NUMPY FUNDAMENTALS

WHAT IS NUMPY?

- NumPy is the fundamental package for scientific computing in Python.
- It is a Python library that provides a multidimensional array object, various derived objects (such as masked arrays and matrices), and it provide fast operations on arrays, including mathematical, logical, shape manipulation, sorting, selecting, I/O, discrete Fourier transforms, basic linear algebra, basic statistical operations, random simulation and much more.
- At the core of the NumPy package, is the ndarray object. This encapsulates n-dimensional arrays of homogeneous data types.

NUMPY ARRAYS VS PYTHON SEQUENCES

- NumPy arrays have a fixed size at creation, unlike Python lists (which can grow dynamically).
- Changing the size of a ndarray will create a new array and delete the original.
- The elements in a NumPy array are all required to be of the same data type (homogeneous), and thus will be the same size in memory.
- NumPy arrays facilitate advanced mathematical and other types of operations on large numbers of data.
- Typically, such operations are executed more efficiently and with less code than is possible using Python's built-in sequences.
- A growing plethora of scientific and mathematical Python-based packages / libraries like TensorFlow, Pandas, Scikit-learn etc. are using NumPy arrays. though these typically support Python-sequence input, they convert such input to NumPy arrays prior to processing, and they often output NumPy arrays.

CREATING NUMPY ARRAYS

- NumPy is used to work with arrays. The array object in NumPy is called ndarray.
- We can create a NumPy ndarray object by using the array() function.
- There are two ways to importing the NumPy module:
  - **import numpy as np**
  - **from numpy import ***

```
# Importing the NumPy library & rename as np
import numpy as np
a = np.array([1,2,3,4])
# It's output called as vector or 1D array
print('a :',a)
```

```
a : [1 2 3 4]
```

```
# Importing the NumPy library using 2nd way
from numpy import *
b = array([1,2,3,4])
print('b :',b)
```

```
b : [1 2 3 4]
```

CREATING 2D NUMPY ARRAYS

- 2D array are represented as collection of rows and columns.
- In machine learning and data science NumPy 2D array known as a matrix.
- Specially use to store and perform an operation on input values.

```
# 2D array (it is matrix)
from numpy import *
b = array([[1,2,3],[4,5,6]])
print(b)
```

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

CREATING 3D NUMPY ARRAYS

- In machine learning and data science NumPy 3D array known as a tensor.
- Specially used to store and perform an operation on three-dimensional data like colour image.

NOTE:

There are so many ways to create numpy arrays depending on situations for that we use other function that are provided by numpy library.

```
# 3D array (also called as tensor)
c = array([[[1,2,3]],[[5,6,7]],[[8,9,10]]])
print(c)
```

```
[[[ 1  2  3]]

 [[ 5  6  7]]

 [[ 8  9 10]]]
```

CREATING NUMPY ARRAY WITH DIFFERENT DATATYPE USING DTYPE

- It refers to the data type of elements stored in a NumPy array.
- Allows you to create arrays with different data types, such as integers, floating-point numbers, and more.
- When creating NumPy arrays, you can indeed specify the data type of the elements using the dtype parameter.
- Syntax: np.array([1,2,3] , dtype=float)

```
# dtype
import numpy as np
f = np.array([1,2,3,4] , dtype=float)
c = np.array([1,2,3,4] , dtype=complex)
# non integer value consider as True
tf = np.array([1,2,0,4] , dtype=bool)
print('float :',f)
print('complex :',c)
print('boolean :',tf)
```

```
float : [1. 2. 3. 4.]
complex : [1.+0.j 2.+0.j 3.+0.j 4.+0.j]
boolean : [ True  True False  True]
```

np.ones() function:

- Returns a new array of given shape and dtype, where the element's value is set to 1 & Default dtype is float.
- It is useful in deep learning to initialize the weights values.
- Syntax: np.ones(shape, dtype=None, order='C')

```
# np.ones() function
import numpy as np
o = np.ones((2,4), dtype=int)
print(o)
```

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

np.zeros() function:

- Returns a new array of given shape and type, where the element's value as 0.
- Default dtype is float .
- Syntax: np.zeros(shape, dtype=float, order='C')

```
# np.zeros() function
import numpy as np
z = np.zeros((3,3))
print(z)
```

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

np.arange() function:

- Used to create arrays containing regularly spaced values within a specified range.
- It generates values starting from start, up to (but not including) stop, with increments of step.
- If step is not provided, it defaults to 1.
- Syntax: np.arange(start, stop, step, dtype=None)

```
# np.arange()
import numpy as np
# with start and stop argument
e = np.arange(5,10)
print(e)
# with start, stop and step argument
e1 = np.arange(5,10,2)
print(e1)
```

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

np.reshape() function:

- Used to change the shape (dimensions) of an array without changing its data.

- Returns a new array with the same data but with a different shape.
- Useful when we want to convert a 1D array into a two-dimensional array or vice versa.
- It can also be used to create arrays with a specific shape, such as matrices and tensors.
- **Syntax:** np.reshape(arr, new_shape, order='C')
  ~ a: input array.
  ~ new_shape: shape of new array
  ~ order: {'C', 'F', 'A'}, optional

```
# np.reshape() function
import numpy as np
r = np.arange(1,13)
re = np.reshape(r,(2,6))
print(re)
re1 = np.reshape(r,(6,2))
print(re1)
re3 = np.reshape(r,(3,4))
print(re3)
```

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

NOTE
New array dimension product is equal to number of items that are present in inside the original array.

np.ones() function:
- Returns a new array of given shape and dtype, where the element's value is set to 1.
- Default dtype is float.
- It is useful in deep learning to initialize the weights values
- **Syntax:** np.ones(shape, dtype, order='C')

```
# np.ones() function
import numpy as np
o = np.ones((2,4), dtype=int)
print(o)
```

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

np.zeros() function:
- Returns a new array of given shape and type, where the element's value as 0.
- Default dtype is float .
- **Syntax:** np.zeros(shape, dtype=float, order='C')

```
# np.zeros() function
import numpy as np
z = np.zeros((3,3))
print(z)
```

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

np.random.random() function:
- Used for generating random numbers.
- Here 1st random is class name and other one is method name follows OOP concept.
- **Syntax:** np.random.random(shape, dtype)

```
# np.random.random() function
import numpy as np
# creating default random varible between 0 to 1
r = np.random.random((3,3))
print(r)
```

```
[[0.32138088 0.05043471 0.29236916]
 [0.67078606 0.7525749  0.51963277]
 [0.85785698 0.07023304 0.84409173]]
```

np.linspace() function : (Linear/ linearly space)
- Returns evenly spaced numbers over a specified interval.
- Use for plotting the ML algorithm result.
- **Syntax:** np.linspace(start, stop, num=50, dtype=float, axis=0)
  ~ start: starting value of the sequence
  ~ stop: end value of the sequence
  ~ num: number for spacing & default is 50
  ~ axis: axis for evenly spaced numbers & default is 0.

```
# np.linspace() function
import numpy as np
ls = linspace(1,10,5)
print(ls)
```

```
[ 1.    3.25  5.5   7.75 10.  ]
```

np.identity() function:
- Returns a square identity matrix of size n x n means diagonally items are 1 and remain all numbers becomes 0's
- **Syntax:** np.identity(n, dtype=float)
  ~ n: size of the identity matrix
  ~ dtype: we can use another datatype

```
# np.identity() functioin
import numpy as np
i = np.identity(3, dtype=int)
print(i)
```

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

ATTRIBUTES OF NUMPY ARRAYS
- NumPy array is the most used construct of numpy in machine learning and deep learning.
- Let us look into some important attributes of this numpy array.

```
# Arrays for applying numpy arrays attributes
import numpy as np
a1 = np.arange(10)
a2 = np.arange(12, dtype=float).reshape(3,4)
a3 = np.arange(8).reshape(2,2,2)
print(a1)
print(a2)
print(a3)
```

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

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

arr.ndim attribute:
- Returns the number of dimensions of a given numpy array.

```
# arr.ndim attribute
print(a1.ndim)
print(a2.ndim)
print(a3.ndim)
```

```
1
2
3
```

arr.shape attribute:
- Determine the dimensions of the array and returns a tuple of integers that represent the size of the array in each dimension.

```
# arr.shape attribute
print(a1.shape)
print(a2.shape)
# Made with two 2D array of shape 2, 2
print(a3.shape)
```

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

arr.size attribute:
- Returns the total number of elements in the array.

```
# arr.size attribute
print(a1.size)
print(a2.size)
print(a3.size)
```

```
10
12
8
```

arr.itemsize attribute:

- Returns the size (in bytes) of each element in the array.

```
# arr.itemsize attribute
print(a1.itemsize) # int32 = 4 bytes
print(a2.itemsize) # int64 = 8 bytes
print(a3.itemsize) # int32 = 4 bytes
```

```
4
8
4
```

arr.dtype attribute:

- Returns the datatype of the elements in the array.

```
# arr.dtype attribute
print(a1.dtype)
print(a2.dtype)
print(a3.dtype)
```

```
int32
float64
int32
```

Changing datatype using .astype() method:

- Change the data type of the elements in the array.
- More useful in converting the float datatype reduction in integer value.

```
# arr.astype() method
print(a2)
print('a2 dtype :',a2.dtype)
change_dtype = a2.astype(int32)
print(change_dtype)
print('a2 dtype :',change_dtype.dtype)
```

```
[[ 0.  1.  2.  3.]
 [ 4.  5.  6.  7.]
 [ 8.  9. 10. 11.]]
a2 dtype : float64
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
a2 dtype : int32
```

NUMPY ARRAY OPERATIONS

Use for performing mathematical operations

```
# Creating the arrays for operations
from numpy import *
a1 = arange(12).reshape(3,4)
a2 = arange(12,24).reshape(3,4)
print(a1)
print(a2)
```

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

SCALAR OPERATIONS

- Scalar operation is an operation between a scalar value (a single number) and an array.
- It can be performed using arithmetic operators such as +, -, *, and /.
- Scalar value perform operation with each individual element in the array.

```
# scalar operations, here 2 scalar value
from numpy import *
print(a1 + 2) # addition of each element
print(a1 * 2) # multiplication of each element
print(a1 ** 2) # exponent of each element
```

```
[[ 2  3  4  5]
 [ 6  7  8  9]
 [10 11 12 13]]
[[ 0  2  4  6]
 [ 8 10 12 14]
 [16 18 20 22]]
[[  0   1   4   9]
 [ 16  25  36  49]
 [ 64  81 100 121]]
```

COMPARISON / RELATIONAL OPERATIONS

- NumPy provides comparison operators such as ==, <, >, <=, >= etc. for comparing elements in two arrays.
- This operations return a boolean array with the same shape as the input arrays.
  ~ True indicates condition is satisfied
  ~ False indicates condition not satisfied

```
# comparison operators
print(a1 >= 0)
print(a1 == 2)
```

```
[[ True  True  True  True]
 [ True  True  True  True]
 [ True  True  True  True]]
[[False False  True False]
 [False False False False]
 [False False False False]]
```

VECTOR OPERATIONS

- A vector operation is an operation between two arrays of the same size.
- Vector operations can also be performed using arithmetic operators.
- When two arrays are added, the corresponding elements in each array are added together & also similarly for another operations.

```
# vector Operations
print(a1 + a2) # addition
print(a1 * a2) # multiplication
```

```
[[12 14 16 18]
 [20 22 24 26]
 [28 30 32 34]]
[[  0  13  28  45]
 [ 64  85 108 133]
 [160 189 220 253]]
```

NOTE

Vector operations can only be performed on arrays of the same shape. If the arrays have different shapes, NumPy will raise a ValueError.

NUMPY ARRAY FUNCTIONS

Some common NumPy array function that is use in machine learning and deep learning etc.

NUMPY ARRAY MATHEMATICAL FUNCTIONS:

Use for performing mathematical function

```
# Creating arrays for numpy mathematical function
import numpy as np
a1 = np.random.random((3,3))
a1 = np.round(a1*100)
print(a1)
```

```
[[58. 75. 51.]
 [81.  9. 58.]
 [13. 71. 68.]]
```

np.min() & np.max() function:

- Return the minimum and maximum of element in array.
- But we can also use NumPy min and max to compute the minima and maxima of each column and rows.
  ~ column-wise represents axis=0
  ~ row-wise represents axis=1

```
# min() & max()
print(np.min(a1))
print(np.max(a1))
```

```
9.0
81.0
```

```
# min() & max() with axis parameter
# column wise each minimum element
print('min :',np.min(a1, axis=0))
# row wise each maximum element
print('max :',np.max(a1, axis=1))
```

```
min : [13.  9. 51.]
max : [75. 81. 71.]
```

np.sum() function:

- Used to calculate the sum of elements in a NumPy array.
- Also used to find the sum of all elements in the array or along a specific axis of a multi-dimensional array.
  ~ column-wise represents axis=0
  ~ row-wise represents axis=1

```
# sum()
print('sum of array :', np.sum(a1))
# sum() with axis parameter
print('sum of cols :',np.sum(a1, axis=0))
print('sum of rows :',np.sum(a1, axis=1))
```

```
sum of array : 484.0
sum of cols : [152. 155. 177.]
sum of rows : [184. 148. 152.]
```

np.prod() function:

- Return the product of all element in an array.
- Along with the axis parameter to calculate the product along a specific axis of a multi-dimensional array.
- It is a common operation in various mathematical and statistical calculations.

```
# prod()
print('product of array :', np.prod(a1))
# prod() with axis parameter
print('product of cols :',np.prod(a1, axis=0))
print('product of rows :',np.prod(a1, axis=1))
```

```
product of array : 588742745338800.0
product of cols : [ 61074.  47925. 201144.]
product of rows : [221850.  42282.  62764.]
```

np.round() function:

- Used to rounds the elements of an array to the nearest integer or to a specified number of decimals.

```
# round()
import numpy as np
arr = np.array([1.23,2.49,4.51])
print(np.round(arr))
```

```
[1. 2. 5.]
```

np.ceil() function:

- Used to rounds the elements of an array up to the nearest integer.

```
# ceil()
import numpy as np
arr3 = np.array([1.23,2.49,4.11,1.00])
print(np.ceil(arr3))
```

```
[2. 3. 5. 1.]
```

np.floor() function:

- Used to rounds the elements of an array down to the nearest integer.

```
# floor()
import numpy as np
arr1 = np.array([1.23,2.49,4.51,4.80])
print(np.floor(arr1))
```

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

```
print(np.floor(np.random.random((2,3))*100))
```

```
[[73.  1.  0.]
 [ 1. 82. 34.]]
```

np.log() function:

- To calculate the natural logarithm of an array or a scalar.

```
import numpy as np
ar = np.arange(1,10).reshape(3,3)
# log()
print(np.log(ar))
```

```
[[0.         0.69314718 1.09861229]
 [1.38629436 1.60943791 1.79175947]
 [1.94591015 2.07944154 2.19722458]]
```

np.exp() function:

- To calculate the exponential of an array or a scalar.

```
import numpy as np
ar = np.arange(1,10).reshape(3,3)
# exp()
print(np.exp(ar))
```

```
[[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]]
```

np.dot() function:

- Function takes two array arguments and returns their dot product.
- The dot product of two vectors and specifying the condition that they must have the same dimensionality.

```
# Creating two the arrays with same dimensions
import numpy as np
arr1 = np.arange(12).reshape(3,4)
arr2 = np.arange(12,24).reshape(4,3)
print(arr1)
print(arr2)
```

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

```
# dot()
print(np.dot(arr1,arr2))
```

```
[[114 120 126]
 [378 400 422]
 [642 680 718]]
```

NUMPY ARRAY STATISTICAL FUNCTIONS

Here, only a few functions related to statistics have been introduced. We will cover the remaining functions in the statistics session.

np.mean() function:

- Used to calculate the arithmetic mean or average of the elements in each array.
- The mean is the sum of all the values in the array divided by the total number of values.
- It is a common measure of central tendency.

```
# mean()
import numpy as np
arr = np.array([[2,5,11],[9,98,93],[17,21,40]])
print('mean :',np.mean(arr))
```

```
mean : 32.888888888888886
```

np.median() function:

- Used to calculate the median of the elements in an array.
- The median is the middle value of an array when it is ordered.
- It is a measure of central tendency that is less affected by outliers than the mean.

```
# median
print('median',np.median(arr))
```

```
median 17.0
```

np.var() function:

- Used to calculate the variance of the elements in an array.
- Variance is a measure of how much the values in a dataset vary from the mean.
- It gives you an idea of the spread or dispersion of the data points.

```
# var()
print('variance :',np.var(arr))
```

```
variance : 1230.9876543209875
```

np.std() function:

- Used to calculate the standard deviation of the elements in an array.
- The standard deviation is a measure of how much the values in a dataset deviate from the mean.
- It is another measure of the spread or dispersion of the data points, like variance.

```
# std()
data = np.array([5, 10, 15, 20, 25])
print("SD:", np.std(arr))
```

```
SD: 35.08543364875212
```

INDEXING IN NUMPY

- In NumPy, each element in an array is associated with a number. The number is known as an array index.
- NumPy array indexing refers to the process of accessing elements or subarrays within a NumPy array.
- In short, fetching the element from an array.

