Numpy

(Code: Subhajit Das)

- **Array-processing package:** It provides a high-performance multidimensional array object, and tools for working with these arrays.
- **Scientific computing:** It is the fundamental package for scientific computing with Python.
- **Data container:** Besides its obvious scientific uses, NumPy can also be used as an efficient multi-dimensional container of generic data.
- **Array creation:** Arrays in NumPy can be created by multiple ways, with various number of Ranks, defining the size of the Array.
- **Array operations:** Basic operations such as addition of two arrays can be performed using NumPy

Numpy Array

A NumPy array, also known as an ndarray, is a fundamental data structure provided by the NumPy library in Python

- **Homogeneous:** All elements in a NumPy array are of the same type.
- **Multidimensional:** NumPy arrays can be multidimensional, meaning they can have multiple axes or dimensions.
- **Indexed by integers:** Elements in a NumPy array are indexed by a tuple of positive integers.
- **Fixed size:** Unlike Python lists, NumPy arrays have a fixed size. Changing the size of an array will create a new array, and the original array will be deleted.

Numpy Array vs Python List

- **Data Type:** NumPy arrays are homogeneous (all elements are of the same type), while Python lists can be heterogeneous (can hold different data types).
- **Memory Usage:** NumPy arrays are more memory efficient than Python lists. This is because lists in Python store additional information about each element, such as its type and reference count.
- **Performance:** NumPy arrays are optimized for numerical computations and can perform element-wise operations efficiently. On the other hand, Python lists are not optimized for numerical computations, which can result in slower mathematical operations due to Python's interpretation overhead.
- **Size:** NumPy arrays have a fixed size at creation, unlike Python lists which can grow dynamically.
- **Functionality:** NumPy provides a lot of functionality for operations on arrays, such as mathematical functions, which are not available in Python lists.

Installing Pandas

In [1]:
```
# pip install numpy
```

Importing Pandas

In [2]:
```
import numpy as np
```

Numpy Array vs Python List in Code

In [3]:
```
# Python List
list = [1, 2, 3]
print("This is list: ",(list*2))
print(type(list))

# Numpy Array
array = np.array([1, 2, 3])
print("This is numpy array: ",(array*2)) # data without comma
print(type(array))
```

```
This is list:  [1, 2, 3, 1, 2, 3]
<class 'list'>
This is numpy array:  [2 4 6]
<class 'numpy.ndarray'>
```

Numpy Array vs Python which is faster?

In [4]:
```
%timeit (j**6 for j in range(1,8))
```

```
559 ns ± 156 ns per loop (mean ± std. dev. of 7 runs, 1000000 loops each)
```

In [5]:
```
%timeit np.arange(1,8)**6 # it's much faster, but in this case due to some computat
```

```
2.27 µs ± 91.1 ns per loop (mean ± std. dev. of 7 runs, 100000 loops each)
```

Creating Numpy Array

Syntax: np.array(var/data)

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

print(num)
print(type(num))
```

```
[1 2 3 4]
<class 'numpy.ndarray'>
```

In [7]:
```
# Creating array through user input
li = [] # blank list

for i in range(1,4):
  user = int(input("Enter the number: ")) # Taking user input
  li.append(user) # Add the values one by one in the list

print(np.array(li)) # To print the numpy array
```

```
Enter the number: 4
Enter the number: 6
Enter the number: 9
[4 6 9]
```

Dimensions of Array

1-Dimension Array: 1D array is a simple list of values. It has only one axis (axis-0). The shape attribute for a 1D array returns a tuple with just one element (n,), where n is the length of the array.

In [8]:
```
a = np.array([1, 2, 3, 4])

print(a)
print(a.shape)
print(a.ndim) # ndim will print dimension of array
```

```
[1 2 3 4]
(4,)
1
```

2-Dimension Array: A 2D array has two axes (axis-0 and axis-1). The shape attribute for a 2D array returns a tuple with two elements (n,m), where n is the number of rows and m is the number of columns.

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

print(b)
print(b.shape)
print(b.ndim)
```

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

3-Dimension Array: 3D array has three axes (axis-0, axis-1, and axis-2). The shape attribute for a 3D array returns a tuple with three elements (n,m,p), where n is the number of matrices, m is the number of rows, and p is the number of columns,

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

print(c)
print()

print(c.shape)
print(c.ndim)
```

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

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

(2, 2, 3)
3
```

Higher-Dimension Array

In [11]:
```
d = np.array([[[[1, 2], [3, 4]], [[5, 6], [7, 8]]], [[[9, 10], [11, 12]], [[13, 14]

print(d)
print()

print(d.shape)
print(d.ndim)
```

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

  [[ 5  6]
   [ 7  8]]]


 [[[ 9 10]
   [11 12]]

  [[13 14]
   [15 16]]]]

(2, 2, 2, 2)
4
```

Very High-Dimension Array(8D)

In [12]:
```
e = np.array([1,2,3,4], ndmin=8)

print(e)
print(e.shape)
print(e.ndim)
```

```
[[[[[[[[1 2 3 4]]]]]]]]
(1, 1, 1, 1, 1, 1, 1, 4)
8
```

Creating Numpy Array with Special Functions

- Array filled with 0's **Syntax: np.zeros(array_data)**
- Array filled with 1's **Syntax: np.ones(array_data)**
- Create an empty array **Syntax: np.empty(array_data)**
- An array with a range of elements **Syntax: np.arange(array_data)**
- Array diagonal element filled with 1's **Syntax: np.eye(array_data)**
- Create an array with values that are specified linearly in a specified interval **Syntax: np.linspace(start_num,end_num,num=interval_num)**

Array filled with 0's

In [13]:
```
zero1 = np.zeros(4) # For 1D array
zero2 = np.zeros((2,4)) # For 2D array

print(zero1)
print()
print(zero2)
```

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

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

Array filled with 1's

In [14]:
```
one1 = np.ones(4) # For 1D array
one3 = np.ones((2,3,4)) # For 3D array

print(one1)
print()
print(one3)
```

```
[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.]]]
```

Create an empty array

In [15]:
```
emp1 = np.empty(4) # For 1D array
emp2 = np.empty((2,4)) # For 2D array

print(emp1) # Basically, this can use prev array data
print()
print(emp2)
```

```
[1. 1. 1. 1.]

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

An array with a range of elements

In [16]:
```
rn1 = np.arange(4) # For 1D array
rn2 = np.arange(0, 10).reshape(2, 5) # For 2D array, here reshape will define the a

print(rn1)
print()
print(rn2)
```

```
[0 1 2 3]

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

Array diagonal element filled with 1's

In [17]:
```
di1 = np.eye(4)
di2 = np.eye(4,7)

print(di1)
print()
print(di2)
```

```
[[1. 0. 0. 0.]
 [0. 1. 0. 0.]
 [0. 0. 1. 0.]
 [0. 0. 0. 1.]]

