29.0)],
              dtype=[('name', (np.str_, 10)), ('age', np.int32), ('weight', np.float64)])
```

```
# Sorting according to the name
b = np.sort(a, order='name')
print('Sorting according to the name', b)
```

```
# Sorting according to the age
```

```
b = np.sort(a, order='age')  
print('\nSorting according to the age', b)
```

➡ Sorting according to the name [('Mansi', 7, 29.) ('Sana', 2, 21.)]

Sorting according to the age [('Sana', 2, 21.) ('Mansi', 7, 29.)]

- Finding Min and Max in Structured Array:- You can find the minimum and maximum of a structured array using the `np.min()` and `np.max()` functions and pass the fields in the function.