NOTE: Array start form 0 index.

1D INDEXING IN NUMPY ARRAY

- NumPy array indexing is used to access values in the 1D & multi-dimensional arrays.
- Indexing is an operation, use this feature to get a selected set of values from a NumPy array.
- It just like normal indexing like list and, we can you positive or negative indexing.
  ~ positive indexing: array start from 0 index position
  ~ negative indexing: array start from end -1 index position
- **Syntax:** array[index_position]

```
import numpy as np
arr1 = np.arange(10)
print('array :',arr1)
# positive indexing
print(arr1[0],arr1[4],arr1[6],arr1[2])
# negative indexing
print(arr1[-1],arr1[-5],arr1[-3],arr1[-7])
```

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

2D INDEXING IN NUMPY ARRAY

- 2D numpy arrays are like a table with rows and columns.
- For accessing elements, we need to specify the row index and column index of the element.
- **Syntax:** array[row_index , column_index_that_row]

```
import numpy as np
arr2 = np.arange(12).reshape(3,4)
print('2D numpy array:')
print(arr2)
print('Accessing elements:')
print(arr2[2,3], arr2[1,0], arr2[2,1])
```

```
2D numpy array:
[[ 0  1  2  3]
 [ 4  5  6  7]
 [ 8  9 10 11]]
Accessing elements:
11 4 9
```

NOTE: Array rows & cols start from 0 index.

3D INDEXING IN NUMPY ARRAY

- 3D numpy arrays are like a table with rows and columns.
- For accessing elements, we need to specify the row index and column index of the element.
- **Syntax:** array[arr_index , row_index , column_index_of_row]

```
import numpy as np
arr3 = np.arange(8).reshape(2,2,2)
print(arr3)
```

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

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

```
# 3D Indexing in numpy array
print(arr3[1,1,0],arr3[0,1,1],arr3[0,0,1])
```

```
6 3 1
```

NOTE: Array rows & cols start from 0 index.

SLICING IN NUMPY

- NumPy array slicing is used to extract some portion of data from the actual array.
- NumPy slicing is slightly different.
- Slicing can be done with the help of (:).
- **Syntax:** array[start : stop : step]
  ~ start: index by default considers as '0'
  ~ stop: index considers as a length of the array.
  ~ step: default is '1'.

1D SLICING IN NUMPY AARRY

- For 1D numpy arrays we use basic slicing, step slicing and omitting the indices.

```
import numpy as np
arr = np.arange(10)
print(arr)
```

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

```
print('Slicing with start:',arr[6:])
print('Slicing with stop:',arr[:4])
print('Slicing with step:',arr[::3])
print('Slicing with start & stop:',arr[2:6])
print('Slicing with start,stop & step:',arr[1:8:2])
print('Negative start & stop slicing:',arr[-6:-1])
```

```
Slicing with start: [6 7 8 9]
Slicing with stop: [0 1 2 3]
Slicing with step: [0 3 6 9]
Slicing with start & stop: [2 3 4 5]
Slicing with start,stop & step: [1 3 5 7]
Negative start & stop slicing: [4 5 6 7 8]
```

2D SLICING IN NUMPY ARRAY

- A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.
- To slice a 2D NumPy array, we can use the same syntax as for slicing a 1D NumPy array.
- The only difference is that we need to specify a slice for each dimension of the array and use comma ',' for separating the rows and columns.
- **Syntax:**

array[row_start : row_stop : row_step , col_start : col_stop : col_step]
~ row_start: specifies starting index
~ row_stop: stopping index
~ row_step: step size for the rows respectively
~ col_start: specifies starting index
~ col_stop: stopping index
~ col_step: step size for the columns respectively

```
import numpy as np
arr = np.arange(12).reshape(3,4)
print(arr)
```

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

```
# Slicing 2D array
print(arr[0,:]) # print row
print(arr[2,:]) # print 3rd index row
print(arr[:,3]) # print 4th index column
```

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

```
# slicing sub-arrays
print(arr[1:,1:3])
print(arr[::2,1::2])
print(arr[::2,::3])
print(arr[0:2,1:])
```

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

3D SLICING IN NUMPY ARRAY

- A 2D NumPy array can be thought of as a matrix, where each element has two indices, row index and column index.

```
# Slicing 3D numpy array
import numpy as np
arr = np.arange(27).reshape(3,3,3)
print(arr)
```

```
[[[ 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]]]
```

```
# print 2nd position element of 3D array
print(arr[1])
```

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

```
# print 1st elements of 0 index
print(arr[0, 1])
# print 2nd column of index 1
print(arr[1,:,1])
```

```
[3 4 5]
[10 13 16]
```

```
print(arr[2,1:,1:])
```

```
[[22 23]
 [25 26]]
```

```
print(arr[::2,0,::2])
```

```
[[ 0  2]
 [18 20]]
```

RESHAPING IN NUMPY

In reshaping we commonly use reshape() and transpose() but sometimes we need to use revel() function.

```
# creating array
import numpy as np
arr = np.arange(1,10).reshape(3,3)
print(arr)
```

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

np.ravel() function:

- Converting the n-dimensional array into flatten (1D) array.
- **Syntax:** arr.ravel() or np.ravel(arr)

```
print('Original 2D array :')
print(arr)
print('Conveting into 1D array using ravel() :')
print(arr.ravel())
```

```
Original 2D array :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Conveting into 1D array using ravel() :
[1 2 3 4 5 6 7 8 9]
```

np.transpose() or .T function :

- Applied on 2D arrays to swipe the rows and columns of an array.
- Using transpose() function or we can also use the short name .T to transpose a 2D array.
- **Syntax:** arr.transpose() or arr.T

```
# example of transpose()
print('Original 2D array :')
print(arr)
print('Transpose of 2D Array :')
print(arr.transpose()) # OR print(arr.T)
```

```
Original 2D array :
[[1 2 3]
 [4 5 6]
 [7 8 9]]
Transpose of 2D Array :
[[1 4 7]
 [2 5 8]
 [3 6 9]]
```

ITERATION ON NUMPY ARRAY

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.

ITERATION ON 1D NUMPY ARRAY

- It will go through each element one by one.

```
import numpy as np
# For loop on 1D array :
arr = np.arange(10)
for ele in arr:
    print(ele, end=' ')
```

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

ITERATION ON 2D NUMPY ARRAY

- It will go through all the rows.

```
import numpy as np
# For loop on 2D array
arr = np.arange(12).reshape(3,4)
for ele in arr:
    print(ele)
```

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

ITERATION ON 3D NUMPY ARRAY

- It will go through all the 2-D arrays.

```
import numpy as np
# For loop on 3D array
import numpy as np
arr = np.array([[[1, 2],[4, 5]],[[7, 8],[10, 11]]])
for ele in arr:
    print(ele)
```

```
[[1 2]
 [4 5]]
[[ 7  8]
 [10 11]]
```

np.nditer() function:

- It is a NumPy function that provides an efficient way to iterate over elements of a NumPy array.
- It allows iterating over multiple arrays simultaneously and provides a number of optional arguments that can be used to customize the iteration process.

```
# nd.iter() function
import numpy as np
arr = np.array([[[1, 2],[4, 5]],[[7, 8],[10, 11]]])
for ele in np.nditer(arr):
    print(ele, end=' ')
```

```
1 2 4 5 7 8 10 11
```

STACKING IN NUMPY

- Stacking is the concept of joining arrays in NumPy.
- Arrays having the same dimensions can be stacked.
- We can stack arrays along different axes using the functions.
  - ~ **column-wise represents axis=0**
  - ~ **row-wise represents axis=1**
  - ~ **np.hstack() :** horizontal stacking
  - ~ **np.vstack() :** vertical stacking
- Sometimes we have multiple data source means data come from databases, API and another data comes from web scrapping etc. so that data is similar data for multiple sources then we can stack the data for data analysis.

```
# creating two arrays for hstack() and vstack():
import numpy as np
arr1 = np.arange(12).reshape(3,4)
arr2 = np.arange(12,24).reshape(3,4)
```

NOTE: Shape/dimension of the array should be same

np.hstack() function:

- Horizontal stacking concatenates the arrays in sequence horizontally (column-wise).
- This function stacks arrays horizontally (along axis 1)
- **Syntax:** np.hstack((arr1, arr2))

```
# Ex. of np.hstack()
print(np.hstack((arr1,arr2)))
```

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

np.vstack() function:

- Vertical stacking means concatenates the arrays in sequence vertically (row-wise).
- This function stacks arrays vertically (along axis 0).
- **Syntax:** np.vstack((arr1, arr2))

```
# Ex. of np.vstack()
print(np.vstack((arr1, arr2)))
```

```
[[ 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 IN NUMPY

- Splitting is reverse operation of stacking.
- We can split the arrays into sub-arrays of the same shape
  - ~ **np.hsplit():** horizontal splitting
  - ~ **np.vsplit():** vertical splitting.

```
# creating array for hsplit() and vsplit() :
import numpy as np
arr = np.arange(1,13).reshape(3,4)
print(arr)
```

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

NOTE: Only used to split an array into sub-arrays of equal size

np.hsplit() function:

- hsplit() function is used to split a numpy array into multiple sub-arrays horizontally (column-wise).
- Pass the input array and the number of sub-arrays as arguments.
- **Syntax:** np.split(arr, sub_arrays_size)
  - ~ arr: input array
  - ~ sub_arrays_size: number for splitting the array

```
# Ex. of hsplit()
print(np.hsplit(arr, 2))
```

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

np.vsplit() function:

- vsplit() function is used to split a numpy array into multiple sub-arrays vertically (row-wise).
- Pass the input array and the number of sub-arrays as arguments.
- **Syntax:** np.vsplit(arr, sub_arrays_size)
  - ~ arr: input array
  - ~ sub_arrays_size: number for splitting the array

```
# Ex. of vsplit()
print(np.vsplit(arr, 3))
```

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

ADAVANCED NUMPY

NUMPY ARRAYS VS PYTHON LIST

let's compare NumPy arrays and Python lists based on the factors you mentioned: speed, memory, and convenience.

Speed:

- NumPy arrays are generally faster than Python lists for numerical operations due to their fixed data type and memory layout.
- NumPy operations are optimized for performance using low-level C libraries.

```
# python list
a = [i for i in range(10000000)]
b = [i for i in range(10000000,20000000)]
import time
c = []
start = time.time()
for i in range(len(a)):
    c.append(a[i]+b[i])
print(time.time()-start)
```

```
4.092959642410278
```

```
# python numpy
import numpy as np
a = np.arange(10000000)
b = np.arange(10000000,20000000)
start = time.time()
c = a + b
print(time.time()-start)
```

```
0.10450100898742676
```

Memory:

- NumPy arrays use less memory compared to Python lists, especially for large datasets, due to their efficient memory layout and data type specification.

```
# python list
import sys
a = [i for i in range(10000000)]
print(sys.getsizeof(a))
```

```
89095160
```

```
# python numpy
import sys
# default float
a = np.arange(10000000,dtype=np.int32)
print(sys.getsizeof(a))
```

```
40000112
```

Convenience:

- Writing code with NumPy is often more concise and intuitive for numerical operations compared to using plain Python lists.

In summary, if we dealing with numerical computations and performance is crucial, NumPy arrays are a better choice due to their speed and memory efficiency. However, if you need a more flexible and versatile data structure, Python lists might be more convenient.

FANCY INDEXING IN NUMPY

- It allowing us to use an array or a list of indices rather than using a slice or a single integer index.
- More advanced indexing and selection of elements from an array.
- To perform fancy indexing, we can use an array or a list of indices to select specific elements or subarrays from an array.
- More useful in pandas.
- **Syntax:**
  ~ Row-wise: array[[row_indices]]
  ~ Column-wise: array[: , [column_indices]]

```
import numpy as np
arr = np.arange(20).reshape(5,4)
print(arr)
```

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

```
print(arr[[0,2,4]]) # row-wise
```

```
[[ 0  1  2  3]
 [ 8  9 10 11]
 [16 17 18 19]]
```

```
print(arr[:,[0,2,3]]) # column-wise
```

```
[[ 0  2  3]
 [ 4  6  7]
 [ 8 10 11]
 [12 14 15]
 [16 18 19]]
```

np.random.randint() function:

- It is used to generate a random integer within a specified range and shape.
- **Syntax:** np.random.randint(low, high, size, dtype=int)
  ~ low: lowest integer to be drawn from the distribution & it is inclusive
  ~ high: If high is not None, one integer is drawn from the range [low, high). If high is None, one integer is drawn from the range [0, low).
  ~ size: shape of the output array
  ~ dtype: datatype of the output array

```
import numpy as np
arr = np.random.randint(1,70,16).reshape(4,4)
print(arr)
```

```
[[ 4 48 43  9]
 [50 12 40 24]
 [38 65 67  7]
 [32  2 49 68]]
```

NOTE:
Output array dimension number product is equal to number of items that are present in inside the original array.

BOOLEAN INDEXING IN NUMPY

- It is way of selecting elements from an array based on a boolean condition.
- Boolean mask is a numpy array containing truth values (True/False) that correspond to each element in the array.
- Boolean masking allows for the filtering of values in numpy arrays.

NOTE:
For condition we use any operator that is satisfied the boolean condition like relational or bitwise operator etc.

```
# find all numbers less than and equal 40
import numpy as np
# boolean mask condition
bool_mask = arr <= 40
print('Boolean mask :')
print(bool_mask)
print('Boolean condition :',arr[bool_mask])
```

```
Boolean mask :
[[ True False False  True]
 [False  True  True  True]
 [ True False False  True]
 [ True  True False False]]
Boolean condition : [ 4  9 12 40 24 38  7 32  2]
```

- More examples of boolean condition:

```
# 1.find out even numbers
print(arr[arr%2 == 0])

# 2.find all numbers greater than 50 and are even
# Here we use bitwise because of boolean
print(arr[(arr%2 == 0) & (arr>50)])

# 3.find all numbers not divisible by 7
print(arr[(arr%7 != 0)])
```

```
[ 4 48 50 12 40 24 38 32  2 68]
[68]
[ 4 48 43  9 50 12 40 24 38 65 67 32  2 68]
```

BROADCASTING IN NUMPY

- An array with a smaller shape is expanded to match the shape of a larger one, this is called broadcasting.
- The term broadcasting describes how numpy treats arrays with different shapes during arithmetic operations.
- Smaller array is "broadcast" across the larger array so that they have compatible shapes.
- Use in vectorization.