[[1. 0. 0. 0. 0. 0. 0.]
 [0. 1. 0. 0. 0. 0. 0.]
 [0. 0. 1. 0. 0. 0. 0.]
 [0. 0. 0. 1. 0. 0. 0.]]
```

Create an array with values that are specified linearly in a specified interval

In [18]:
```
linsp = np.linspace(0,12,num=4)

print(linsp)
```

```
[ 0.  4.  8. 12.]
```

Creating Numpy Array with Random Numbers

1. rand(): The function is used to generate random value between 0 to 1 **Syntax: np.random.rand(array_data)**
2. randn(): The function is used to generate random value close to zero, this can return positive or negative numbers **Syntax: np.random.randn(array_data)**
3. ranf(): The function for doing random sampling in numpy. It returns an array of specified shape and fills it with random floats in the half open interval [0.0,1.0) **Syntax: np.random.ranf(array_data)**
4. randint(): The function is used to generate a random number between a range **Syntax: np.random.randint(min, max, total_value)**
5. choice(): This function generates a random sample from a given 1-D array. **Syntax: np.random.choice(1D var)**

rand()

In [19]:
```
ran1 = np.random.rand(4) # For 1D array
ran2 = np.random.rand(4,6) # For 2 D array

print(ran1) # Here data will create randomly
print()
print(ran2)
```

```
[0.08485713 0.23141622 0.26169901 0.39238324]

[[0.74857978 0.85091325 0.50375637 0.70505317 0.52553998 0.70513079]
 [0.10141823 0.43619082 0.38993295 0.4751448  0.13928727 0.7280108 ]
 [0.85519374 0.42101185 0.42915338 0.41645176 0.57032456 0.60233302]
 [0.50782837 0.15884688 0.01424419 0.68982297 0.72298657 0.34715381]]
```

randn()

In [20]:
```
rann1 = np.random.randn(4) # For 1D array
rann3 = np.random.randn(2, 4, 5) # For 3D array

print(rann1) # Here data will create randomly
print()
print(rann3)
```

```
[-0.37120482 -0.50057098  1.09910952 -0.12300393]

[[[-1.79740589  0.5535384   0.34015986  0.97178933 -0.13669907]
  [-0.60677684  0.60141317  1.00493717 -0.54503479  0.00976331]
  [-0.60757131 -2.54243951  0.13731453  0.04795244  1.54489466]
  [-0.93828984 -0.02004901 -0.02626399  0.25462949  0.14741524]]

 [[-1.00170846 -0.76365074  0.15519969  0.4464341   1.33409996]
  [ 1.26982489  0.04343138 -0.69571643 -0.15189046  0.51140943]
  [ 0.10727837  0.04455947 -0.48339613  0.59063243 -0.52695   ]
  [ 1.22142942 -0.1190371  -1.0052343   1.35717064  0.04573264]]]
```

ranf()

In [21]:
```
ranf1 = np.random.ranf(4) # For 1D array
ranf2 = np.random.ranf((4,6)) # For 2D array

print(ranf1) # Here data will create randomly
print()
print(ranf2)
```

```
[0.08523901 0.65715886 0.76161086 0.61738397]

[[0.76742695 0.05287837 0.03582305 0.36017993 0.74571846 0.40471436]
 [0.52846912 0.74947432 0.93732828 0.41366169 0.79327015 0.36126579]
 [0.80690903 0.22391444 0.11648136 0.19664942 0.08201029 0.36358197]
 [0.37803217 0.80928594 0.44843018 0.9883228  0.54922248 0.21847964]]
```

randint()

In [22]:
```
randin1 = np.random.randint(1, 20, 4) # For 1D array
randin2 = np.random.randint(1, 20, (4,6)) # For 2D array

print(randin1) # Here data will create randomly
print()
print(randin2)
```

```
[10  1  8  4]

[[18  8  1 19  6  8]
 [19 16 14 19 13 10]
 [ 9  8 15  2  1  4]
 [19 10 14 19 15 12]]
```

Choice:

In [23]:
```
array = np.array([1, 2, 3, 4, 5]) # Only for 1D array

print(np.random.choice(array))
```

```
4
```

Data Types: A data type object (an instance of numpy.dtype class) describes how the bytes in the fixed-size block of memory corresponding to an array item should be interpreted. It describes the following aspects of the data: Type of the data (integer, float, Python object, etc.)

Syntax: var_name.dtype

Functions to use Data Types

1. **bool_:** Boolean (True or False) stored as a byte
2. **int_:** Default integer type (same as C long; normally either int64 or int32)
3. **intc:** Identical to C int (normally int32 or int64)
4. **intp:** Integer used for indexing (same as C ssize_t; normally either int32 or int64)
5. **int8:** Byte (-128 to 127)
6. **int16**: Integer (-32768 to 32767)
7. **int32:** Integer (-2147483648 to 2147483647)
8. **int64:** Integer (-9223372036854775808 to 9223372036854775807)
9. **unit8:** Unsigned integer (0 to 255)
10. **unit16:** Unsigned integer (0 to 65535)
11. **unit32:** Unsigned integer (0 to 4294967295)
12. **unit64:** Unsigned integer (0 to 18446744073709551615)
13. **float_:** Shorthand for float64
14. **float16:** Half precision float: sign bit, 5 bits exponent, 10 bits mantissa
15. **float32:** Single precision float: sign bit, 8 bits exponent, 23 bits mantissa
16. **float64:** Double precision float: sign bit, 11 bits expone, 52 bits mantissa
17. **complex_:** Shorthand for complex128
18. **complex64:** Complex number, represented by two 32-bit floats (real and imaginary components)
19. **complex128:** Complex number, represented by two 64-bit floats (real and imaginary components)

In [24]:
```
var = np.array([1, 2, 3, 4])
print("Data Type: ", var.dtype)
```

```
Data Type:  int64
```

In [25]:
```
var = np.array([1.6, 2, 3.2, 4])
print("Data Type: ", var.dtype)
```

```
Data Type:  float64
```

In [26]:
```
var = np.array(['A', 'B', 'C', 'D'])
print("Data Type: ", var.dtype)
```

```
Data Type:  <U1
```

In [27]:
```
var = np.array(['A', 1, 2, 3, 'D'])
print("Data Type: ", var.dtype)
```

```
Data Type:  <U21
```

List of characters that are used to represent dtype in Numpy

1. **b** : boolean
2. **i** : (signed) integer
3. **u** : unsigned integer
4. **f** : floating-point
5. **c** : complex-floating point
6. **m** : timedelta
7. **M** : datetime
8. **O** : (Python) objects
9. **S** , **a** : zero-terminated bytes (not recommended)
10. **U** : Unicode string
11. **V** : raw data (void) / the fixed chunked of memory for other types

In [28]:
```
# Convert int64 to int8
var = np.array([1, 2, 3, 4], dtype = np.int8)
print("Data Type: ", var.dtype)
```

```
Data Type:  int8
```

In [29]:
```
# Convert int64 to float
var = np.array([1, 2, 3, 4], dtype = "f")
print("Data Type: ", var.dtype)
```

```
Data Type:  float32
```

In [30]:
```
# Convert int64 to float through function
var = np.array([1, 2, 3, 4])
new_var = np.float32(var)

print("Data Type: ", var.dtype)
print("New Data Type: ", new_var.dtype)
print()

print(var)
print(new_var)
```

```
Data Type:  int64
New Data Type:  float32

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

In [31]:
```
# Convert int64 to float directly with astype() function
var = np.array([1, 2, 3, 4])
var_new = var.astype(float)

print("Data Type: ", var.dtype)
print("New Data Type: ", var_new.dtype)
```

```
Data Type:  int64
New Data Type:  float64
```

Shape & Reshape

Shape: In NumPy, the shape of an array is a tuple of integers that describe the number of elements in each dimension.