```
# same shape
import numpy as np
a = np.arange(6).reshape(2,3)
b = np.arange(6,12).reshape(2,3)
print(a)
print(b)
print('Addtion with same shape:')
print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[ 6  7  8]
 [ 9 10 11]]
Addtion with same shape:
[[ 6  8 10]
 [12 14 16]]
```

```
# different shape
import numpy as np
a = np.arange(6).reshape(2,3)
b = np.arange(3).reshape(1,3)
print(a)
print(b)
print('Addtion with different shape:')
print(a+b)
```

```
[[0 1 2]
 [3 4 5]]
[[0 1 2]]
Addtion with different shape:
[[0 2 4]
 [3 5 7]]
```

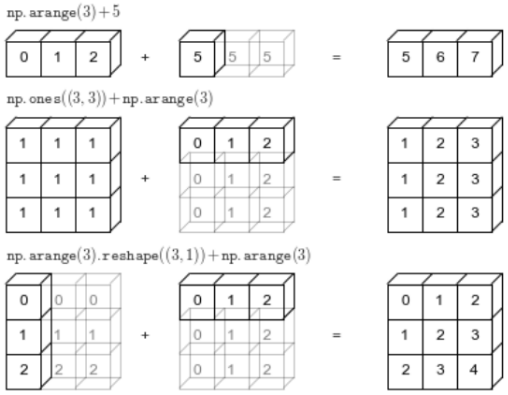
RULES FOR BROADCASTING

1.Make the two arrays have the same number of dimensions.

- If the numbers of dimensions of the two arrays are different, add new dimensions with size 1 to the head of the array with the smaller dimension.
  ~ Ex.1: (3,2) & (3) → (3,2) & (1,3)
  ~ Ex.2: (3,3,3) & (3) → (3,3,3) & (1,1,3)

2.Make each dimension of the two arrays the same size.

- If the sizes of each dimension of the two arrays do not match, dimensions with size 1 are stretched to the size of the other array.
- If there is a dimension whose size is not 1 in either of the two arrays, it cannot be broadcasted, and an error is raised.
  ~ Ex.1: (3,2) & (3) → (3,2) & (1,3) → (3,2) & (3,3)
  ~ Ex.2: (3,3,3) & (3) → (3,3,3) & (1,1,3) → (3,3,3) & (3,3,3)



- More examples to understanding broadcasting:

```
import numpy as np
n1 = np.arange(3) + 5
print(n1)
```

```
[5 6 7]
```

```
n1 = np.ones((3,3), dtype=int) + np.arange(3)
print(n1)
```

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

```
n1 = np.ones((3,3), dtype=int) + np.arange(3)
print(n1)
```

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

```
n1 = np.arange(3).reshape((3,1)) + np.arange(3)
print(n1)
```

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

```
n1 = np.arange(12).reshape(4,3)
n2 = np.arange(3)
print(n1+n2)
```

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

- When shapes are not valid:

```
# ValueError: not be broadcast shapes (3,4) (3,)
import numpy as np
n1 = np.arange(12).reshape(3,4)
n2 = np.arange(3)
print(n1+n2)
```

```
# ValueError: not be broadcast shapes (4,3) (3,4)
import numpy as np
n1 = np.arange(12).reshape(3,4)
n2 = np.arange(12).reshape(4,3)
print(n1+n2)
```

```
# ValueError: not be broadcast shapes (4,4) (2,2)
import numpy as np
n1 = np.arange(16).reshape(4,4)
n2 = np.arange(4).reshape(2,2)
print(n1+n2)
```

WORKING WITH MATHEMATICAL FORMULAS IN NUMPY ARRAY

- For calculating the uncommon function that are not in build-in function in numpy library but we create our own function here let's discuss some mathematical formulas that are used in data science.

Sigmoid function:

- The sigmoid function is often used in logistic regression and artificial neural networks to introduce non-linearity.
- Calculating each item sigmoid
- Use in Deep learning and Machine learning algorithms
- Sigmoid range between 0 to 1
- Formula: 1 / (1+e^-x)

```
import numpy as np
def Sigmoid(array):
    return 1/(1+np.exp(-(array)))
a = np.arange(10)
s = Sigmoid(a)
print(s)
```

```
[0.5        0.73105858 0.88079708 0.95257413 0.98
201379 0.99330715
 0.99752738 0.99908895 0.99966465 0.99987661]
```

Mean squared error (MSE):

- In data science and machine learning to measure the average squared difference between the predicted values and the actual values.
- It is often used to assess the performance of regression models.
- It is loss function.

```
actual  = np.random.randint(1,50,25)
predicted  = np.random.randint(1,50,25)
```

```
print(predicted)
```

```
[29 22 42  9 13 38 39 40 44 46 46  7 42 29  4  8  7 2
6  8  6 12 10 13 39
 32]
```

```
print(actual)
```

```
[32 49 37 44 47 11 29  8 20 38  5 27 25 43 29  3 42
8 31  5 27 37 41 37
 38]
```

```
print(np.mean((actual - predicted)**2))
```

```
508.4
```

```
def MSE(atcual, predicted):
    return np.mean((actual-predicted)**2)
MSE(actual , predicted)
```

```
508.4
```

WORKING WITH MISSING VALUES

- Dealing with missing values is a common task in data analysis and machine learning.
- Numpy provides a few ways to handle missing values.

np.nan:

- NumPy has a special value called NaN (Not a Number) that can represent missing values or undefined data in arrays.

np.isnan() function :

- Returns a boolean array where True indicates a NaN value.

```
# working with missing values > np.nan
a = np.array([1,2,3,np.nan,4,np.nan,5])
print("a :",a)

# identfiying missing value using np.isnan
b = np.isnan(a)
print("boolean array :",b)
print('b :',a[~(np.isnan(a))]) # boolean indexing
```

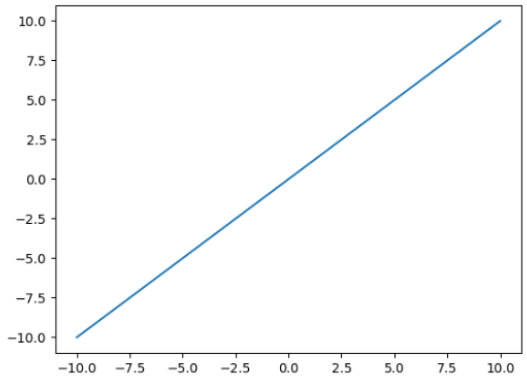
```
a : [ 1.  2.  3. nan  4. nan  5.]
boolean array : [False False False  True False  True
False]
b : [1. 2. 3. 4. 5.]
```

PLOTTING GRAPHS

We can use NumPy in combination with Matplotlib to create and plot graphs.

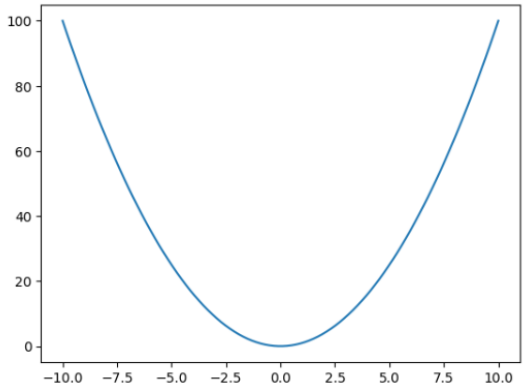
```
# plotting 2D plot
# x = y (straight line)
import numpy as np
import matplotlib.pyplot as plt
x = np.linspace(-10,10,100)
y = x
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f92229c160>]
```



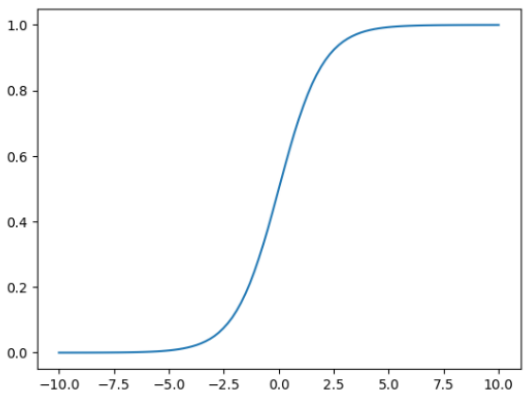

```
# parabola : The shape of this graph is a parabola.
# y = x^2
x = np.linspace(-10,10,100)
# applying scalar operation on x
y = x**2 # each element of x is square
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f922387550>]
```




```
# sigmoid graph
x = np.linspace(-10,10,100)
y = 1/(1 + np.exp(-x))
plt.plot(x,y)
```

```
[<matplotlib.lines.Line2D at 0x1f9224a9900>]
```



NUMPY TRICKS

np.sort() function:

- Return a sorted copy of an array.
  ~ column-wise represents axis=0
  ~ row-wise represents axis=1
- **Syntax:** np.argsort(arr, axis=-1, kind, order(optional))
  ~ axis: default is -1 (the last axis)
  ~ kind: Sorting algorithm. The default is 'quicksort'

```
import numpy as np
a = np.random.randint(1,20,size=15)
print(a)
b = np.random.randint(1,20,20).reshape(5,4)
print(b)
```

```
[ 5  7 14 15  6  7  9  5 11 19 13 15 16 15  7]
[[14 17  2 13]
 [10 14  5 11]
 [ 2 11 19  6]
 [14 19  7  7]
 [10  1 17  1]]
```

```
# ascending order(default)
asc = np.sort(a)
print("a:",asc)
# desending order
des = np.sort(a)[::-1]
print("d:",des)
```

```
a: [ 5  5  6  7  7  7  9 11 13 14 15 15 15 16 19]
d: [19 16 15 15 15 14 13 11  9  7  7  7  6  5  5]
```

```
c = np.sort(b, axis=0) # column-wise sort
r = np.sort(b, axis=-1) # row-wise sort
print('column-wise sort :')
print(c,'\n')
print('row-wise sort :')
print(r)
```

```
column-wise sort :
[[ 2  1  2  1]
 [10 11  5  6]
 [10 14  7  7]
 [14 17 17 11]
 [14 19 19 13]]

row-wise sort :
[[ 2 13 14 17]
 [ 5 10 11 14]
 [ 2  6 11 19]
 [ 7  7 14 19]
 [ 1  1 10 17]]
```

np.append() function:

- Appends values along the mentioned axis at the end of the array.
- **Syntax:** np.append(arr, values, axis)
  ~ values: [array_like] values to be added in the arr

```
import numpy as np
arr = np.random.randint(1,20,size=15)
print(arr)
arr1 = np.random.randint(1,20,20).reshape(5,4)
print(arr1,'\n')
# Append elements in existing 1D array
append = np.append(arr,200)
print(append)
# Append column in 2D array
app = np.append(arr1,np.ones((arr1.shape[0],1)),axis=1)
print(app)
```

```
[ 9 11  2  9  6 19 10  6  4  9 11 12  1  8  9]
[[10  3 17 19]
 [ 8 12 11 13]
 [18  7  9  6]
 [ 8 15 12 19]
 [ 1  9  9  7]]

[  9  11   2   9   6  19  10   6   4   9  11  12   1   8
9 200]
[[10.  3. 17. 19.  1.]
 [ 8. 12. 11. 13.  1.]
 [18.  7.  9.  6.  1.]
 [ 8. 15. 12. 19.  1.]
 [ 1.  9.  9.  7.  1.]]
```

np.concatenate() function:

- Return a sequence of concatenate arrays along an existing axis.
- It is replacement of hstack() & vstack() function.
  ~ column-wise represents axis=0
  ~ row-wise represents axis=1

- **Syntax:** np.cancatenate([arr1, arr2..], axis)

```
import numpy as np
c = np.arange(6).reshape(2,3)
d = np.arange(6,12).reshape(2,3)
print(c)
print(d)
```

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

```
# default concatenate row wise
cc = np.concatenate((c,d),axis=0)
print(cc)
```

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

```
# concatenate column wise
cc = np.concatenate((c,d),axis=1)
print(cc)
```

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

np.unique() function:

- Returns the sorted unique elements of an array.
- For example, consider a scenario where a single student registers for multiple courses. In this case, we aim to identify the unique users who have purchased the courses to ensure their usefulness.
- **Syntax:** np.unique(arr, axis)

```
import numpy as np
uni = np.array([1,2,3,3,4,4,5,5,6,6])
print("elements :",uni)
print('Unique Values :',np.unique(uni))
```

```
elements : [1 2 3 3 4 4 5 5 6 6]
Unique Values : [1 2 3 4 5 6]
```

np.expand_dims() function:

- Return an expanded the shape of an array.
- It is useful in 1D to 2D conversion or 3d to 4D conversion
- This function use in ML prediction and In DL to create batches of images.
- **Syntax:** np.expand_dims(arr, axis)

```
import numpy as np
arr = np.array([1,4,5,6,6])
print("1D array :",arr)
print('1D array shape', arr.shape)
# Expand the dimension of 1D array with column-wise
r = np.expand_dims(arr,axis=0)
print('2D :',r)
print('2D array shape:',r.shape)
```

```
1D array : [1 4 5 6 6]
1D array shape (5,)
2D : [[1 4 5 6 6]]
2D array shape: (1, 5)
```

```
# axis 1 is row-wise
print(np.expand_dims(arr,axis=1))
print('2D array shape:',a.shape)
```

```
[[1]
 [4]
 [5]
 [6]
 [6]]
2D array shape: (1, 5)
```

np.argmax() function:

- Returns the indices of the maximum values along an axis.
- Syntax: np.argmax(arr, axis)

np.argmin() function:

- Returns the indices of the minimum values along an axis.
- **Syntax**: np.argmin(arr, axis)

- Most of the time, both functions are used on 1D arrays in data science.

~ column-wise represents axis=0

~ row-wise represents axis=1

Note:
If same element occurrence in array while performing the functions, then prefer first occurrence element index.

```
import numpy as np
arr = np.array([1,2,3,4,4,20,5,5,0,6])
print("array :",arr)
```

```
array : [ 1  2  3  4  4 20  5  5  0  6]
```

```
# 1D array
print('max element index :',np.argmax(arr))
# 2D array
arr1 = np.arange(6,12).reshape(2,3)
print(arr1)
# column-wise (each column max index)
print('column max index:',np.argmax(arr1, axis=0))
# row-wise (each row max index)
print('row max index :',np.argmax(arr1, axis=1))
```

```
max element index : 5
[[ 6  7  8]
 [ 9 10 11]]
column max index: [1 1 1]
row max index : [2 2]
```

```
# 1D array
print('min element index :',np.argmin(arr))
# 2D array
arr1 = np.arange(6,12).reshape(2,3)
print(arr1)
# column-wise (each column min index)
print('column mix index:',np.argmin(arr1, axis=0))
# row-wise (each row min index)
print('row min index :',np.argmin(arr1, axis=1))
```

```
min element index : 8
[[ 6  7  8]
 [ 9 10 11]]
column mix index: [0 0 0]
row min index : [0 0]
```

np.where() function:

- Returns the indices of elements in an input array where the given condition is satisfied.
- **Syntax:** np.where(condition)

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

# Ex.Find all indices with value greater than 4
print(np.where(arr>4))

# Ex.Replace all values less 5 with 0
print(np.where(arr<5, 0, arr))
```

```
array : [1 2 3 3 4 4 5 5 6 6]
(array([6, 7, 8, 9], dtype=int64),)
[0 0 0 0 0 0 5 5 6 6]
```

np.percentile() function:

- Return compute the nth percentile of the given data (array elements) along the specified axis.
- Used in five summery to calculate interquartile.
- **Syntax:** np.percentile(arr, q, axis)
  ~ q: percentages to compute, value must be between 0 to 100 inclusive.

```
p = np.arange(1,101)
# 0th percentile (minimum)
print(np.percentile(p, 0))
# 50th percentile (median)
print(np.percentile(p, 50))
# 100th percentile (maximum)
print(np.percentile(p, 100))
```

```
1.0
50.5
100.0
```

np.cumsum() function:

- Return the cumulative sum of the elements along a given axis.
- **Syntax:** np.cumsum(arr, axis)

np.cumprod() function:

- Return the cumulative product of elements along a given axis.
- **Syntax:** np.cumprod(arr, axis)

~ column-wise represents axis=0
~ row-wise represents axis=1

```
import numpy as np
arr = np.array([1,2,3,4,5])
print("array :",arr)
print('cumulative sum :',np.cumsum(arr))
arr1 = np.arange(1,7).reshape(2,3)
print(arr1)
print('cumulative sum :',np.cumsum(arr1))
# column-wise
print(np.cumsum(arr1,axis=0))
# row-wise
print(np.cumsum(arr1,axis=1))
```

```
array : [1 2 3 4 5]
cumulative sum : [ 1  3  6 10 15]
[[1 2 3]
 [4 5 6]]
cumulative sum : [ 1  3  6 10 15 21]
[[1 2 3]
 [5 7 9]]
[[ 1  3  6]
 [ 4  9 15]]
```

```
import numpy as np
arr = np.array([1,2,3,4,5])
print('cumulative prod :',np.cumprod(arr))
arr1 = np.array([[1, 2, 3], [4, 5, 6]])
print(np.cumprod(arr1))
# column-wise:cumulative product for each column
print(np.cumprod(arr1,axis=0))
# row-wise:cumulative product for each row
print(np.cumprod(arr1,axis=1))
```

```
cumulative prod : [  1   2   6  24 120]
[  1   2   6  24 120 720]
[[ 1  2  3]
 [ 4 10 18]]
[[  1   2   6]
 [  4  20 120]]
```

np.corrcoef() function:

- Return Pearson product-moment correlation coefficients.
- Pearson's r, which is a measure of the linear correlation between two variables.
- Correlation coefficient range generally in between 1 to -1.
- If coefficient is 0 means no change.
  ~ Coefficient is 1 means both are positively correlated: means when experience increases then its salary also increases.
  ~ Coefficient is -1 means both are negatively correlated: means if experience is less than its salary also less.
- **Syntax:** np.corrcoef(arr1, arr2)

```
import numpy as np
salary = np.array([20000,40000,25000,35000,60000])
experience = np.array([1,3,2,4,2])
print(np.corrcoef(salary,experience))
```

```
[[1.         0.25344572]
 [0.25344572 1.        ]]
```

np.histogram() function:

- Compute the histogram of a dataset.
- Useful in statistics.
- **Syntax:** np.histogram(arr, bins=10, range).
  ~ bin: range of values of grouped together
  ~ range: lower & upper range of bins

```
import numpy as np
data = np.array([1,2,2,3,0,3,3,4,0,4,5,6,7,6,7])
bins = [0,1,2,3,4,5,6,7]
print(np.histogram(data,bins))
```

```
(array([2, 1, 2, 3, 2, 1, 4], dtype=int64), array
([0, 1, 2, 3, 4, 5, 6, 7]))
```

```
# change in bins range
import numpy as np
arr = np.array([11,53,28,50,38,30,68,9,78,2,21])
print(arr)
bins=[0,40,80]
print(np.histogram(arr,bins))
```

```
[11 53 28 50 38 30 68  9 78  2 21]
(array([7, 4], dtype=int64), array([ 0, 40, 80]))
```

np.flip() function:

- reverses the order of array elements along the specified axis, preserving the shape of the array.
- **Syntax:** np.flip(arr, axis)

```
import numpy as np
# 1D array
arr = np.array([1,2,3,4,5])
print("Before flip :",arr)
print('After flip :',np.flip(arr))
```

```
Before flip : [1 2 3 4 5]
After flip : [5 4 3 2 1]
```

```
import numpy as np
arr1 = np.arange(0,4).reshape(2,2)
print(arr1)
```

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

```
# 2D array
print(np.flip(arr1))
print(np.flip(arr1,axis=0)) # column-wise
print(np.flip(arr1,axis=1)) # row-wise
```

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

np.isin() function:

- Used to determine whether each element in an input array is contained in a second array.
- It returns a Boolean array of the same shape as the input, where each element is True if the corresponding element in the input is found in the second array, and False otherwise.
- Use in panda's library.
- **Syntax:** np.isin(arr, test_arr)

```
import numpy as np
arr = np.array([2,6,5,2,100,3,4,5])
test_arr = [20,3,40,5,0,100]
print("array :",arr)
print(np.isin(arr,test_arr))
print('elements :',arr[np.isin(arr,test_arr)])
```

```
array : [  2   6   5   2 100   3   4   5]
[False False  True False  True  True False  True]
elements : [  5 100   3   5]
```

np.put() function:

- Replaces specific elements of an array with given values.
- Array indexed works on flattened array.
- **Syntax:** np.put(arr, [index_position], [values])

```
import numpy as np
ar = np.array([1,2,3,4,5])
print("Before array :",ar)
np.put(ar,[0,2,4],[90,50,33])
print('After putting values:',ar)
```

```
Before array : [1 2 3 4 5]
After putting values: [90  2 50  4 33]
```

NOTE: Permanent changes in array.

np.delete() function:

- Returns a new array with the deletion of sub-arrays along with the mentioned axis.
- **Syntax:** np.delete(arr, [index_position], axis)

```
import numpy as np
arr = np.array([1,2,3,4,5,6])
print("array :",arr)
print('Del specific item:',np.delete(arr,0))
print('Del multiple items:',np.delete(arr,[4,5]))
```

```
array : [1 2 3 4 5 6]
Del specific item: [2 3 4 5 6]
Del multiple items: [1 2 3 4]
```

np.clip() function:

- Used to Clip (limit) the values in an array.
- Useful in certain scenarios of machine learning and deep learning.
- **Syntax:** np.clip(arr, a_min, a_min)

```
import numpy as np
arr = np.array([6,27,6,6,84,35,57,59,57,38,51,46])
print('array:',arr)
print(np.clip(arr, a_min=20, a_max=60))
```

```
array: [ 6 27  6  6 84 35 57 59 57 38 51 46]
[20 27 20 20 60 35 57 59 57 38 51 46]
```

SET FUNCTIONS IN NUMPY

Here we discuss some useful set function to perform set operations.

```
# Creating the two arrays
import numpy as np
arr1 = np.array([1,2,3,4,5])
arr2 = np.array([3,4,5,6,7])
print('arr1 :',arr1)
print('arr2 :',arr2)
```

```
arr1 : [1 2 3 4 5]
arr2 : [3 4 5 6 7]
```

np.union1d() function:

- Return the unique, sorted array of values that are in either of the two input arrays.
- **Syntax:** np.union1d(arr1, arr2)

```
# np.union1d()
print(np.union1d(arr1,arr2))
```

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

np.intersect1d() function:

- Return the sorted, unique values that are in both of the input arrays.
- **Syntax:** np.intersect(arr1, arr2)

```
# np.intersect1d()
print(np.intersect1d(arr1,arr2))
```

```
[3 4 5]
```

np.setdiff1d() function:

- Return the unique values in ar1 that are not in ar2.
- **Syntax:** np.setdiff1d(arr1, arr2)

```
# np.setdiff1d()
print(np.setdiff1d(arr1,arr2)) # for arr1
print(np.setdiff1d(arr1,arr2)) # for arr2
```

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

np.setxor1d() function:

- Return the sorted, unique values that are in only one (not both) of the input arrays.
- **Syntax:** np.setxor1d(arr1, arr2)

```
# np.setxor1d()
print(np.setxor1d(arr1,arr2))
```

```
[1 2 6 7]
```

np.in1d() function:

- Returns a boolean array the same length as ar1 that is True where an element of ar1 is in ar2 and False otherwise.
- Syntax: np.in1d(arr1, arr2)

```
# np.in1d()
print(np.in1d(arr1,3))
print(np.in1d(arr1,[1,3,4]))
```

```
[False False  True False False]
[ True False  True  True False]
```

EXTRA NUMPY FUNCTION THAT ARE USE IN TASK

np.tile() function:

- Repeating an array by repeating an input array multiple times along specified dimension.
- Useful when you want to create larger arrays by tiling or repeating smaller arrays.
- **Syntax:** np.tile(arr, reps)
  ~ arr: input array that you want to repeat
  ~ reps: a tuple specifying the number of times you want to repeat

```
import numpy as np
arr1 = np.array([1, 2])
arr2 = np.array([[1, 2], [3, 4]])
print('cols repeats 3 times:',np.tile(arr1, 3))
print('Repeats rows and cols by using tuple:')
# 2 is row & 3 is cols
print(np.tile(arr2, (2, 3)))
```

```
cols repeats 3 times: [1 2 1 2 1 2]
Repeats rows and cols by using tuple:
[[1 2 1 2 1 2]
 [3 4 3 4 3 4]
 [1 2 1 2 1 2]
 [3 4 3 4 3 4]]
```

np.nan_to_num() function:

- Replace NaN values with a specified value, which in this case would be the mode of the non-NaN values.
- Missing value concept
- **Syntax:** np.nan_to_num(arr, arr2)
  ~ arr: input array
  ~ nan: a default is 0 or we can replace nan value

```
import numpy as np
arr = np.array([5, np.nan, 1, np.nan])
print('Before :',arr)
# default nan value is 0
print('After :',np.nan_to_num(arr, nan=0))
```

```
Before : [ 5. nan  1. nan]
After : [5. 0. 1. 0.]
```

np.broadcast_to() function:

- It allows you to create a new array with a specified shape by broadcasting the original data to that shape.
- Useful when you want to perform element-wise operations on arrays with different shapes, but compatible dimensions.
- **Syntax:** np.broadcast_to(arr, shape)
  ~ arr: array to broadcast
  ~ shape: shape of the desired array.

```
import numpy as np
arr = np.array([1,2,3])
print('Original array :', arr)
print('Broadcated array:')
print(np.broadcast_to(arr, (3,3)))
```

```
Original array : [1 2 3]
Broadcated array:
[[1 2 3]
 [1 2 3]
 [1 2 3]]
```

np.broadcast() function:

- Produce an object that mimics broadcasting.
- "Mimics broadcasting" means that NumPy makes it appear as if the arrays have been expanded to a common shape, allowing you to perform element-wise operations without manually duplicating data.
- **Syntax:** np.broadcast(arr1, arr2, ...)

```
import numpy as np
a = np.array([[1, 2], [3, 4]])
b = np.array([10, 20])
result = a + b  # Broadcasting happens here
print(result)
```

```
[[11 22]
 [13 24]]
```

np.empty() function:

- Return a new array of given shape and type, with random values
- **Syntax:** np.empty(shape, dtype=float)

```
import numpy as np
b = np.empty(2, dtype=int)
print("Matrix b : \n", b)
a = np.empty([2, 2])
print("\nMatrix a : \n", a)
c = np.empty([3, 3])
print("\nMatrix c : \n", c)
```

```
Matrix b :
 [10 20]

Matrix a :
 [[0.00000000e+000 1.15694187e-311]
 [1.15694187e-311 1.15694187e-311]]

Matrix c :
 [[0.00000000e+000 0.00000000e+000 0.00000000e+000]
 [0.00000000e+000 0.00000000e+000 7.74694933e-321]
 [1.15648771e-311 2.00485144e-307 0.00000000e+000]]
```

NOTE: Empty, unlike zeros, does not set the array values to zero, and may therefore be marginally faster.

np.any() function:

- Tests whether any elements in a given array or along a specified axis evaluate to True.
- It returns a single Boolean value or an array of Boolean values, depending on the input.
- **Syntax:** np.any(arr, axis, keepdims)

```
import numpy as np
# if any element in the entire array is True
arr1 = np.array([False, False, False])
print('arr1:',np.any(arr1))
# if any element along a specific axis is True
arr2 = np.array([[False, True, False],
                 [False, False, False]])
print('arr2:',np.any(arr2, axis=0))
# Using keepdims
arr3 = np.array([[False, True, False],
                 [False, False, False]])
print('arr3:',np.any(arr3, axis=1, keepdims=True))
```

```
arr1: False
arr2: [False  True False]
arr3: [[ True]
 [False]]
```

np.argsort() function:

- Returns the indices that would sort an array and also sorted along with axis.
- **Syntax:** np.argsort(arr, axis=-1, kind , order)
  ~ axis: default is -1 (the last axis)
  ~ kind: sorting algorithm. The default is 'quicksort'

```
import numpy as np
arr = np.array([3, 1, 4, 1, 5, 9, 2, 6])
print(np.argsort(arr))
```

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

```
import numpy as np
arr = np.array([[3, 1, 4],[1, 5, 9],[2, 6, 5]])
print(arr)
print("Sorted indices along axis 0:")
print(np.argsort(arr, axis=0))
print("Sorted indices along axis 1:")
print(np.argsort(arr, axis=1))
```

```
[[3 1 4]
 [1 5 9]
 [2 6 5]]
Sorted indices along axis 0:
[[1 0 0]
 [2 1 2]
 [0 2 1]]
Sorted indices along axis 1:
[[1 0 2]
 [0 1 2]
 [0 2 1]]
```

EXTRA NUMPY METHODS/FUNCTIONS

np.random.seed() function:

- Save the state of a random function.
- The value in the NumPy random seed saves the state of randomness.
- For i.e., if we call the seed function using value 1 multiple times, the computer displays the same random numbers.
- Syntax: np.random.seed(seed_value)

```
import numpy as np
# wihtout seed_value
np.random.seed()
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[22, 55, 79, 63],
       [33, 81, 45, 69],
       [10, 38, 74, 46]])
```

```
# save the random state at position 0
np.random.seed(0)
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[45, 48, 65, 68],
       [68, 10, 84, 22],
       [37, 88, 71, 89]])
```

```
# save the random state at position 1
np.random.seed(1)
np.random.randint(1,100,12).reshape(3,4)
```

```
array([[38, 13, 73, 10],
       [76,  6, 80, 65],
       [17,  2, 77, 72]])
```

np.random.shuffle() function:

- We can get the random positioning of different integer values in the numpy array or we can say that all the values in an array will be shuffled randomly.
- **Syntax:** np.random.shuffle(x)
  ~ x: It is a sequence you want to shuffle such as list.

```
import numpy as np
a = np.array([12,41,33,67,89,100])
print(a)
```

```
[ 12  41  33  67  89 100]
```

```
# after applying the np.random.shuffle() function
np.random.shuffle(a)
```

```
a
```

```
array([ 67, 100,  41,  12,  33,  89])
```

NOTE: Permanent changes into the array or any sequence.

np.random.choice() function:

- We can get the random samples of one dimensional array and return the random samples of numpy array.
- **Syntax:** np.random.choice(a, size, replace=True)
  ~ a : 1D array of numpy having random samples.
  ~ size : output shape of random samples of numpy array.
  ~ replace: whether the sample is with or without replacement.

```
import numpy as np
a = np.array([12,41,33,67,89,100])
np.random.choice(a,5)
```

```
array([12, 41, 67, 89, 33])
```

```
np.random.choice(a,4)
```

```
array([100, 100,  12,  33])
```

```
# to stop repeated number in choice
np.random.choice(a,4,replace=False)
```

```
array([ 33, 100,  67,  41])
```

np.swapaxes() function:

- Interchange two axes of an array.
- **Syntax:** np.swapaxes(arr, axis1, axis2)
  ~ arr : input array.
  ~ axis1 : first axis.
  ~ axis2 : second axis.

```
#Example
x = np.array([[1,2,3],[4,5,6]])
print(x)
print(x.shape)
print("Swapped")
x_swapped = np.swapaxes(x,0,1)
print(x_swapped)
print(x_swapped.shape)
```

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

NOTE: It is not same as reshaping

np.random.uniform() function:
- Draw samples from a uniform distribution in rangge [low - high); high not included.
- **Syntax:** np.random.uniform(low, high, size=None)
  ~ low: lower bound of sample; default value is 0
  ~ high: upper bound of sample; defalut value is 1.0
  ~ size: shape of the desired sample. If the given shape is, e.g., (m, n, k), then m * n * k samples are drawn.
- Return the random samples as numpy array.
- Whenever we need to test our model on uniform data and we might not get truly uniform data in real scenario, we can use this function to randomly generate data for us.

np.repeat() function:
- Repeat elements of an array. `repeats` parameter says no of time to repeat.
- **Syntax:** np.repeat(a, repeats, axis)
  ~ a: Input array.
  ~ repeats: the number of repetitions for each element. repeats is broadcasted to fit the shape of the given axis.
  ~ axis : 0 or 1

np.count_nonzero() function:
- This function counts the number of non-zero values in the array. https://numpy.org/doc/stable/reference/generated/numpy.count_nonzero.html
- Returns number of non-zero values in the array along a given axis. Otherwise, the total number of non-zero values in the array is returned.
- **Syntax:** np.count_nonzero(arr, axis, keepsdims)
  ~ arr : the array for which to count non-zeros.
  ~ axis : axis or tuple of axes along which to count non-zeros. Default is None, meaning that non-zeros will be counted along a flattened version of arr.
  ~ keepdims : If this is set to True, the axes that are counted are left in the result as dimensions with size one.

np.allclose() function:
- Returns True if two arrays are element-wise equal within a tolerance. The tolerance values are positive, typically very small numbers.
- The relative difference `(rtol * abs(b))` and the absolute difference `atol` are added together to compare against the `absolute difference` between `a` and `b`.
- If the following equation is element-wise True, then `allclose` returns `True`.
- absolute(a - b) <= (atol + rtol * absolute(b))
- **Syntax:** numpy.allclose(arr1, arr2, rtol, atol, equal_nan=False)
  ~ arr1 : input 1st array.
  ~ arr2 : input 2nd array.
  ~ rtol : the relative tolerance parameter.

np.unravel_index() function:
- Converts a flat index or array of flat indices into a tuple of coordinate arrays.
- **Syntax:** np.unravel_index(indices, shape)

np.flatten() function:
- Return 1D array
- **Syntax:** np.flatten(arr)

np.around() function:
Syntax: np.around(arr1, arr2)

PANDAS

WHAT IS PANDAS?

- Pandas is a fast, powerful, flexible, and easy to use open-source data analysis and manipulation tool, built on top of the Python programming language.
- about pandas: https://pandas.pydata.org/about/index.html

USE OF PANDAS

- **Data Cleaning and Preparation:**
  Pandas provides tools for handling missing data, data alignment, transformation, and data normalization.
- **Time Series Analysis:**
  Pandas offers specialized functionality for working with time series data.
  Pandas integrate well with visualization libraries like Matplotlib and Seaborn which are used for Data Visualization.