Syntax: var_name.shape

In [32]:
```
sh = np.array([[1,2,3,4],[1,2,3,4],[1,2,3,4]])
print(sh.shape)
print(sh)
```

```
(3, 4)
[[1 2 3 4]
 [1 2 3 4]
 [1 2 3 4]]
```

Reshape: reshape is a function that changes the shape of an array without changing its data. The new shape must be compatible with the original shape.

Syntax: np.reshape()

In [33]:
```
# Convert 1D to 2D Array
resh = np.array([1,2,3,4,5,6,7,8])
print(resh)
print()

new_resh2 = resh.reshape(4,2) # 4 rows and 2 columns
print(new_resh2)
print(new_resh2.shape)
print(new_resh2.ndim)
```

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

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

In [34]:
```
# Convert 1D to 3D Array and 3D to 2D array and 3D to 1D array
resh = np.array([1,2,3,4,5,3,2,4,9,8,7,4])
print(resh)
print()

new_resh3 = resh.reshape(2, 3, 2) # 2 arrays, each containing 3 rows and 2 columns
print(new_resh3)
print(new_resh3.shape)
print(new_resh3.ndim)
print()

two_resh = new_resh3.reshape(4, 3)
print(two_resh)
print(two_resh.shape)
print(two_resh.ndim)
print()

one_resh = new_resh3.reshape(-1)
print(one_resh)
print(one_resh.shape)
print(one_resh.ndim)
```

```
[1 2 3 4 5 3 2 4 9 8 7 4]

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

 [[2 4]
  [9 8]
  [7 4]]]
(2, 3, 2)
3

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

[1 2 3 4 5 3 2 4 9 8 7 4]
(12,)
1
```

Arithmetic Operations: These operations can be performed on arrays of any size, and the results will be an array of the same size. For example, if we have two arrays, a and b, of the same size

Types of arithmetic operations in Numpy:

1. **Addition:** a+b or np.add(a,b)
2. **Subtraction:** a-b or np.subtract(a,b)
3. **Multiplication:** a*b or np.multiply(a,b)
4. **Division:** a/b or np.divide(a,b)
5. **Exponentiation:** a**b or np.power(a,b)
6. **Modulus:** a%b or np.mod(a,b)
7. **Reciprocal:** 1/a or np.reciprocal(a)

In 1D array

In [35]:
```
# Arithmetic Operation in 1D array
var_add = np.array([1,2,3,4])

new_var_add = var_add + 6

print(new_var_add)
```

```
[ 7  8  9 10]
```

In [36]:
```
# Arithmetic Operation in 1D array
var_pow = np.array([1, 2, 3, 4])

new_var_pow = np.power(var_pow, 6)

print(new_var_pow)
```

```
[   1   64  729 4096]
```

In [37]:
```
# Using reciprocal function
x = np.array([1, 2, 3])

y = np.reciprocal(x)

print(y)
```

```
[1 0 0]
```

In [38]:
```
# Arithmetic Operation in 1D array with 2 arrays
var_sub1 = np.array([1,2,3,4])
var_sub2 = np.array([5,6,7,8])

new_var_sub = var_sub1 - var_sub2

print(new_var_sub)
```

```
[-4 -4 -4 -4]
```

In [39]:
```
# Arithmetic Operation in 1D array with 2 arrays with function
var_mul1 = np.array([1,2,3,4])
var_mul2 = np.array([5,6,7,8])

new_var_mul = np.multiply(var_mul1, var_mul2)

print(new_var_mul)
```

```
[ 5 12 21 32]
```

In 2D array

In [40]:
```
# Arithmetic Operation in 2D array with 2 arrays
var_add1_2D = np.array([[1,2,3,4], [5,6,7,8]])
var_add2_2D = np.array([[5,6,7,8], [1,2,3,4]])

new_var_add_2D = var_add1_2D + var_add2_2D

print(var_add1_2D)
print()

print(var_add2_2D)
print()

print(new_var_add_2D)
```

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

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

[[ 6  8 10 12]
 [ 6  8 10 12]]
```

Arithmetic Functions of Numpy Array:

1. **np.min(var_name):** Returns the smallest value in the array
2. **np.max(var_name):** Returns the largest value in the array
3. **np.argmin(var_name):** Returns the index of the smallest value in the array
4. **np.argmax(var_name):** Returns the index of the largest value in the array
5. **np.sqrt(var_name):** Returns the square root of each element in the array
6. **np.sin(var_name):** Returns the sine of each element in the array
7. **np.cos(var_name):** Returns the cosine of each element in the array
8. **np.tan(var_name):** Returns the tan of each element in the array
9. **np.cumsum(var_name):** returns the cumulative sum of the elements in the array. The cumulative sum is calculated by adding each element in the array sequentially

1-D array

In [41]:
```
# Getting min, max and square value and their position in 1D array
var_1D = np.array([3,7,8,5,6,9,4,9])

print("Min value in the array: ", np.min(var_1D), "and the position is: ", np.argmi
print("Min value in the array: ", np.max(var_1D), "and the position is: ", np.argma
print("Square value: ", np.sqrt(var_1D))
```

```
Min value in the array:  3 and the position is:  0
Min value in the array:  9 and the position is:  5
Square value:  [1.73205081 2.64575131 2.82842712 2.23606798 2.44948974 3.
 2.         3.        ]
```

In [42]:
```
# Getting sin, cos, tan and cumsum value in 1D array
var_maths = np.array([3,7,8,5,6,9])

print("Sin value: ",np.sin(var_maths))
print("Cos value: ",np.cos(var_maths))
print("Tan value: ",np.tan(var_maths))
print("Cumsum value: ",np.cumsum(var_maths))
```

```
Sin value:  [ 0.14112001  0.6569866   0.98935825 -0.95892427 -0.2794155   0.41211
849]
Cos value:  [-0.9899925   0.75390225 -0.14550003  0.28366219  0.96017029 -0.91113
026]
Tan value:  [-0.14254654  0.87144798 -6.79971146 -3.38051501 -0.29100619 -0.45231
566]
Cumsum value:  [ 3 10 18 23 29 38]
```

2-D array

In [43]:
```
# Getting min and max value and their position in 2D array
var_2D = np.array([[3, 7, 8, 5], [4, 2, 6, 1]])

print("Min value in the array: ", np.min(var_2D, axis = 0), "and the position is: "
print("Min value in the array: ", np.max(var_2D, axis = 1), "and the position is: "
```

```
Min value in the array:  [3 2 6 1] and the position is:  [0 1 1 1]
Min value in the array:  [8 6] and the position is:  [2 2]
```

In [44]:
```
# Getting sin, cos, tan and cumsum value in 1D array
var_maths_2D = np.array([[3, 7, 8], [4, 2, 6]])

print("Sin value: ",np.sin(var_maths_2D))
print()
print("Cos value: ",np.cos(var_maths_2D))
print()
print("Tan value: ",np.tan(var_maths_2D))
print()
print("Cumsum value: ",np.cumsum(var_maths_2D))
```

```
Sin value:  [[ 0.14112001  0.6569866   0.98935825]
 [-0.7568025   0.90929743 -0.2794155 ]]

Cos value:  [[-0.9899925   0.75390225 -0.14550003]
 [-0.65364362 -0.41614684  0.96017029]]

Tan value:  [[-0.14254654  0.87144798 -6.79971146]
 [ 1.15782128 -2.18503986 -0.29100619]]

Cumsum value:  [ 3 10 18 22 24 30]
```

Brodcasting: Broadcasting is a NumPy feature that allows you to perform arithmetic operations on arrays with different shapes. When you broadcast two arrays, the smaller array is "broadcast" across the larger array so that they have compatible shapes. This allows you to perform operations on arrays of different sizes without having to manually resize them.

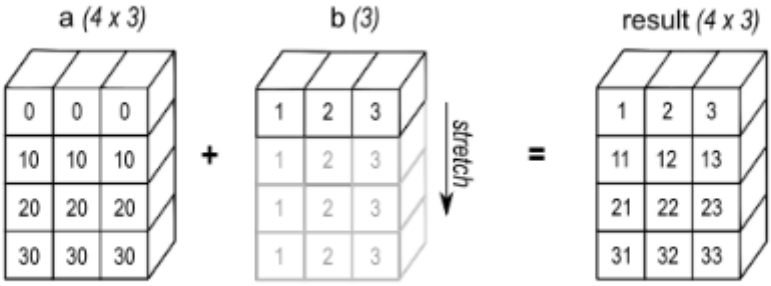
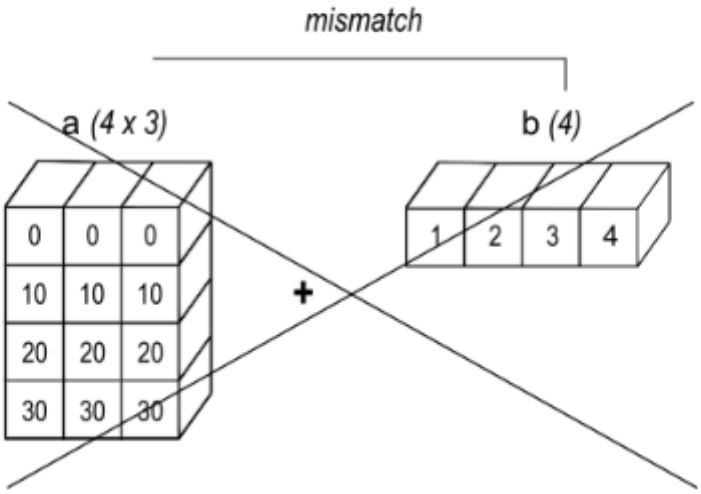


Figure 2

A one dimensional array added to a two dimensional array results in broadcasting if number of 1-d array elements matches the number of 2-d array columns.



In [45]:
```
# Brodcast with two different array
brod1 = np.array([1,2,3])
print(brod1)
print(brod1.shape)
print(brod1.ndim)
print()

brod2 = np.array([[1],[2],[3],[4]])
print(brod2)
print(brod2.shape)
print(brod2.ndim)
print()

print(brod1 + brod2)
```

```
[1 2 3]
(3,)
1

[[1]
 [2]
 [3]
 [4]]
(4, 1)
2

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

In [46]:
```
# Brodcast with same array
brod_2D1 = np.array([[1],[2],[3],[4]])
print(brod_2D1)
print(brod_2D1.shape)
print(brod_2D1.ndim)
print()

brod_2D2 = np.array([[1,2,3,4],[2,1,3,4],[3,4,2,1],[4,3,2,1]])
print(brod_2D2)
print(brod_2D2.shape)
print(brod_2D2.ndim)
print()

print(brod_2D1 + brod_2D2)
```

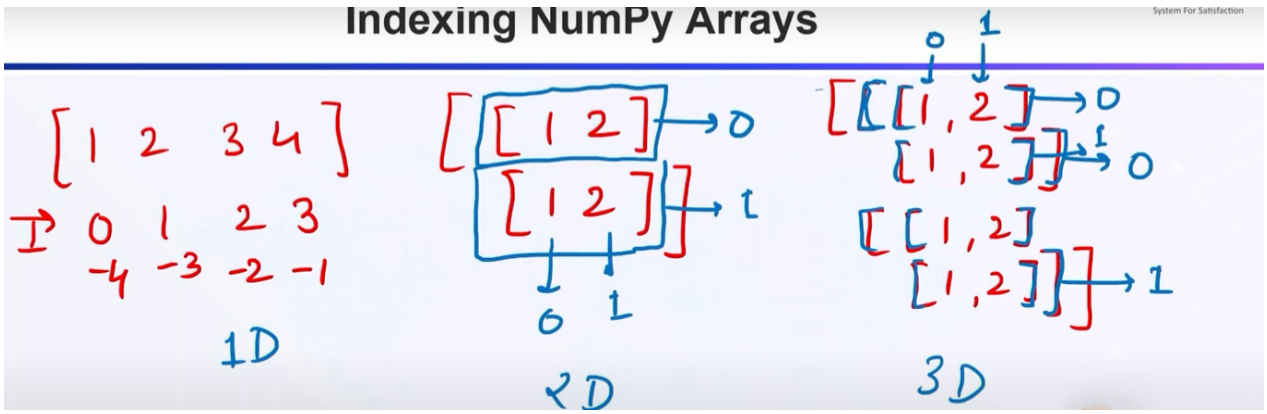
```
[[1]
 [2]
 [3]
 [4]]
(4, 1)
2

[[1 2 3 4]
 [2 1 3 4]
 [3 4 2 1]
 [4 3 2 1]]
(4, 4)
2

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

Indexing & Slicing

Indexing: Integer array indexing allows selection of arbitrary items in the array based on their N-dimensional index.



In [47]:
```
# Indexing in 1-D array
var = np.array([4, 8, 9, 6])
#               0, 1, 2, 3
#               -4,-3,-2,-1

print(var[2])
print(var[-4])
```

```
9
4
```

In [48]:
```
# Indexing in 2-D array
var_index2D = np.array([[4, 8, 7, 1],[9, 6, 5, 2]])

print(var_index2D[0, 1])
print(var_index2D[1, 0])
```

```
8
9
```

In [49]:
```
# Indexing in 3-D array
var_index2D = np.array([[[4, 8, 7, 1],[9, 6, 5, 2]]])

print(var_index2D[0, 1, 1])
print(var_index2D[0, 0, -2])
print(var_index2D[0, -2, -2])
```