BASIC DATA STRUCTURE IN PANDAS

- Pandas provide two types of classes for handling data:
  ~ Series
  ~ DataFrame

IMPORTING PANDAS

There is different way to import pandas library:

1.Most common way is importing statement with alias:

~ import pandas as pd

2.Importing all the function and class:

~ from pandas import *

3.Importing the specific function and class library:

~ from pandas import DataFrame, read_csv

PANDAS SERIES

- A Pandas Series is like a column in a table.
- It is a one-dimenational labelled array capable of holding of any datatype.
- Integers, string, floating point numbers, python object etc.
- Each value in a pandas series is associated with the index.
- The default index value of it is from 0 to number – 1, or you can specify your own index values.

NOTE:

- 'Series' is a class provided by the panda's library.
- When you create a 'Series' in your code, means your creating an 'Series' object of that 'Series' class.
- **Syntax**: pd.Series(data, index, name)

Parameters:

data:	It can be array, list, dictionary, csv file or excel file
index:	Default, also create custom index
name:	Series name

CREATING A PANDAS SERIES

- Pandas series can be created from the lists, dictionary and from other scalar values etc.
- Series can be created in different ways, here are some ways to create a series.

CREATE AN EMPTY SERIES

Code:

```
import pandas as pd
empty_series = pd.Series()
print(empty_series)
```

Output:

```
Series([ ], dtype: float64)
```

CREATE A SERIES FROM LISTS

We first creating a list after that we can create series of list.

```
# String
```

Code:

```
import pandas as pd
import numpy as np
countries = [ 'India' , 'Nepal' , 'Srilanka' , 'Bhutan' ]
print(pd.Series(countries))
```

Output:

```
0        India
1        Nepal
2     Srilanka
3       Bhutan
dtype: object
```

 # Custom index and name

Code:

```
marks = [ 58, 93, 89, 60 ]
subjects = [ 'C++' , 'Python', 'R', 'Java' ]
print(pd.Series(marks, index=subjects, name='student'))
```

Output:

```
C++       58
Python    93
R          89
Java       60
Name: student, dtype: int64
```

CREATE A SERIES FROM DICTIONARY

We must first create a dictionary then we can perform series on dictionary.

Code:

```
import pandas as pd
marks = { 'maths' : 78,  'english' : 70, 'science' : 89 }
pd.Series(marks, name='student score')
```

Output:

```
maths     78
english   70
science   89
Name: student score, dtype: int64
```

PANDAS SERIES ATTRIBUTE

In Pandas, a Series object has several important attributes that is commonly used attributes of a Pandas Series include:

size attribute:

- Returns the number of elements in a Series, including any elements that might contain missing or NaN (Not-a-Number) values.

Code:

```
import pandas as pd
data = [ 10, 20, 30, None, 50 ]
print(pd.Series(data).size)
data1 = [ 10, 20, 30, 50 ]
print(pd.Series(data1).size)
```

Output:

```
5
4
```

dtype attribute:

- Returns the data type of the elements in the Series.
- Used to check the data type of the data within the Series.

Code:

```
import pandas as pd
data = [ 10, 20, 30, 50 ]
print(pd.Series(data).dtype)
```

Output:

```
int64
```

name attribute:

- Assign a name to a series when creating it or later using the name attribute.
- The name is typically used in the context of DataFrames, where a Series can represent a column.

Code:

```
import pandas as pd
data = [ 10, 20, 30, 50 ]
print(pd.Series(data, name='my_data').name)
```

Output:

```
my_data
```

is_unique attribute:

- Returns a boolean value indicating whether all the values in the Series are unique (no duplicates) or not.

Code:

```
import pandas as pd
data = [ 10, 20, 30, 40, 50 ]
print(pd.Series(data).is_unique)
data1 = [ 10, 20, 30, 40, 50 ]
print(pd.Series(data1).is_unique)
```

Output:

```
True
False
```

values attribute:

• Returns the data in the Series as a NumPy array.

Code:
```
import pandas as pd
marks = { 'maths' : 78,  'english' : 70, 'science' : 89 }
print(pd.Series(marks).values)
```
Output:
```
[ 78 70 89 ]
```

Series using read_csv() function:

Parameters:

squeeze=True:

• Attribute specifies that the result should be squeezed into a series if it has only one column.

• Here we intentionally use this attribute to avoid the dataframe to understand the about series.

Code:
```
import pandas as pd
subs = pd.read_csv('subs.csv', squeeze=True )
print(subs)
print(type(subs))
```
Output:
```
0        48
1        57
         ...
363    144
364    172
Name: Subscribers gained, Length: 365, dtype: int64
<class 'pandas.core.series.Series'>
```

index_col:

• Used to specify which column in the CSV file should be used as the index for the resulting DataFrame/Series.

• The index is a way to uniquely identify each row in the DataFrame/Series.

• By default, if you don't specify the index_col parameter, Pandas will create a default integer index starting from 0.

Code:
```
import pandas as pd
vk = pd.read_csv('kohli_ipl.csv', index_col= 'match_no',
                                squeeze=True )
print(vk)
print(type(vk))
```
Output:
```
match_no
1              1
2             23
              ..
214           25
215            7
Name: runs, Length: 215, dtype: int64
<class 'pandas.core.series.Series'>
```

Code:
```
import pandas as pd
m = pd.read_csv('bollywood.csv',index_col='movie',squeeze=True)
print(m)
print(type(m))
```
Output:
```
movie
Uri: The Surgical Strike          Vicky Kaushal
Battalion 609                        Vicky Ahuja
                                              ...
Company (film)                       Ajay Devgn
Awara Paagal Deewana          Akshay Kumar
Name: lead, Length: 1500, dtype: object
<class 'pandas.core.series.Series'>
```

NOTE: Above dataset used to perform series methods

SERIES METHODS

These methods are most used in pandas series.

head() method:

• Used to display the first few rows (default is 5) of a DataFrame or Series.

• If we provide negative number as parameter then return all rows.

Code:
```
import pandas as pd
print(vk.head()) # default is 5
print(vk.head(2))
```
Output:
```
match_no
1        1
2       23
3       13
4       12
5        1
Name: runs, dtype: int64
match_no
1        1
2       23
Name: runs, dtype: int64
```

tail() method:

• Used to display the last few rows (default is 5) of a DataFrame or **Series.**

Code:
```
import pandas as pd
print(vk.tail())
print(vk.tail(2))
```
Output:
```
match_no
211        0
212       20
213       73
214       25
215        7
Name: runs, dtype: int64
match_no
214       25
215        7
Name: runs, dtype: int64
```

sample() method:

• To randomly select a specified number of rows (default is 1) or elements from a dataframe or series.

• Useful to obtain a random sample from your data for data exploration, analysis, or testing.

• Ex. if you have biased in your datasets then we can use this method.

Code:
```
import pandas as pd
print(vk.sample()) # default 1
print(vk.sample(3)) # three random row from dataset
```
Output:
```
match_no
118      33
Name: runs, dtype: int64
match_no
202       5
150       8
110      82
Name: runs, dtype: int64
```

value_counts() method:

• To count the frequency of values that occur multiple times in a series.

Code:
```
import pandas as pd
print(m.value_counts())
```
Output:
```
Akshay Kumar             48
Amitabh Bachchan         45
                         ..
Akanksha Puri             1
Edwin Fernandes           1
Name: lead, Length: 566, dtype: int64
```

sort_values() method:

• Used to sort the values within a pandas series.

• By default, it sorts the values in ascending order, but you can specify the sorting order using the ascending parameter.

• **Syntax:** series.sort_values(ascending=True, inplace=False)

Parameters:

inplace:	when true then sort and replace sorted data with original data If false (default), it returns a new series with the sorted values while leaving the original series unchanged
ascending:	True (ascending) or False (descending)

Code:

```
import pandas as pd
print(vk.sort_values()) # default is ascending=True
```

Output:

```
match_no
135      0
8        0
        ..
126    109
128    113
Name: runs, Length: 215, dtype: int64
```

 # with inplace=True for permanent changes in series

Code:

```
print(vk.sort_values(inplace=True)
print(vk)
```

Output:

```
match_no
128    113
126    109
        ...
8        0
135      0
Name: runs, Length: 215, dtype: int64
```

METHOD CHAINING:

- It is practice of applying multiple operations or methods to a Series in a single line of code.
- This approach is both efficient and readable, making it easier to perform complex data manipulations and transformations.

Code:

```
import pandas as pd
print(vk.sort_values(ascending=False).head(1).values[0])
```

Output:

```
113
```

sort_index() method:

- It similar in concept to the sort_values() method, but in this method it s orts the index (row labels) of the Series.
- Both methods allow you to control the sorting order, either ascending o r descending, and both can be used with the inplace parameter to modif y the original Series.
- **Syntax:** series.sort_index(ascending=True, inplace=False)

Code:

```
import pandas as pd
print(vk.sort_index()) # default is ascending=True
```

Output:

```
match_no
1        1
2       23
        ...
214     25
215      7
Name: runs, Length: 215, dtype: int64
```

 # descending with inplace=True for permanent changes

```
print(vk.sort_index(ascending=False, inplace=True)
print(vk)
```

Output:

```
match_no
215      7
214     25
        ..
2       23
1        1
Name: runs, Length: 215, dtype: int64
```

SERIES MATHEMATICAL METHODS

Common statistical methods in Pandas Series for analyzing data:

count() method:

- Count the non-null elements in the Series.

Code:

```
print(vk.count())
print(subs.count())
```

Output:

```
215
365
```

sum() method:

- Used to calculate the sum of all the elements in a Series.

Code:

```
print(subs.count())
```

Output:

```
49510
```

product() method :

- Used to calculate the product of all elements in the Series.
- It multiplies all the values together and returns the result.

Code:

```
print(subs.count())
print(vk.subs)
```

Output:

```
215
```

mean() method:

- Calculates the mean (average) of the elements in a Series.

Code:

```
import pandas as pd
print('avg yt subs of channel:', subs.mean())
print('avg ipl runs of virat kohli:', vk.mean())
```

Output:

```
avg yt subs of channel: 135.64383561643837
avg ipl runs of virat kohli: 30.855813953488372
```

median() method:

- Calculates the median of the elements in a Series, which is the middle value when the data is sorted.
- It is a measure of central tendency.

Code:

```
import pandas as pd
print(subs.median())
print(vk.median())
```

Output:

```
123.0
24.0
```

mode() method:

- Returns the mode(s) of the elements in a Series, which is the most frequently occurring value(s).

Code:

```
import pandas as pd
print('Most frequent lead actor :', m.mode())
```

Output:

```
Most frequent lead actor : 0    Akshay Kumar
Name: lead, dtype: object
```

std() method:

- Computes the standard deviation of the elements in a Series, which measures the spread or dispersion of the data.

Code:

```
import pandas as pd
print(vk.std())
```

Output:

```
26.22980132830278
```

var() method:

- Calculates the variance of the elements in a Series, which is the average of the squared differences from the mean.

Code:

```
import pandas as pd
print(subs.var())
print(vk.var())
```

Output:

```
3928.1585127201565
688.0024777222343
```

min() method:

- Returns the minimum value in a Series or DataFrame.

Code:

```
import pandas as pd
print('minimum subs: ', subs.min())
print('minimum runs: ', vk.min())
```

Output:

```
minimum subs: 33
minimum runs: 0
```

max() method:

- The max() method returns the maximum value in a Series or DataFrame.

Code:

```
import pandas as pd
print('maximum subs: ', subs.max())
print('maximum runs: ', vk.max())
```

Output:

```
maximum subs: 396
maximum runs: 113
```

describe() method:

- It is a convenient function to generate descriptive statistics of a numeric Series.
- It provides a summary of various statistical measures, giving you insights into the data's distribution and central tendency.
- It's provides the following statistics:

count:	number of non-null elements in the Series.
mean:	mean (average) of the Series.
std:	standard deviation, which measures the spread of the data.
min:	minimum value in the Series.
25%:	25th percentile (lower quartile).
50%:	median (50th percentile).
75%:	75th percentile (upper quartile).
max:	maximum value in the Series.

NOTE: the describe() works on numeric data

Code:

```
import pandas as pd
print(subs.var())
print(vk.var())
```

Output:

```
count    215.000000
mean      30.855814
std       26.229801
min        0.000000
25%        9.000000
50%       24.000000
75%       48.000000
max      113.000000
Name: runs, dtype: float64

count    365.000000
mean     135.643836
std       62.675023
min       33.000000
25%       88.000000
50%      123.000000
75%      177.000000
max      396.000000
Name: Subscribers gained, dtype: float64
```

SOME IMPORTANT SERIES METHODS

These are some common methods and functions available for working with Pandas Series in Python:

astype() method:

- This method is used to cast the data type of the elements in a Series to the specified data type (e.g., int, float, str).
- Useful to reduce the memory space
- **Syntax:** series.astype(dtype)

Code:

```
import pandas as pd
import sys
print('Original size of dataset:',sys.getsizeof(vk))
vk_size =vk.astype('int32')
print('Reduce size of dataset:',sys.getsizeof(vk_size))
```

Output:

```
Original size of dataset: 3456
Reduce size of dataset: 2596
```

between() method:

- Checks if each element in the Series falls within the specified range.
- Returns a boolean Series.
- **Syntax:** series.between(left, right, inclusive=True)

Code:

```
import pandas as pd
print(vk.between(95,110)) # return boolean values
print(vk[vk.between(95,110)]) # printing values
```

Output:

```
match_no
1      False
2      False
       ...
214    False
215    False
Name: runs, Length: 215, dtype: bool
match_no
82      99
120    100
123    108
126    109
164    100
Name: runs, dtype: int64
```

clip() method:

- Clips values in the Series to be within the specified lower and upper bounds.
- **Syntax:** series.clip(lower, upper)

Code:

```
import pandas as pd
print(subs.clip(100,160))
```

Output:

```
0      100
1      100
       ...
363    144
364    160
Name: Subscribers gained, Length: 365, dtype: int64
```

drop_duplicates() method:

- Removes duplicate values from the Series.
- **Syntax:** series.drop_duplicates(keep='first', inplace=False)

Code:

```
import pandas as pd
temp = pd.Series([1,1,3,3,3,5,5])
# default first
print(temp.drop_duplicates())
# deleting first occurance
print(temp.drop_duplicates(keep='last'))
```

Output:

```
0    1
2    3
5    5
dtype: int64
1    1
4    3
6    5
dtype: int64
```

duplicated() function:

- Used to identify and mark duplicate values in a Series (column) of a DataFrame.
- It returns a Boolean Series.
- **Syntax:** Series.duplicated()

Code:

```
import pandas as pd
temp = pd.Series([1,1,3,3,3,5,5])
print(temp.duplicated()) # True means duplicate
print('Duplicate value count:',temp.duplicated().sum())
```

Output:

```
0    False
1     True
2    False
3     True
4     True
5    False
6     True
dtype: bool
Duplicate value count: 4
```

isnull() method:

- Returns a boolean Series indicating whether each element is NaN (missing data).