```
6
7
7
```

Slicing: Slicing in python means taking elements from one given index to another given index. We pass slice instead of index like this: [start:end]

In [50]:
```
# Slicing in 1-D array
var = np.array([4, 8, 9, 6])
#               0, 1, 2, 3
#               -4,-3,-2,-1

print(var[1:3])
print("8 to end: ",var[1:])
print("Negative Slicing",var[-4:-1])
print(var[-1:-4]) # This will return empty list

print("Skiping value",var[0:3:2]) # It will start eliminating value from end
```

```
[8 9]
8 to end:  [8 9 6]
Negative Slicing [4 8 9]
[]
Skiping value [4 9]
```

In [51]:
```
# Slicing in 2-D array
var_slice2D = np.array([[4, 8, 7, 1],[9, 6, 5, 2]])

print(var_slice2D[0, 1:3])
print(var_slice2D[1, 0:2])
print(var_slice2D[-1, -4:-2])
```

```
[8 7]
[9 6]
[9 6]
```

In [52]:
```
# Slicing in 3-D array
var_slice3D = np.array([[[4, 8, 7, 1], [9, 6, 5, 2]],
                        [[10, 20, 30, 40], [50, 60, 70, 80]]])

print(var_slice3D[0, 1, 1:3])
print(var_slice3D[1, 1, -3:-1])
print(var_slice3D[0, 0, 0:3])
```

```
[6 5]
[60 70]
[4 8 7]
```

Iterating Numpy Arrays: Iterating means going through elements one by one. As we deal with multi-dimensional arrays in numpy, we can do this using basic for loop of python.

Syntax: for i in var_name: print(i)

In [53]:
```
# Iterate in 1D Array
it_1D = np.array([8,6,9,4,5])
print(it_1D)
print()

for i in it_1D:
  print(i)
```

```
[8 6 9 4 5]

8
6
9
4
5
```

In [54]:
```
# Iterate in 2D Array
it_2D = np.array([[1, 2, 3, 4],[5, 6, 7, 8]])
print(it_2D)
print()

for i in it_2D:
  print(i)
```

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

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

In [55]:
```
# Iterate in 2D Array with nested loop
it_2DNest = np.array([[1, 2, 3, 4],[5, 6, 7, 8]])
print(it_2DNest)
print()

for i in it_2DNest:
  for j in i:
    print(j)
```

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

1
2
3
4
5
6
7
8
```

In [56]:
```
# Iterate in 3D Array with nested loop
it_3DNest = np.array([[[1, 2, 3], [6, 7, 8]],
                      [[10, 20, 30], [60, 70, 80]]])
print(it_3DNest)
print()

for i in it_3DNest:
  for j in i:
    for k in j:
      print(k)
```

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

 [[10 20 30]
  [60 70 80]]]

1
2
3
6
7
8
10
20
30
60
70
80
```

In [57]:
```
# Iterate in 3D Array with function
it_3D = np.array([[[1, 2, 3], [6, 7, 8]],
                      [[10, 20, 30], [60, 70, 80]]])
print(it_3D)
print()

for i in np.nditer(it_3D):
  print(i)
```

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

 [[10 20 30]
  [60 70 80]]]

1
2
3
6
7
8
10
20
30
60
70
80
```

In [58]:
```
# Store and changing the iterate data type
# Iterate in 3D Array with function
it_3D = np.array([[[1, 2, 3], [6, 7, 8]],
                      [[10, 20, 30], [60, 70, 80]]])

# Flags will store the data and op_dtypes will change the data type
for i in np.nditer(it_3D, flags = ['buffered'], op_dtypes=["S"]):
  print(i)
```

```
b'1'
b'2'
b'3'
b'6'
b'7'
b'8'
b'10'
b'20'
b'30'
b'60'
b'70'
b'80'
```

In [59]:
```
# Iterate in 3D Array with Enumerate
# If you need to access the elements of the array by their index, you can use the e
it_3D = np.array([[[1, 2, 3], [6, 7, 8]],
                      [[10, 20, 30], [60, 70, 80]]])

for i,d in np.ndenumerate(it_3D):
  print(i,d) # It is printing the index value of each element
```

```
(0, 0, 0) 1
(0, 0, 1) 2
(0, 0, 2) 3
(0, 1, 0) 6
(0, 1, 1) 7
(0, 1, 2) 8
(1, 0, 0) 10
(1, 0, 1) 20
(1, 0, 2) 30
(1, 1, 0) 60
(1, 1, 1) 70
(1, 1, 2) 80
```

Copy Vs View

Copy:A copy is a new array that owns its data. Any changes made to the copy will not affect the original array, and any changes made to the original array will not affect the copy. This is because the copy is physically stored at another location.

Syntax = var_name.copy()

In [60]:
```
var_copy = np.array([4, 6, 9, 8])

co = var_copy.copy()

print(var_copy)
print(co)
```

```
[4 6 9 8]
[4 6 9 8]
```

In [61]:
```
# Data is not changing
var_copy = np.array([4, 6, 9, 8])

co = var_copy.copy()
var_copy[0] = 5

print(var_copy)
print(co)
```

```
[5 6 9 8]
[4 6 9 8]
```

View:A view is just a different way of seeing the data of the original array. It does not own the data. Any changes made to the view will affect the original array, and any changes made to the original array will affect the view. This is because the view shares the same memory location as the original array.

Syntax = var_name.view()

In [62]:
```
var_view = np.array([4, 6, 9, 8])

vi = var_view.view()

print(var_view)
print(vi)
```

```
[4 6 9 8]
[4 6 9 8]
```

In [63]:
```
# Data is changing
var_view = np.array([4, 6, 9, 8])

vi = var_view.view()
var_view[0] = 5

print(var_view)
print(vi)
```

```
[5 6 9 8]
[5 6 9 8]
```

Join & Split

Join: Joining means putting contents of two or more arrays in a single array. In SQL we join tables based on a key, whereas in NumPy we join arrays by axes.

Syntax: np.concatenate((var1, var2))

Other Functions:

1. Height wise: **Syntax: dstack((var1, var2))**
2. Vertically: **Syntax: vstack((var1, var2))**
3. Horizontally: **Syntax: hstack((var1, var2))**
4. Stack: **Syntax: stack((var1, var2))**

In [64]:
```
# Join in 1-D array
var_jo1D_1 = np.array([2, 6, 8, 9])
var_jo1D_2 = np.array([7, 4, 1, 3])

jo1D = np.concatenate((var_jo1D_1, var_jo1D_2)) # Joining 2 different array
jo1D_same = np.concatenate((var_jo1D_1, var_jo1D_1)) # Joining 2 different array

print(jo1D)
print(jo1D_same)
```

```
[2 6 8 9 7 4 1 3]
[2 6 8 9 2 6 8 9]
```

In [65]:
```
# Join in 2-D array
var_jo2D_1 = np.array([[2, 6, 8, 9], [7, 4, 1, 3]])
var_jo2D_2 = np.array([[7, 4, 1, 3], [2, 6, 8, 9]])

print("First 2D array:\n",var_jo2D_1)
print()
print("Second 2D array:\n",var_jo2D_2)
print()

jo2D_axis0 = np.concatenate((var_jo2D_1, var_jo2D_2)) # By default, it will take ax
jo2D_axis1 = np.concatenate((var_jo2D_1, var_jo2D_2), axis = 1)

print(jo2D_axis0)
print()
print(jo2D_axis1)
```

```
First 2D array:
 [[2 6 8 9]
 [7 4 1 3]]

Second 2D array:
 [[7 4 1 3]
 [2 6 8 9]]

[[2 6 8 9]
 [7 4 1 3]
 [7 4 1 3]
 [2 6 8 9]]

[[2 6 8 9 7 4 1 3]
 [7 4 1 3 2 6 8 9]]
```

In [66]:
```
# Join through stack() function
var_jo2D_1 = np.array([[2, 6, 8, 9], [7, 4, 1, 3]])
var_jo2D_2 = np.array([[7, 4, 1, 3], [2, 6, 8, 9]])

jo2D_stack = np.stack((var_jo2D_1, var_jo2D_2))
jo2D_dstack = np.dstack((var_jo2D_1, var_jo2D_2)) #height wise
jo2D_vstack = np.vstack((var_jo2D_1, var_jo2D_2)) # column wise
jo2D_hstack = np.hstack((var_jo2D_1, var_jo2D_2)) # row wise

print("This is joining through stack:\n",jo2D_stack)
print()
print("This is joining through height:\n",jo2D_dstack)
print()
print("This is joining through vertically:\n",jo2D_vstack)
print()
print("This is joining through horizontally:\n",jo2D_hstack)
print()
```

```
This is joining through stack:
 [[[2 6 8 9]
  [7 4 1 3]]

 [[7 4 1 3]
  [2 6 8 9]]]

This is joining through height:
 [[[2 7]
  [6 4]
  [8 1]
  [9 3]]

 [[7 2]
  [4 6]
  [1 8]
  [3 9]]]

This is joining through vertically:
 [[2 6 8 9]
 [7 4 1 3]
 [7 4 1 3]
 [2 6 8 9]]

This is joining through horizontally:
 [[2 6 8 9 7 4 1 3]
 [7 4 1 3 2 6 8 9]]
```

Split: We can split arrays into arrays of the same shape by indicating the position after which the split should occur.

Syntax: np.split(var_name, num) Num, to split how many parts

In [67]:
```
# Split in 1-D array
var_split = np.array([2, 6, 8, 9])

sp = np.split(var_split, 2) # 2 is for, I want to split the arrray in 2 parts

print(sp)
print(type(sp)) # Now, it's in list format
print(sp[0])
```

```
[array([2, 6]), array([8, 9])]
<class 'list'>
[2 6]
```

In [68]:
```
# Split in 2-D array
var_split_2D = np.array([[2, 6, 8, 9], [7, 4, 1, 3]])

sp = np.split(var_split_2D, 2) # We  have divide here equally
sp_ax = np.split(var_split_2D, 2, axis=1)

print(sp)
print()
print("In the basis of axis:\n",sp_ax)
print()
print(sp[0])
print()
print(sp_ax[0])
```

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

In the basis of axis:
 [array([[2, 6],
       [7, 4]]), array([[8, 9],
       [1, 3]])]

[[2 6 8 9]]

[[2 6]
 [7 4]]
```

Numpy Array Functions:

1. **Search Array:** You can search an array for a certain value, and return the indexes that get a match. **Syntax: np.where()**
2. **Search Sorted Array:** Used to find the index value where the new elements should be inserted in the sorted array, and the order of an array be preserved. If the same elements are already present in a sorted array, then we can specify to insert the left or right of it. It performs a binary search in array and then give the result. **Syntax: np.searchsorted()**
3. **Sort Array:** This function returns a sorted copy of an array. **Syntx: np.sort()**
4. **Filter Array:** Getting some elements out of an existing array and creating a new array out of them is called filtering.

In [69]:
```
# Search in 1-D array
var_search = np.array([2, 6, 8, 9, 8])
#index                 0, 1, 2, 3, 4

se = np.where(var_search == 8)
se_chng = np.where(var_search/6 == 1)

print(se)
print(se_chng)
```

```
(array([2, 4]),)
(array([1]),)
```

In [70]:
```
# Search sorted in 1-D array
var_searchsort = np.array([2, 6, 8, 9, 8])
#index                     0, 1, 2, 3, 4

se_sort = np.searchsorted(var_searchsort, 7)
se_sort_chng = np.searchsorted(var_searchsort, [7,8,9], side='right') # It will sea

print(se_sort)
print(se_sort_chng)
```

```
2
[2 3 5]
```

In [71]:
```
# Sort in 1-D array
# Here, we can't sort in descending format
var_sort = np.array([2, 6, 8, 9, 8])
var_sort_alpha = np.array(['S', 'U', 'B', 'H', 'A'])

sort = np.sort(var_sort)
sort_alpha = np.sort(var_sort_alpha)

print(sort)
print(sort_alpha)
```

```
[2 6 8 8 9]
['A' 'B' 'H' 'S' 'U']
```

In [72]:
```
# Sort in 2-D array
var_sort_2D = np.array([[2, 6, 8], [9, 8, 4]])
var_sort_alpha_2D = np.array([['S', 'U', 'B', 'H'], ['A', 'D', 'A', 'S']])

sort_2D = np.sort(var_sort_2D)
sort_alpha_2D = np.sort(var_sort_alpha_2D)

print(sort_2D)
print()
print(sort_alpha_2D)
```

```
[[2 6 8]
 [4 8 9]]

[['B' 'H' 'S' 'U']
 ['A' 'A' 'D' 'S']]
```

In [73]:
```
# Filter in 1-D array
# We can use this in string value also
var_filter = np.array([2, 6, 8, 9, 8])
f = [True, False, False, True, True]

filter = var_filter[f]

print(filter)
```

```
[2 9 8]
```

Data Manipulation Functions:

1. **Shuffle:** Modifies a sequence in-place by shuffling its contents. It only shuffles the array along the first axis of a multi-dimensional array. **Syntax: np.random.shuffle(var_name)**
2. **Unique:** Returns the sorted unique elements of an array. There are three optional outputs in addition to the unique elements: the indices of the input array that give the unique values, the indices of the unique array that reconstruct the input array, and the number of times each unique value comes up in the input array. **Syntax: np.unique(var_name)**
3. **Resize:** Returns a new array with the specified shape. If the new array is larger than the original array, then the new array is filled with repeated copies of a. Note that this behavior is different from a.resize(new_shape) which fills with zeros instead of repeated copies of a. **Syntax: np.resize(var_name,(shape_size))**
4. **Flatten:** Returns a copy of the array collapsed into one dimension. The 'order' parameter can specify the 'C' style (row-major) or 'F' style (column-major) to flatten. **Syntax: var_name.flatten()**
5. **Ravel:** Returns a contiguous flattened array. The 'order' parameter can specify the 'C' style (row-major) or 'F' style (column-major) to flatten. **Syntax: var_name.ravel()**

In Flatten, we can use orders:

- **C:** This is to flatten the array in row-major (C-style) order.
- **F:** This is to flatten the array in column-major (Fortran-style) order.
- **A:** This means to flatten in column-major order if a is Fortran contiguous in memory, row-major order otherwise.
- **K:** This means to flatten a in the order the elements occur in memory.

In [74]:
```
# Manipulation in 1-D Array
arr_1D = np.array([1, 2, 2, 4, 6, 6, 7, 8, 9])
print(arr_1D)
print()