• **Syntax:** series.isnull()

Code:
```
import pandas as pd
import numpy as np
temp = pd.Series([1,3,np.nan,np.nan,5,np.nan,7,np.nan])
print(temp.isnull()) # return boolean
print('Missing values:',temp.isnull().sum())
```
Output:
```
0    False
1    False
2     True
3     True
4    False
5     True
6    False
7     True
dtype: bool
Missing values: 4
```

dropna() method:
• Removes missing (NaN) values from the Series.
• **Syntax:** series.dropna(axis=0, inplace=False)

Code:
```
import pandas as pd
import numpy as np
temp = pd.Series([1,3,np.nan,np.nan,5,np.nan,7,np.nan])
print(temp.dropna())
```
Output:
```
0   1.0
1   3.0
4   5.0
6   7.0
dtype: float64
```

fillna() method:
• Fills missing (NaN) values in the Series with the specified value.
• **Syntax:** series.fillna(value)

Code:
```
import pandas as pd
import numpy as np
temp = pd.Series([1,3, np.nan,5,np.nan,7,np.nan])
print(temp.fillna())
```
Output:
```
0   1.0
1   3.0
2   0.0
3   5.0
4   0.0
5   7.0
6   0.0
dtype: float64
```

isin() method:
• Checks if each element in the Series is in the provided list of values.
• Returns a boolean Series.
• **Syntax:** series.isin(values)

Code:
```
import pandas as pd
print(vk.isin([49,99])) # return Boolean
print(vk[vk.isin([49,99])]) # printing values
```
Output:
```
match_no
1      False
2      False
       ...
214    False
215    False
Name: runs, Length: 215, dtype: bool
match_no
82     99
86     49
Name: runs, dtype: int64
```

apply() method:
• Applies a given function to each element in the Series and returns a new Series with the results.
• **Syntax:** series.apply(func)

Code:
```
import pandas as pd
print(m.apply(lambda x:x.split()[0].upper()).head(5))
```
Output:
```
movie
Uri: The Surgical Strike                   VICKY
Battalion 609                              VICKY
The Accidental Prime Minister (film)    ANUPAM
Why Cheat India                            EMRAAN
Evening Shadows                            MONA
Name: lead, dtype: object
```

EXTRA SERIES METHOD THAT IS USED IN TASK

to_numeric() method:
• Used to convert the values in a Series (or DataFrame column) to numeric data types.
• Useful when you have a Series containing strings or other non-numeric data, and you want to convert them to numeric types like integers or floating-point numbers.
• **Syntax:** pd.to_numeric(series, errors='coerce', downcast='integer')
Parameters:

errors:	'raise'	Raises an error if any value cannot be converted to a number.
	'coerce'	Replaces non-convertible values with NaN.
	'ignore'	Ignores non-convertible values

Code:
```
import pandas as pd
data = pd.Series(['1', '2', '3.14', 'hello', '5'])
numeric_data = pd.to_numeric(data, errors='coerce')
print(numeric_data)
```
Output:
```
0   1.00
1   2.00
2   3.14
3    NaN
4   5.00
dtype: float64
```

quantile() method:
• Return value at the given quantile.
• **Syntax:** pd.quantile(q=0.5, interpolation)
Parameters:

q:	float or array-like, default 0.5 (50% quantile)
interpolation:	{'linear', 'lower', 'higher', 'midpoint', 'nearest'}

Code:
```
import pandas as pd
data = pd.Series([1, 2, 3, 4, 5])
q = data.quantile()
print(q)
```
Output:
```
3.0
```

PANDAS DATAFRAMES

PANDAS DATAFRAME
• A Pandas DataFrame is a two-dimensional data structure, like a two-dimensional array, or a table with rows and columns.
• Used for data manipulation, analysis, and cleaning, featuring labelled rows and columns, support for various data types, and flexibility for adding, removing, and transforming data.
• It is widely used in data science and analysis tasks for handling structured data efficiently.
• **Syntax:** pd.DataFrame(data, index, columns, dtype, copy)
Parameters:

data:	ndarray, Iterables, dict, or DataFrame
index:	Index or array-like
columns:	Index or array-like
dtype:	dtype, default None
copy:	bool or None, default None

CREATING A DATAFRAME USING LISTS

Code:
```
import pandas as pd
import numpy as np
student_data = [ [100,95,14], [107,87,16], [89,78,12] ]
```

```
print(pd.DataFrame(student_data,
columns= [ 'iq', 'marks', 'package' ] ))
```

Output:

```
    iq  marks  package
0  100     95       14
1  107     87       16
2   89     78       12
```

CREATING A DATAFRAME USING DICTIONARY

Code:

```
import pandas as pd
import numpy as np
dictionary = {
                'iq' : [100,95,14],
                 'marks' : [107,87,16],
                'package' : [89,78,12]
              }
students = pd.DataFrame(dictionary)
print(students)
```

Output:

```
    iq  marks  package
0  100     95       14
1  107     87       16
2   89     78       12
```

CREATING A DATAFARAME USING read_csv() FUNCTION

Code:

```
import pandas as pd
# movie datasets
movies = pd.read_csv('movies.csv')
# ipl matches dataset
ipl = pd.read_csv('ipl.csv')
```

NOTE: Above datasets use for performing dataframe attributes and methods.

PANDAS DATAFRAME ATTRIBUTE

Accessing a DataFrame through its attributes allows us to get the intrinsic properties of the DataFrame.

shape attribute:

- Used to display the total number of rows and columns of a particular data frame.
- Returns a tuple representing the dimensionality of the DataFrame.
- For example, if we have 3 rows and 2 columns in a DataFrame then the shape will be (3,2).

Code:

```
import pandas as pd
print('Shape of the DataFrame:',movies.shape)
print('Shape of the DataFrame:',ipl.shape)
```

Output:

```
Shape of the DataFrame: (1629, 18)
Shape of the DataFrame: (950, 20)
```

dtypes attribute:

- Return datatype of each column present in a dataframe.

Code:

```
import pandas as pd
print(ipl.dtypes)   # or print(movies.dtypes)
```

Output:

```
ID                int64
City             object
Date             object
Season           object
MatchNumber      object
Team1            object
      ...
Team2Players     object
Umpire1          object
Umpire2          object
dtype: object
```

index attribute:

- Display the row labels of a the dataframe object.

Code:

```
import pandas as pd
print(movies.index)
print(ipl.index)
print(students.indexs)
```

Output:

```
RangeIndex(start=0, stop=1629, step=1)
RangeIndex(start=0, stop=950, step=1)
RangeIndex(start=0, stop=3, step=1)
```

columns attribute:

Fetch the label values for columns present in a dataframe.

Code:

```
import pandas as pd
print(ipl.columns)
print(students.columns)
```

Output:

```
Index(['ID','City','Date','Season','MatchNumber','Team1','Team2',
       'Venue', 'TossWinner', 'TossDecision', 'SuperOver',
       'WinningTeam', 'WonBy', 'Margin', 'method',
       'Player_of_Match', 'Team1Players', 'Team2Players',
       'Umpire1', 'Umpire2'],  dtype='object')
Index(['iq', 'marks', 'package'], dtype='object')
```

values attribute:

- Represent the values/data of dataframe in numpy array from.

Code:

```
import pandas as pd
# print(movies.values)
# print(ipl.values)
print(students.values)
```

Output:

```
[[100 107  89]
 [ 95  87  78]
 [ 14  16  12]]
```

PANDAS DATAFRAME METHODS

These methods are most used in Pandas DataFrame.

head() method:

- Used to display the first few rows (default is 5) of a DataFrame.

Code:

```
import pandas as pd
print(movies.head()) # default is 5
print(ipl.head(2))
```

Output:

No output taken

tail() method:

- Used to display the last few rows (default is 5) of a DataFrame.

Code:

```
import pandas as pd
print(movies.tail()) # default is 5
print(movies.tail(2))
```

Output:

No output taken

sample() method:

- To randomly select a specified number of rows or columns (default is 1) or elements from a dataframe.
- Useful to obtain a random sample from your data for data exploration, analysis, or testing.
- Ex. if we have biased in your datasets then we can use this method.

Code:

```
import pandas as pd
print(students.sample()) # default is 1
# specify the number for random sample
# print(ipl.sample(3))
```

Output:

```
    iq  marks  package
1  107     87       16
```

info() method:

- Used to get a concise summary of the dataframe.
- Prints information about a DataFrame including the index dtype and columns, non-null values and memory usage.
- To get a quick overview of the dataset we use the info() method.

Code:

```
import pandas as pd
# print(ipl.info())
# print(movies.info())
```

```
print(students.info())
```
Output:
```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 3 entries, 0 to 2
Data columns (total 3 columns):
 #   Column      Non-Null Count    Dtype
---  ------      --------------    -----
 0   iq          3 non-null        int64
 1   marks       3 non-null        int64
 2   package     3 non-null        int64
dtypes: int64(3)
memory usage: 204.0 bytes
```

PANDAS DATAFRAME MATHEMATICAL METHODS

Common statistical methods in Pandas DataFrame

- **index (0) represents axis=0**
- **column (1) represents axis=1**

sum() metnod:

- Return the sum of the values over the requested axis.
- **Syntax:** DataFrame.sum(axis, numeric_only ,skipna, **kwargs)

Parameters:

axis:	It represents index or column axis, '0' for index and '1' for the column. When the axis=0, method applied over the index axis and when the axis=1 method applied over the column axis.
skipna:	Bool (True or False). The default value is None.
numeric_only:	bool, default False Include only float, int, boolean columns. Not implemented for Series.
**kwargs:	Additional keyword arguments to be passed to the function.

Code:
```
import pandas as pd
print(students.sum()) # default (column-wise)
print(students.sum(axis=1)) # row-wise
```
Output:
```
iq        209
marks     210
package   179
dtype: int64
0   296
1   260
2    42
dtype: int64
```

max() method:

- Used to get the maximum of the values over the requested axis. It returns Series and if the level is specified, it returns the DataFrame.
- **Syntax:** DataFrame.max(axis, skipna, numeric_only **kwargs)

Code:
```
import pandas as pd
print(students.max())
```
Output:
```
iq    100
%     107
lpa    89
dtype: int64
```

min() method:

- Used to get the minimum of the values over the requested axis.
- It returns Series and if the level is specified, it returns the DataFrame.
- **Syntax:** DataFrame.min(axis, skipna, numeric_only, **kwargs)

Code:
```
import pandas as pd
print(students.min())
```
Output:
```
iq       0
%        0
lpa      0
dtype: int64
```

mean() method:

- Used to get the mean of the values over the requested axis.
- It returns Series and if the level is specified, it returns the DataFrame.
- **Syntax:** DataFrame.mean(axis, skipna, numeric_only, **kwargs)

Code:
```
import pandas as pd
print(students.mean())
```
Output:
```
iq       41.8
%        42.0
lpa      35.8
dtype: int64
```

median() method:

- Used to get the median of the values over the requested axis.
- It returns Series and if the level is specified, it returns the DataFrame.
- **Syntax:** DataFrame.median(axis, skipna, **kwargs)

Code:
```
import pandas as pd
print(students.mean())
```
Output:
```
iq        14.0
%         16.0
lpa       12.0
dtype: int64
```

mode() method:

- we can get each element mode along the specified axis.
- When this method applied to the DataFrame, it returns the DataFrame which consists of the modes of each column or row.
- **Syntax:** DataFrame.mode(axis=0, drpna=True)

Parameters:

dropna:	It represents the bool, and the default is True. It does not consider the null values.

Code:
```
import pandas as pd
print(students.mode())
```
Output:
```
   iq   %   lpa
0   0   0     0
```

std() method:

- Return sample standard deviation over requested axis. By default the standard deviations are normalized by N-1.
- It is a measure that is used to quantify the amount of variation or dispersion of a set of data values.
- **Syntax:** DataFrame.std(axis, skipna, level, ddof=1, numeric_only, **kwargs)

Parameters:

dropna:	It represents the bool, and the default is True. It does not consider the null values.
ddof:	int, default 1 Delta Degrees of Freedom. The divisor used in calculations is N - ddof, where N represents the number of elements.

Code:
```
import pandas as pd
print(students.std())
```
Output:
```
iq       51.197656
%        51.122402
lpa      43.990908
dtype: float64
```

var() function:

- Returns the unbiased variance over the specified axis.
- **Syntax:** DataFrame.var(axis, skipna, level, ddof=1)

Code:
```
import pandas as pd
print(students.var())
```
Output:
```
iq        2621.2
%         2613.5
lpa       1935.2
dtype: float64
```

sort_values() function: (series or dataframe)

- sort_values() is used to sort the values within a Pandas Series.
- By default, it sorts the values in ascending order, but you can specify the sorting order using the ascending parameter, where ascending=True sorts in ascending order, and ascending=False sorts in descending order

- **Syntax:** series.sort_values(axis=0,ascending=True,inplace=False, kind='quicksort', na_position='last')

Parameters:

axis :	axis to direct sorting
ascending:	If True, sort values in ascending order, otherwise descending.
inplace:	If True, perform operation in-place.
kind:	choice of sorting algorithm.
na_position:	argument 'first' puts NaNs at the beginning, 'last' puts NaNs at the end.

Code:

```
import pandas as pd
movies = pd.read_csv('movies.csv')
movies.sort_values()
# Sorting based on column
movies.sort_values(by='title_x', ascending=False)
# Sorting with multiple columns movies.columns
movies.sort_values(by=['year_of_release', 'title_x' ],
                   ascending=[False, False])
```

Output:

	Name	college	branch	cgpa	package
0	nitish	bit	eee	6.66	4.0
1	ankit	iit	it	8.25	5.0
2	rupesh	vit	cse	6.41	6.0
3	NaN	NaN	NaN	NaN	NaN
4	mrityunjay	NaN	me	5.60	6.0
5	NaN	vlsi	ce	9.00	7.0
6	rishabh	ssit	civ	7.40	8.0
7	NaN	NaN	cse	10.00	9.0
8	aditya	NaN	bio	7.40	NaN
9	NaN	git	NaN	NaN	NaN

code:

```
students = pd.DataFrame( {
        'name':[ 'nitish', 'ankit', 'rupesh', np.nan, 'mrityunjay',
                 np.nan, 'rishabh', np.nan, 'aditya', np.nan ],
        'college':[ 'bit', 'iit', 'vit', np.nan,np.nan, 'vlsi', 'ssit',
                    np.nan, np.nan, 'git' ],
        'branch':[ 'eee', 'it', 'cse', np.nan, 'me', 'ce', 'civ',' cse',
                   'bio', np.nan ],
        'cgpa':[6.66, 8.25, 6.41, np.nan, 5.6, 9.0, 7.4,10, 7.4,
                 np.nan ],
        'package':[4, 5, 6, np.nan, 6, 7, 8, 9, np.nan, np.nan ] } )
print(students)
# Sorting with NaN values
students.sort_values(by='name', na_position='first')
```

Output:

	Name	college	branch	cgpa	package
3	NaN	NaN	NaN	NaN	NaN
5	NaN	vlsi	ce	9.00	7.0
7	NaN	NaN	cse	10.00	9.0
9	NaN	git	NaN	NaN	NaN
8	aditya	NaN	bio	7.40	NaN
1	ankit	iit	it	8.25	5.0
4	mrityunjay	NaN	me	5.60	6.0
0	nitish	bit	eee	6.66	4.0
6	rishabh	ssit	civ	7.40	8.0
2	rupesh	vit	cse	6.41	6.0

rank() method: (series)

- Used to compute numerical data ranks (1 through n) along axis.
- After sorting (by default in ascending order), the position is used to determine the rank that is returned.
- If data contains equal values, then they are assigned with the average of the ranks of each value by default.
- **Syntax:** Series.rank(axis=0, method='average', numeric_only= NoDefault. no_default, na_option='keep',ascending=True, pct=False)

Parameters:

axis:	index to direct ranking
method:	{'average', 'min', 'max', 'first', 'dense'}
numeric_only:	Include only float, int, boolean data. Valid only for DataFrame or Panel objects
na_option :	{'keep', 'top', 'bottom'}
ascending :	False for ranks by high (1) to low (N)
pct :	Computes percentage rank of data

Code:

```
import pandas as pd
batsman = pd.read_csv('batsman_runs_ipl.csv')
# apply rank() function on specific column
batsman['ranks'] =
batsman['batsman_run'].rank(ascending=False)
batsman.sort_values('ranks')
```

Output:

	batter	batsman_run	batting_rank
569	V Kohli	6634	1.0
462	S Dhawan	6244	2.0
130	DA Warner	5883	3.0
430	RG Sharma	5881	4.0
493	SK Raina	5536	5.0

sort_index(): (Series or DataFrame)

- Pandas Series.sort_index() function is used to sort the index labels of the given series or DataFrame.
- **Syntax:** Series.sort_index(axis=0, ascending=True, inplace=False, kind='quicksort', na_position='last')

Code:

```
import pandas as pd
marks = { 'maths' : 78,  'english' : 70, 'science' : 89 }
marks_series = pd.Series(marks)
print(marks_series)
```

Output:

```
maths    78
english  70
science  89
dtype: int64
```

 # alphabetically sorted index for string index

Code:

```
print(marks_series.sort_index())
# print(marks_series.sort_index(ascending=False))
```

Output:

```
english  70
maths    78
science  89
dtype: int64
```

set_index() : (dataframe)

- Used to re-assign a row label using the existing column of the DataFrame.
- It can assign one or multiple columns as a row index.
- **Syntax:** DataFrame.set_index(keys, drop=True, append=False, inplace=False, verify_integrity=False)

Parameters:

keys:	Column name or list of column name.
append:	Appends the column to existing index column if True.
inplace:	Makes the changes in the dataframe if True.
verify_integrity:	Checking new index column for duplicates if True.

Code:

```
import pandas as pd
# set batter column as index
batsman.set_index('batter').head()
# batsman.set_index('batter',inplace=True)
```

Output:

batter	batsman_run	batting_rank
A Ashish Reddy	280	166.5
A Badoni	161	226.0
A Chandila	4	535.0
A Chopra	53	329.0
A Choudhary	25	402.5

reset_index() : (series and dataframe)

- It is opposite to set_index()
- Convert the series into dataframe.
- Function reset the index to the default integer index beginning at 0, We can simply use the reset_index() function.