# shuffle, Here we can't store value with shuffle parameter
np.random.shuffle(arr_1D)
print("Shuffled array:\n", arr_1D)
print()

# unique
unique_arr = np.unique(arr_1D, return_index = True, return_counts = True) # To get
print("Unique elements:\n", unique_arr)
print()

# resize
resized_arr = np.resize(arr_1D, (2, 4))
print("Resized array:\n", resized_arr)
print()

# flatten
flattened_arr = arr_1D.flatten() # By default it will take c-style
print("Flattened array:\n", flattened_arr)
print()

# ravel
raveled_arr = arr_1D.ravel() # By default it will take c-style
print("Raveled array:\n", raveled_arr)
```

```
[1 2 2 4 6 6 7 8 9]

Shuffled array:
 [6 2 2 6 8 9 1 7 4]

Unique elements:
 (array([1, 2, 4, 6, 7, 8, 9]), array([6, 1, 8, 0, 7, 4, 5]), array([1, 2, 1, 2,
1, 1, 1]))

Resized array:
 [[6 2 2 6]
 [8 9 1 7]]

Flattened array:
 [6 2 2 6 8 9 1 7 4]

Raveled array:
 [6 2 2 6 8 9 1 7 4]
```

In [75]:
```
# Manipulation in 2-D Array
arr_2D = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(arr_2D)
print()

# shuffle, Here we can't store value with shuffle parameter
np.random.shuffle(arr_2D)
print("Shuffled array:\n", arr_2D)
print()

# unique
unique_arr = np.unique(arr_2D, return_index = True, return_counts = True)
print("Unique elements:\n", unique_arr)
print()

# resize
resized_arr = np.resize(arr_2D, (2, 4))
print("Resized array:\n", resized_arr)
print()

# flatten
flattened_arr = arr_2D.flatten() # By default it will take c-style
print("Flattened array:\n", flattened_arr)
print()

# ravel
raveled_arr = arr_2D.ravel() # By default it will take c-style
print("Raveled array:\n", raveled_arr)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

Shuffled array:
 [[1 2 2]
 [7 8 9]
 [4 6 6]]

Unique elements:
 (array([1, 2, 4, 6, 7, 8, 9]), array([0, 1, 6, 7, 3, 4, 5]), array([1, 2, 1, 2,
1, 1, 1]))

Resized array:
 [[1 2 2 7]
 [8 9 4 6]]

Flattened array:
 [1 2 2 7 8 9 4 6 6]

Raveled array:
 [1 2 2 7 8 9 4 6 6]
```

In [76]:
```
# Flatten Function for 2-D Array
arr_2D_flat = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(arr_2D_flat)
print()

flatten_arr_c = arr_2D_flat.flatten(order = 'C')
flatten_arr_f = arr_2D_flat.flatten(order = 'F')
flatten_arr_a = arr_2D_flat.flatten(order = 'A')
flatten_arr_k = arr_2D_flat.flatten(order = 'K')

print("Flattened array of C:\n", flatten_arr_c)
print("Flattened array of F:\n", flatten_arr_f)
print("Flattened array of A:\n", flatten_arr_a)
print("Flattened array of K:\n", flatten_arr_k)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

Flattened array of C:
 [1 2 2 4 6 6 7 8 9]
Flattened array of F:
 [1 4 7 2 6 8 2 6 9]
Flattened array of A:
 [1 2 2 4 6 6 7 8 9]
Flattened array of K:
 [1 2 2 4 6 6 7 8 9]
```

In [77]:
```
# Ravel Function for 2-D Array
arr_2D_rev = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(arr_2D_rev)
print()

ravel_arr_c = arr_2D_rev.ravel(order = 'C')
ravel_arr_f = arr_2D_rev.ravel(order = 'F')
ravel_arr_a = arr_2D_rev.ravel(order = 'A')
ravel_arr_k = arr_2D_rev.ravel(order = 'K')

print("Ravel array of C:\n", ravel_arr_c)
print("Ravel array of F:\n", ravel_arr_f)
print("Ravel array of A:\n", ravel_arr_a)
print("Ravel array of K:\n", ravel_arr_k)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

Ravel array of C:
 [1 2 2 4 6 6 7 8 9]
Ravel array of F:
 [1 4 7 2 6 8 2 6 9]
Ravel array of A:
 [1 2 2 4 6 6 7 8 9]
Ravel array of K:
 [1 2 2 4 6 6 7 8 9]
```

Insert and Delete

Insert and Delete Functions:

1. **Insert:** Inserting a value in the array. **Syntax: np.insert(var_name, array_position, data)**

2. **Append:** It will add function in the last of the array. **Syntax: np.append(var_name, data)**
3. **Delete:** Delete a value with index number. **Syntax: np.delete(var_name, index_num)**

In [78]:
```
# Insert in 1-D Array
var_in = np.array([1, 2, 2, 4, 6, 6, 7, 8, 9])

print(var_in)
np.insert(var_in, 4, 9)
```

```
[1 2 2 4 6 6 7 8 9]
```

Out[78]:
```
array([1, 2, 2, 4, 9, 6, 6, 7, 8, 9])
```

In [79]:
```
# Insert in 1-D Array
var_in = np.array([1, 2, 2, 4, 6, 6, 7, 8, 9])

print(var_in)
np.insert(var_in, (0, 1, 4), 9)
```

```
[1 2 2 4 6 6 7 8 9]
```

Out[79]:
```
array([9, 1, 9, 2, 2, 4, 9, 6, 6, 7, 8, 9])
```

In [80]:
```
# Insert in 2-D Array
var_in_2D = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(var_in_2D)
print()

var_axis0 = np.insert(var_in_2D, 2, 5, axis = 0) # Inserting 1 data in row
var_axis1 = np.insert(var_in_2D, 2, (5, 6, 5), axis = 1) # Inserting 3 data in colu

print(var_axis0)
print()
print(var_axis1)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

[[1 2 2]
 [4 6 6]
 [5 5 5]
 [7 8 9]]

[[1 2 5 2]
 [4 6 6 6]
 [7 8 5 9]]
```

In [81]:
```
# Append in 1-D Array
var_ap = np.array([1, 2, 2, 4, 6, 6, 7, 8, 9])

print(var_ap)
np.append(var_ap, (80, 90))
```

```
[1 2 2 4 6 6 7 8 9]
```

Out[81]:
```
array([ 1,  2,  2,  4,  6,  6,  7,  8,  9, 80, 90])
```

In [82]:
```
# Append in 2-D Array
var_ap_2D = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(var_ap_2D)
print()

var_axis0 = np.append(var_ap_2D, [[5, 6, 5]], axis = 0)
# var_axis1 = np.append(var_ap_2D, [[6, 5, 6,]], axis = 1)

print(var_axis0)
# print(var_axis1)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