• **Syntax:** DataFrame.reset_index(inplace=False)

Code:

```
import pandas as pd
# reset the index column
batsman.reset_index().head()
# batsman.reset_index(inplace=True)
```

Output:

	batter	batsman_run	batting_rank
0	A Ashish Reddy	280	166.5
1	A Badoni	161	226.0
2	A Chandila	4	535.0
3	A Chopra	53	329.0
4	A Choudhary	25	402.5

 # How to replace existing index without loosing

Code:

```
batsman.reset_index().set_index('batting_rank').head(3)
# series to dataframe using reset_index
marks_series.reset_index()
```

Output:

batting_rank	index	batter	batsman_run
166.5	0	A Ashish Reddy	280
226.0	1	A Badoni	161
535.0	2	A Chandila	4

Output:

	index	0
0	maths	67
1	english	57
2	science	89

rename() method: (dataframe)

• Used to rename any index, column, or row.

• **Syntax:** DataFrame.rename(mapper, index, columns, axis, copy, inplace, level)

Parameters:

mapper, index, columns:	Dictionary value, key refers to the old name and value refers to new name. Only one of these parameters can be used at once.
axis:	int or string value, 0/'row' for Rows and 1/'columns' for columns.
copy:	copies underlying data if True.
inplace:	makes changes in original Data Frame if True.
level:	specify level in dataframe if it have multiple level index.

Code:

```
import pandas as pd
# display specific columns to from movies dataset
movie_list = movies[['title_x','imdb_rating','imdb_votes']].head(3)
movie_list
```

Output:

	title_x	imdb_rating	imdb_votes
0	Uri: The Surgical Strike	8.4	35112
1	Battalion 609	4.1	73
2	The Accidental Prime Minister (film)	6.1	5549

 # Set the index and rename the columns

Code:

```
movie_list.set_index('title_x', inplace=True)
movie_list.rename(columns={'imdb_votes' : 'votes',
                           'imdb_rating' : 'rating'}).head(3)
```

Output:

title_x	rating	votes
Uri: The Surgical Strike	8.4	35112
Battalion 609	4.1	73
The Accidental Prime Minister (film)	6.1	5549

 # Rename the index labels

Code:

```
movie_list.rename(index={ 'Uri: The Surgical Strike' : 'Uri' ,
                          'Battalion 609' : 'Battalion' }).head(3)
```

Output:

title_x	rating	votes
Uri	8.4	35112
Battalion	4.1	73
The Accidental Prime Minister (film)	6.1	5549

unique() method: (series)

• Return unique values based on a hash table.

• Multiple missing values consider as one NaN value.

• **Syntax**: Series.unique()

Code:

```
import pandas as pd
temp = pd.Series([1,1,2,2,3,3,4,np.nan,np.nan])
print(temp.unique())
# return unique ipl season form ipl dataset
print(ipl['Season'].unique())
# count the unique ipl season form ipl dataset
print(ipl['Season'].unique().shape[0])
```

Output:

```
array([ 1.,  2.,  3.,  4., nan])
['2022' '2021' '2020/21' '2019' '2018' '2017' '2016' '2015' '2014'
 '2013' '2012' '2011' '2009/10' '2009' '2007/08']
15
```

nunique() method: (series and dataframe)

• The number of unique observations over the requested axis.

• It returns Series with a number of distinct observations.

• **Syntax :** DataFrame.nunique(axis=0, dropna=True)

Parameters:

axis:	It represents index or column axis, '0' for index and '1' for the column. When the axis=0, function applied over the index axis and when the axis=1 function applied over the column axis
dropna:	It represents the bool (True or False), and the default is True. It does not include NaN in the counts.

Code:

```
import pandas as pd
# Count unique values of the dataframe
temp = pd.DataFrame([1,1,2,2,3,3,4,np.nan,np.nan])
# count with default parameter
print('without dropna parameter :',temp.nunique()[0])
# count with dropna=False parameter
print('with dropna parameter :',temp.nunique(dropna=False)[0])
# return unique ipl season form ipl dataset
print('ipl season count :',ipl['Season'].nunique())
```

Output:

```
without dropna parameter: 4
with dropna parameter: 5
ipl season count: 15
```

FUNCTIONS/METHODS FOR HANDLING MISSING VALUES

isnull() method: (series and dataframe)

• Returns the DataFrame of the boolean values.

• If the resultant DataFrame consists of the True, it indicates that the element is a null value and if it is False, it indicates that the element is not a null value.

• **Syntax:** Series.isnull()

Code:

```
# apply on entire dataframe
students.isnull()
# return boolean values
students['name'].isnull()
# display nan values
students['name'][students['name'].isnull()]
```

Output:

```
3    NaN
5    NaN
7    NaN
9    NaN
Name: name, dtype: object
```

notnull() method: (series and dataframe)

• Used to detect the existing values.

• It returns a DataFrame consisting of bool values for each element in DataFrame that indicates whether an element is not a null value.

- While detecting the existing values, this method does not consider the characters such as empty strings '' or numpy.inf as null values.
- **Syntax:** DataFrame.notnull()

Code:
```
# apply on entire dataframe
students.notnull()
# detecting existing values
students['name'][students['name'].notnull()]
```
Output:
```
0       nitish
1        ankit
2       rupesh
4    mrityunjay
6       rishabh
8        aditya
Name: name, dtype: object
```

hasnans attribute: (series)
- Returns a boolean value.
- It returns True if the given Series object has missing values in it else it returns False.
- **Syntax:** Series.hasnans

Code:
```
students['name'].hasnans
```
Output:
```
True
```

drona() method: (series or dataframe)
- It removes the missing values and returns the DataFrame with NA entries dropped from it or None if inplace=True.
- **Syntax:** DataFrame.dropna(axis=0, how='any', thresh, subset, inplace=False)

Parameters:

axis:	{0 or 'index', 1 or 'columns'}, default 0. Determine if rows or columns which contain missing values are removed. • 0, or 'index': Drop rows that contain missing values • 1, or 'columns': Drop columns that contain the missing value
how:	{'any', 'all'}, default 'any'. Determine if row or column is removed from DataFrame when we have at least one NA or all NA. • 'any': If any NA values are present, drop that row or column. • 'all': If all values are NA, drop that row or column.
thresh:	int, optional. Require that many non-NA values.
subset:	array-like, optional labels along another axis to consider, e.g. If you are dropping rows these would be a list of columns to include.
inplace:	bool, default False. If True, do operation inplace and return None.

Code:
```
# apply on sereis
students['name'].dropna()
# apply on dataframe & remove rows if any one columns have nan value
students.dropna()
# with how parameter
students.dropna(how='all')
# remove nan value from specific column
students.dropna(subset=[ 'name' ])
# remove nan values based on mutiple columns
students.dropna(subset=[  'name','college' ])
```
Output:

	Name	college	branch	cgpa	package
0	nitish	bit	eee	6.66	4.0
1	ankit	iit	it	8.25	5.0
2	rupesh	vit	cse	6.41	6.0
3	NaN	NaN	NaN	NaN	NaN
4	mrityunjay	NaN	me	5.60	6.0
5	NaN	vlsi	ce	9.00	7.0
6	rishabh	ssit	civ	7.40	8.0
7	NaN	NaN	cse	10.00	9.0
8	aditya	NaN	bio	7.40	NaN
9	NaN	git	NaN	NaN	NaN

fillna() method: (series and dataframe)
- Fills NA/NaN values using the specified method.
- It returns the DataFrame object with missing values filled or None if inplace=True.
- **Syntax:** DataFrame.fillna(value, method, axis, inplace=False, limit, downcast)

Parameters:

value:	scalar, dict, Series, or DataFrame.Value to use to fill
method:	{'backfill', 'bfill', 'pad', 'ffill', None}, default None.
axis:	{0 or 'index', 1 or 'columns'}. Axis along which to fill missing values.

Note: Most of the time apply on specific columns

Code:
```
# Series
students['name'].fillna('unknown')
# Replace students package nan values with its average package
students['package'].fillna(students['package'].mean())
```
Output:
```
0       nitish
1        ankit
2       rupesh
3      unknown
4     mrityunjay
5      unknown
6      rishabh
7      unknown
8       aditya
9      unknown
Name: name, dtype: object

0   4.000000
1   5.000000
2   6.000000
3   6.428571
4   6.000000
5   7.000000
6   8.000000
7   9.000000
8   6.428571
9   6.428571
Name: package, dtype: float64
```

drop_duplicates() method: (series and dataframe)
- It returns a DataFrame with duplicate rows removed.
- Considering certain columns is optional. Indexes, including time indexes, are ignored.
- **Syntax:** DataFrame.drop_duplicates(subset, keep='first',inplace= False, ignore_index=False)

Code:
```
import pandas as pd
import numpy as np
temp = pd.Series([ 1,1,1,2,3,3,4,4 ])
temp.drop_duplicates()
```
Output:
```
0   1
3   2
4   3
6   4
dtype: int64
```

 # drop_duplicates() function with keep parameter

Code:
```
marks = pd.DataFrame( [ [100,80,10], [120,100,14],
                        [80,70,14], [80,70,14] ],
                        columns= [ 'iq', 'marks', 'package' ] )
print(marks)
print(marks.drop_duplicates(keep='last'))
```
Output:

	iq	marks	package
0	100	80	10
1	120	100	14
2	80	70	14
3	80	70	14

	iq	marks	package
0	100	80	10
1	120	100	14
3	80	70	14

drop() method: (series and dataframe)

- It drops specified labels from rows or columns.
- It removes rows or columns by specifying label names and correspondin g axis, or by specifying directly index or column names.
- When using a multi-index, labels on different levels can be removed by s pecifying the level.
- It returns the DataFrame or None. DataFrame without the removed ind ex or column labels or None if inplace=True.
- It raises KeyError exception if any of the labels are not found in the sele cted axis.
- **Syntax:** DataFrame.drop(labels,axis=0,index,columns,level , inplace=False,errors='raise')

Code:

```
temp = pd.Series([10,2,3,8,10])
print(temp)
# apply on sereis with index parameter
print(temp.drop(index=[0,3]))
students.drop(index=[1,3],columns=['branch','cgpa']).head(4)
```

Output:

```
0    10
1     2
2     3
3     8
4    10
dtype: int64
1     2
2     3
4    10
dtype: int64
      name      college    package
0   nitish        bit        4.0
2  rupesh         vit        6.0
1  mrityunjay     NaN        6.0
1    NaN          vlsi       7.0
```

apply() method: (series and dataframe)

- Using this method we can apply different functions on rows and column s of the DataFrame.
- The objects passed to the method are Series objects whose index is eith er the DataFrame's index (axis=0) or the DataFrame's columns (axis=1).
- **Syntax:** DataFrame.apply(func, axis=0, raw=False, result_type, args, **kwds)

Code:

```
points_df = pd.DataFrame({
          '1st point' : [(3,4),(-6,5),(0,0),(-10,1),(4,5)],
          '2nd point' : [(-3,4),(0,0),(2,2),(10,10),(1,1)] })
# print(points_df)
# creating function for apply() function
def euclidean(row):
    point_A = row['1st point']
    point_B = row['2nd point']
    return ((point_A[0] - point_B[0])**2 +
            (point_A[1] - point_B[1])**2)**0.5
# apply on DataFrame and asign new column
points_df['distance'] = points_df.apply(euclidean, axis=1)
print(points_df)
```

Output:

```
    1st point   2nd point    distance
0      (3, 4)      (-3, 4)    6.000000
1     (-6, 5)      (0, 0)     7.810250
2      (0, 0)      (2, 2)     2.828427
3    (-10, 1)    (10, 10)    21.931712
4      (4, 5)      (1, 1)     5.000000
```

nlargest() method: (series and dataframe)

- Returns a specified number of rows, starting at the top after sorting the DataFrame by the highest value for a specified column.
- This method is equivalent to df.sort_values(columns, ascending=False).head(n), but more performant.
- **Syntax:** DataFrame.nlargest(n, columns, keep='last')

Parameters:

n:	Required, a Number, specifying the number of rows to return
columns:	Optional, A String (column label), or a list of column labels, specifying the column(s) to order by
keep:	{'all','first''last'} Optional, default 'last', specifying what to do with d

Code:

```
import pandas as pd
df = pd.Series([10, 20, 65, 0, 30])
largest_values = df.nlargest(3 , keep='last')
# Get 3 largest values
largest_values
```

Output:

```
2   65
4   30
1   20
dtype: int64
```

nsmallest() method: (series and dataframe)

- Used to get n least values from a data frame or a series.
- This method is equivalent to df.sort_values(columns, ascending=True).head(n), but more performant.
- **Syntax:** DataFrame.nsmallest(n, columns, keep='last')

Code:

```
import pandas as pd
df = pd.Series([10, 20, 65, 0, 30])
smallest_values = df.nsmallest(3)
# Get 3 smallest values
smallest_values
```

Output:

```
3   0
0   10
1   20
dtype: int64
```

insert() method: (dataframe)

- Used to insert a column as a specific position in a pandas dataframe
- **Syntax:** df.insert(loc, column, value, allow_duplicates=False)

Parameters:

loc:	(int) The index where the new column is to be inserted. The index must be in the range, 0 <= loc <= len(columns).
column:	(str, num, or hashable object) The label (column name) for the inserted column.
value:	(scaler, series, or array-like) The column values.
allow_duplicates:	(bool) Optional argument. Determines whether you can have duplicate columns or not. It is False by default.

Code:

```
import pandas as pd
data = {
      'Name': ['Jim','Tobi'],
      'Age': [26, 28]
  }
df = pd.DataFrame(data)
df
```

Output:

```
    Name    Age
0   Jim      26
1   Tobi     28
```

 # Insert the New Column at specific position

Code:

```
import pandas as pd
df.insert(1, 'Department' , [ 'Sales' , 'Accounting' ])
df
```

Output:

```
    Name    Department    Age
0   Jim          Sales    26
1   Tobi     Accounting   28
```

copy() method: (series and dataframe)

- Create a copy of a dataframe.
- By default, the copy is a "deep copy" meaning that any changes made in the original DataFrame will NOT be reflected in the copy.
- **Syntax:** df.copy(deep=True)

Parameters:

deep:	Optional. Default True. Specifies whether to make a deep or a shallow copy.

	• By default (deep=True, any changes made in the original DataFrame will NOT be reflected in the copy. • With the parameter deep=False, it is only the reference to the data (and index) that will be copied, and any changes made in the original will be reflected in the copy, and, any changes made in the copy will be reflected in the original.

NOTE:

- Use deep=True (default value) to create a deep copy.
- Use deep=False to create a shallow copy.

Code:

```
import pandas as pd
data = {
        'Name': ['Jim','Tobi'],
        'Age': [26, 28]
  }
df = pd.DataFrame(data)
df
```

Output:

	Name	Age
0	Jim	26
1	Tobi	28

 # create a deep copy

Code:

```
# deep copy is created by default
df1 = df.copy()
df1
```

Output:

	Name	Age
0	Jim	26
1	Tobi	28

 # Make changes to df1

Code:

```
df1.loc[0, 'Name'] = 'Goku'
# display df1
print(df1)
# display the original dataframe
print(df)
```

Output:

	Name	Age
0	Jim	26
1	Tobi	28

	Name	Age
0	Jim	26
1	Tobi	28

 # Create a shallow copy of a pandas dataframe

Code:

```
df2 = df.copy(deep=False)
df2
```

Output:

	Name	Age
0	Jim	26
1	Tobi	28

 # Make changes to df2

Code:

```
df2.loc[0, 'Name'] = 'Goku'
# display df2
print(df2)
# display the original dataframe
print(df)
```

Output:

	Name	Age
0	Jim	26
1	Tobi	28

	Name	Age
0	Jim	26
1	Tobi	28

GROUPBY OBJECT

WHAT IS PANDAS GROUPBY?

- Pandas groupby splits all the records from your data set into different categories or groups so that you can analyze the data by these groups.
- When you use the groupby() function on any categorical column of DataFrame, it returns a Groupby object, which you can use other methods on to group the data.
- Generally, we two types of columns in datasets numerical and categorical
- **Numerical Columns:**
  Numerical columns contain data that consists of numbers. These numbers can be integers or floating-point numbers (decimals). Examples of numerical columns include columns like "Age," "Salary," "Temperature," "Number of Items Sold," and "Height.
- **Categorical Columns:**
  Categorical columns contain data that represents categories or discrete values. These values are often labels or strings.
  Categorical columns include columns like "Gender" (with values like "Male" and "Female"), "Product Category" (with values like "Electronics," "Clothing," and "Furniture"), and "Country" (with values like "USA," "Canada," and "UK").