[[1 2 2]
 [4 6 6]
 [7 8 9]
 [5 6 5]]
```

In [83]:
```
# Delete in 1-D Array
var_del = np.array([1, 2, 2, 4, 6, 6, 7, 8, 9])

print(var_del)
np.delete(var_del, (2, 4, 6)) # (2, 4, 6) is the index num
```

```
[1 2 2 4 6 6 7 8 9]
```

Out[83]:
```
array([1, 2, 4, 6, 8, 9])
```

In [84]:
```
# Delete in 2-D Array
var_del_2D = np.array([[1, 2, 2], [4, 6, 6], [7, 8, 9]])
print(var_del_2D)
print()

var_axis0 = np.delete(var_ap_2D, (0,1,0), axis = 0)
var_axis1 = np.delete(var_ap_2D, (0,1,0), axis = 1)

print(var_axis0)
print(8)
print(var_axis1)
```

```
[[1 2 2]
 [4 6 6]
 [7 8 9]]

[[7 8 9]]
8
[[2]
 [6]
 [9]]
```

Matrix in Numpy Array: In the numpy library, a matrix is a two-dimensional data structure where numbers are arranged into rows and columns. Matrix and Array is same looking thing. Here the difference is in arithmetic functions. (Specially in Multiply)

In [85]:
```
# Printing a 2-D matrix
var_mat = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(var_mat)
print(type(var_mat))

print(var_mat.shape)
print(var_mat.ndim)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
<class 'numpy.matrix'>
(3, 3)
2
```

In [86]:
```
# Printing a 2-D array
var_ar = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
print(var_ar)
print(type(var_ar))

print(var_ar.shape)
print(var_ar.ndim)
```

```
[[1 2 3]
 [4 5 6]
 [7 8 9]]
<class 'numpy.ndarray'>
(3, 3)
2
```

In Matrix and Array result of Addition and Substraction are same

In [87]:
```
# Addition in 2-D matrix
var_mat_1 = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
var_mat_2 = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(var_mat_1 + var_mat_2)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

In [88]:
```
# Addition in 2-D array
var_arr_1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
var_arr_2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(var_arr_1 + var_arr_2)
```

```
[[ 2  4  6]
 [ 8 10 12]
 [14 16 18]]
```

In Matrix and Array result of Multiplication are not same because in Matrix multiplication, we have to do with row of a matrix and column of another matrix. **Syntax: var_1.dot(var_2)**

In [89]:
```
# Multiplication in 2-D matrix
var_mat_1 = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
var_mat_2 = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(var_mat_1.dot(var_mat_2))
```

```
[[ 30  36  42]
 [ 66  81  96]
 [102 126 150]]
```

In [90]:
```
# Multiplication in 2-D array
var_arr_1 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
var_arr_2 = np.array([[1, 2, 3], [4, 5, 6], [7, 8, 9]])

print(var_arr_1 * var_arr_2)
```

```
[[ 1  4  9]
 [16 25 36]
 [49 64 81]]
```

Functions of Matrix:

1. **Transpose**: The transpose of a matrix is obtained by flipping it over its diagonal, which switches its row and column indices. This means the first row of the original matrix becomes the first column of the transpose matrix, the second row becomes the second column, and so on. **Syntax: np.transpose(var_name) / var_name.T**
2. **Swapcase**: Swapcase is a string operation that converts lower case letters to upper case and vice versa. It's not typically used with matrices, but with arrays containing string data. **Syntax: np.swapaxes(var_name, 0, 1)) / np.char.swapcase(var_name)**
3. **Inverse**: The inverse of a matrix is a matrix that, when multiplied with the original matrix, yields the identity matrix (a square matrix with ones on the main diagonal and zeros elsewhere). Not all matrices have an inverse - only square matrices that are non-singular (i.e., their determinant is not zero) have an inverse. **Syntax: np.linalg.inv(var_name)**
4. **Power**: Raising a matrix to a power means multiplying it by itself a certain number of times. For example, a matrix to the power of 2 would be the result of multiplying that matrix by itself. **Syntax: np.linalg.matrix_power(var_name, int_value)**
5. **Determinant**: The determinant is a special number that can be calculated from a square matrix. It provides important information about the matrix, such as whether it has an inverse (if the determinant is zero, then the matrix does not have an inverse). The determinant can also be interpreted as the scaling factor when the matrix is used for a linear transformation. **Syntax: np.linalg.det(var_name)**

Transpose

In [91]:
```
# Transpose
var_tran = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
tran_mat = np.transpose(var_tran) # Also we can write, tran_mat = var_tran.T

print(var_tran)
print()
print(tran_mat)
```

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

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

Swapcase

In [92]:
```
# Swapcase
arr = np.array(['Hello', 'Subha'])
swap_case_arr = np.char.swapcase(arr)
print(swap_case_arr)
```

```
['hELLO' 'sUBHA']
```

In [93]:
```
# Swapcase
var_tran = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
swap_arr = np.swapaxes(var_tran, 0, 1)

print(var_tran)
print()
print(swap_arr)
```

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

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

Inverse

In [94]:
```
# Inverse
var_tran = np.matrix([[1, 2], [4, 5]])
inver_mat = np.linalg.inv(var_tran)

print(var_tran)
print()
print(inver_mat)
```

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

[[-1.66666667  0.66666667]
 [ 1.33333333 -0.33333333]]
```

Power

In [95]:
```
# Power
var_tran = np.matrix([[1, 2, 3], [4, 5, 6], [7, 8, 9]])
pow_mat = np.linalg.matrix_power(var_tran, 3) # It will multiply itself 3 times
pow_mat_0 = np.linalg.matrix_power(var_tran, 0) # Multiply 0 times

print(var_tran)
print()
print(pow_mat)
print()
print(pow_mat_0)
```

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

[[ 468  576  684]
 [1062 1305 1548]
 [1656 2034 2412]]

[[1 0 0]
 [0 1 0]
 [0 0 1]]
```

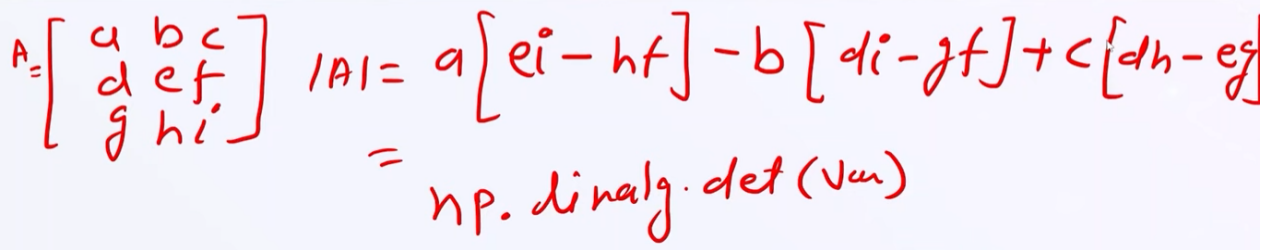
In [96]:
```
# Power with -3
var_tran = np.matrix([[1, 2], [4, 5]])
pow_mat_3 = np.linalg.matrix_power(var_tran, -3) # It will multiply itself -3 times

print(var_tran)
print()
print(pow_mat_3)
```

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

[[-7.88888889  2.88888889]
 [ 5.77777778 -2.11111111]]
```

Determinant



In [97]:
```
# Determinant
var_tran = np.matrix([[1, 2], [4, 5]])
deter = np.linalg.det(var_tran)

print(var_tran)
print()
print(deter)
```

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

-2.9999999999999996
```