NOTE: groupby() always apply on categorical columns

- **Syntax:** DataFrame.groupby(by, axis=0, level, as_index=True, sort=True, group_keys=True, squeeze=False, **kwargs)

Parameters:

by:	mapping, function, str, or iterables
axis:	int, default 0
level:	If the axis is a MultiIndex (hierarchical), group by a particular level or levels
as_index:	For aggregated output, return object with group labels as the index. Only relevant for DataFrame input. as_index=False is effectively "SQL-style" grouped output
sort:	Sort group keys. Get better performance by turning this off. Note this does not influence the order of observations within each group. groupby preserves the order of rows within each group.
group_keys:	When calling apply, add group keys to index to identify pieces
squeeze:	Reduce the dimensionality of the return type if possible, otherwise return a consistent type

 # Importing the imdb movies datasets

Code:

```
import pandas as pd
movies = pd.read_csv('imdb-top-1000.csv')
movies.head(20)
```

Output:

Output not taken

 # Creating group by object

Code:

```
import pandas as pd
genres = movies.groupby('Genre')
```

Output:

Genre	Runtime	IMDB_Rating
Action	129.046512	7.949419
Adventure	134.111111	7.937500
Animation	99.585366	7.930488
Biography	136.022727	7.938636
Comedy	112.129032	7.901290

 # Applying builtIn aggregation function on groupby objects

Code:

```
import pandas as pd
genres.mean() # sum() min() mode() median() std() etc
genres.mean()[['Runtime' , 'IMDB_Rating']].head()
```

Output:

Genre	Runtime	IMDB_Rating
Action	129.046512	7.949419
Adventure	134.111111	7.937500
Animation	99.585366	7.930488
Biography	136.022727	7.938636
Comedy	112.129032	7.901290

 # Find the top 3 genres by total earning

Code:

```
import pandas as pd
movies.groupby('Genre').sum()['Gross'].sort_values(
ascending=False).head(3)
```

Output:

```
Genre
Drama     3.540997e+10
Action    3.263226e+10
Comedy    1.566387e+10
Name: Gross, dtype: float64
```

 # Efficient way

Code:

```
import pandas as pd
movies.groupby('Genre')['Gross'].sum().sort_values(ascending=
False).head(3)
```

Output:

```
Genre
Drama      3.540997e+10
Action     3.263226e+10
Comedy     1.566387e+10
Name: Gross, dtype: float64
```

 # Find the genre with highest average IMDB rating

Code:

```
import pandas as pd
movies.groupby('Genre')['IMDB_Rating'].mean().sort_values(asc
ending=False).head(1)
```

Output:

```
Genre
Western    8.35
Name: IMDB_Rating, dtype: float64
```

 # find director with most popularity

Code:

```
import pandas as pd
movies.groupby('Director')['No_of_Votes'].sum().sort_values(asc
ending=False).head(1)
```

Output:

```
Director
Christopher Nolan    11578345
Name: No_of_Votes, dtype: int64
```

 # Find number of movies done by each actor

Code:

```
import pandas as pd
# movies['Star1'].value_counts()
movies.groupby('Star1')['Series_Title'].count().sort_values
                        (ascending=False).head(3)
```

Output:

```
Star1
Tom Hanks            12
Robert De Niro       11
Clint Eastwood       10
Name: Series_Title, dtype: int64
```

GROUPBY ATTRIBUTES AND METHODS/FUNCTION

len() method :

- To find the total number of groups created by the groupby operation.

 # find total number of groups

Code:

```
import pandas as pd
len(movies.groupby('Genre')) # or movies['Genre'].nunique()
```

Output:

```
14
```

size() method:

- To find the number of items in each group.

 # find items in each group size & sort based on index

Code:

```
import pandas as pd
movies.groupby('Genre').size().head()
```

Output:

```
Genre
Action          172
Adventure       72
Animation       82
Biography       88
Comedy          155
dtype: int64
```

 # sort based on values

Code:

```
import pandas as pd
movies['Genre'].value_counts().head()
```

Output:

```
Drama          289
Action         172
Comedy         155
Crime          107
Biography       88
Name: Genre, dtype: int64
```

first() & last() method:

- To retrieve the first or last item within each group.

Code:

```
import pandas as pd
genres = movies.groupby('Genre')
# retrive the first item of each group using groupby
genres[ 'Series_Title', 'Runtime', 'IMDB_Rating' ].first().head()
# retrive the last item of each group using groupby
genres[ 'Series_Title', 'Runtime', 'IMDB_Rating' ].last().head()
```

Output:

Genre	Series_Title	Runtime	IMDB_Rating
Action	The Dark Knight	152	9.0
Adventure	Interstellar	169	8.6
Animation	Sen to Chihiro no kamikakushi	125	8.6
Biography	Schindler's List	195	8.9
Comedy	Gisaengchung	132	8.6

Genre	Series_Title	Runtime	IMDB_Rating
Action	Escape from Alcatraz	112	7.6
Adventure	Kelly's Heroes	144	7.6
Animation	The Jungle Book	78	7.6
Biography	Midnight Express	121	7.6
Comedy	Breakfast at Tiffany's	115	7.6

nth() method:

- To retrieve the nth item from each group within a DataFrame after performing a groupby operation.
- **Syntax:** DataFrameGroupBy.nth(n, dropna='all')

Code:

```
import pandas as pd
# nth()
genres['Series_Title'].nth(7).head()
```

Output:

```
Genre
Action                Star Wars
Adventure                Queen
Animation        Mononoke-hime
Biography              Amadeus
Comedy                  Amélie
Name: Series_Title, dtype: object
```

get_group() method:

To retrieve a specific group by its name, which is useful for selective group access as opposed to filtering.

Code:

```
import pandas as pd
genres['Series_Title','Genre'].get_group('Family')
```

Output:

	Series_Title	Genre
688	E.T. the Extra-Terrestrial	Family
698	Willy Wonka & the Chocolate Factory	Family

groups attribute:

- To access the groups as a dictionary where keys are unique group labels and values are group indices.

 # index position of movies based on genres

Code:

```
import pandas as pd
genres.groups
```

Output:

```
{'Action': [2, 5, 8, 10, 13, 14, 16, 29, 30, 31, 39, 42, 44, 55, 57, 59, 60
  , 68, 72, 106, 109, 129, 130, 134, 140, 142, 144, 152, 155, 160, ...],
  ...
 'Thriller': [700], 'Western': [12, 48, 115, 691]}
```

describe() method:

- To generate descriptive statistics for each group, providing information like mean, std deviation, min, max, and more.

 # describe each column or specified columns

Code:

```
import pandas as pd
# genres.describe()
genres.describe()['Runtime'].head()
```

Output:

Genre	Count	mean	...	75%	max
Action	172.0	129.046512	...	143.25	321.0
Adventure	72.0	134.111111	...	149.00	228.0
Animation	82.0	9.585366	...	106.75	137.0
Biography	88.0	136.022727	...	146.25	209.0
Comedy	155.0	112.129032	...	124.50	188.0

sample() method:

- To obtain a random sample from each group.
- **Syntax:** pd.DataFrame.groupby().sample(n, replace=False)

Parameters:

n:	Sample size to return for each group.
replace:	default false and allow or disallow sampling of the same row more than once.

Code:

```
import pandas as pd
#genres.sample()
# genres.sample(2, replace=True)
genres.sample(2, replace=True)[['Series_Title', 'Released_Year',
                                'Genre']].head(4)
```

Output:

	Series_Title	Released_Year	Genre
900	Serbuan maut	2011	Action
223	Mad Max: Fury Road	2015	Action
675	Back to the Future Part II	1989	Adventure
406	The Princess Bride	1987	Adventure

nunique() method:

- To count the number of unique values within each group.

Code:

```
import pandas as pd
# genres.nunique()
genres.nunique()[['Series_Title','Released_Year']].head()
```

Output:

Genre	Series_Title	Released_Year
Action	172	61
Adventure	72	49
Animation	82	35
Biography	88	44
Comedy	155	72

agg() method :

- Apply multiple aggrigation function at same time.
- **Syntax :** DataFrameGroupBy.agg(arg, *args, **kwargs)

 # Passing dictionary

Code:

```
import pandas as pd
genres.agg(
    {
      'Runtime' : 'mean',
      'IMDB_Rating' : 'mean',
      'Gross' : 'sum',
    }
).head()
```

Output:

Genre	Runtime	IMDB_Rating	Gross
Action	129.046512	7.949419	3.263226e+10
Adventure	134.111111	7.937500	9.496922e+09
Animation	99.585366	7.930488	1.463147e+10
Biography	136.022727	7.938636	8.276358e+09
Comedy	112.129032	7.901290	1.566387e+10

 # Passing list

Code:

```
import pandas as pd
genres.agg([ 'min' , 'max' , 'mean' ]).[ 'Runtime'].head()
```

Output:

	Runtime		
	min	max	mean
Genre			
Action	45	321	129.046512
Adventure	88	228	134.111111
Animation	71	137	99.585366
Biography	93	209	136.022727
Comedy	68	188	112.129032

Adding both dictionary and list

Code:

```
import pandas as pd
genres.agg( { 'Runtime' : [ 'max' , 'min' , 'mean' ],
              'IMDB_Rating' : [ 'max' , 'min' ], }).head()
```

Output:

	Runtime			IMDB_Rating	
	min	max	mean	min	max
Genre					
Action	45	321	129.046512	9.0	7.6
Adventure	88	228	134.111111	8.6	7.6
Animation	71	137	99.585366	8.6	7.6
Biography	93	209	136.022727	8.9	7.6
Comedy	68	188	112.129032	8.6	7.6

LOOPING ON GROUPS

- Groupby is a method in Pandas that allows you to group a DataFrame or Series by one or more columns.
- Once you have your data grouped, you can perform various operations on each group, such as aggregation, transformation, or filtering.

LOOP OVER GROUPBY

- Once you have your data grouped using the groupby method, you can loop over each group using a for loop.
- **Syntax:**

```
for group_name, group_data in df.groupby('column_name'):
    #perform some analysis or visualization
```

 # Find the highest rated movies of each genre

Code:

```
import pandas as pd
df = pd.DataFrame(columns=movies.columns)
for group, data in genres:
df = df.append(data[data['IMDB_Rating'] ==  data['IMDB_Rating'].
     max()])
df[['Series_Title', 'IMDB_Rating', 'Genre']].head()
```

Output:

	Series_Title	IMDB_Rating	Genre
2	The Dark Knight	9.0	Action
21	Interstellar	8.6	Adventure
23	Sen to Chihiro no kamikakushi	8.6	Animation
7	Schindler's List	8.9	Biography
19	Gisaengchung	8.6	Comedy

SPLIT-APPLY-COMBINE STRATEGY

By “group by” we are referring to a process involving one or more of the f ollowing steps:

- **Splitting** the data into groups based on some criteria.
- **Applying** a function to each group independently.
- **Combining** the results into a new DataFrame.

 # Find number of movies starting with A for each group

Code:

```
import pandas as pd
def alphabet(group):
    return group['Series_Title'].str.startswith('A').sum()
genres.apply(alphabet).head()
```

Output:

```
Genre
Action       10
Adventure    2
Animation    2
Biography    9
Comedy       14
dtype: int64
```

 # Find ranking of each movie in the group according to IMDB score

Code:

```
def ranking(group):
```

```
        # assign the new column for ranking
        group['genre_rank'] = group['IMDB_Rating'].rank
                                        (ascending=False)
    return group
    genres['Series_Title', 'IMDB_Rating'].apply(ranking).head()
```

Output:

	Series_Title	IMDB_Rating	genre_rank
0	The Shawshank Redemption	9.3	1.0
1	The Godfather	9.2	1.0
2	The Dark Knight	9.0	1.0
3	The Godfather: Part II 9.0	2.5	
4	12 Angry Men	9.0	2.5

 # Find normalized IMDB rating group wise
 # normalized formula: (X - Xmin) / (Xmax - Xmin)

Code:

```
import pandas as pd
def normalized(group):
    X = group['IMDB_Rating']
    Xmin = group['IMDB_Rating'].min()
    Xmax = group['IMDB_Rating'].max()
    group['Normalized_Rating'] = (X - Xmin) / (Xmax - Xmin)
    return group
genres['Series_Title', 'IMDB_Rating'].apply(normalized)
```

Output:

	Series_Title	IMDB_Rating	Normalized_Rating
0	The Shawshank Redemption	9.3	1.0
1	The Godfather	9.2	1.0
2	The Dark Knight	9.0	1.0
3	The Godfather: Part II	9.0	2.5
4	12 Angry Men	9.0	2.5

 # Groupby on multiple cols

Code:

```
import pandas as pd
duo = movies.groupby(['Director','Star1'])
# size
print(duo.size())
# get_group
duo['Series_Title', 'Director','Star1'].get_group(( 'Aamir Khan' ,
'Amole Gupte' ))
```

Output:

```
Director                Star1
Aamir Khan              Amole Gupte          1
Aaron Sorkin            Eddie Redmayne       1
Abdellatif Kechiche     Léa Seydoux          1
Abhishek Chaubey        Shahid Kapoor        1
Abhishek Kapoor         Amit Sadh            1
                                             ..
Zaza Urushadze          Lembit Ulfsak        1
Zoya Akhtar             Hrithik Roshan       1
                        Vijay Varma          1
Çagan Irmak             Çetin Tekindor       1
Ömer Faruk Sorak        Cem Yilmaz           1
Length: 898, dtype: int64
```

	Series_Title	Director	Star1
65	Taare Zameen Par	Aamir Khan	Amole Gupte

 # Find the most earning actor->director combo

Code:

```
import pandas as pd
duo['Gross'].sum().sort_values(ascending=False).head(1)
```

Output:

```
Director          Star1
Akira Kurosawa    Toshirô Mifune    2.999877e+09
Name: Gross, dtype: float64
```

 # Find the best(in-terms of metascore(avg)) actor->genre combo

Code:

```
import pandas as pd
movies.groupby(['Star1','Genre'])['Metascore'].mean()
 .reset_index().sort_values('Metascore',ascending=False).head()
```

Output:

	Star1	Genre	Metascore
230	Ellar Coltrane	Drama	100.0
329	Humphrey Bogart	Drama	100.0
360	James Stewart	Mystery	100.0
77	Bertil GuveDrama	100.0	
590	Orson Welles	Drama	100.0

 # Agg on multiple groupby

Code:

```
import pandas as pd
duo.agg(['min','max','mean'])[['Runtime','IMDB_Rating']].head(2)
```

Output:

		Runtime		IMDB_Rating	
		min	max	min	max
Director	Star1				
Aamir Khan	Amole Gupte	165	165	8.4	8.4
Aaron Sorkin	Eddie Redmayne	129	129	7.8	7.8

 # Importing the deliveries datasets

Code:

```
import pandas as pd
ipl = pd.read_csv('deliveries.csv')
ipl.info()
```

Output:

```
<class 'pandas.core.frame.DataFrame'>
RangeIndex: 179078 entries, 0 to 179077
Data columns (total 21 columns):
 #   Column            Non-Null Count    Dtype
---  ------            --------------    -----
 0   match_id          179078 non-null   int64
 1   inning            179078 non-null   int64
 2   batting_team      179078 non-null   object
 3   bowling_team      179078 non-null   object
 4   over              179078 non-null   int64
 5   ball              179078 non-null   int64
 6   batsman           179078 non-null   object
 7   non_striker       179078 non-null   object
 8   bowler            179078 non-null   object
 9   is_super_over     179078 non-null   int64
 10  wide_runs         179078 non-null   int64
 11  bye_runs          179078 non-null   int64
 12  legbye_runs       179078 non-null   int64
 13  noball_runs       179078 non-null   int64
 14  penalty_runs      179078 non-null   int64
 15  batsman_runs      179078 non-null   int64
 16  extra_runs        179078 non-null   int64
 17  total_runs        179078 non-null   int64
 18  player_dismissed  8834 non-null     object
 19  dismissal_kind    8834 non-null     object
 20  fielder           6448 non-null     object
dtypes: int64(13), object(8)
memory usage: 28.7+ MB
```

 # Find the top 10 batsman in terms of runs

Code:

```
import pandas as pd
batsman = ipl.groupby('batsman')
batsman['batsman_runs'].sum().sort_values(ascending=False)
.head(10)
```

Output:

```
batsman
V Kohli          5434
SK Raina         5415
RG Sharma        4914
DA Warner        4741
S Dhawan         4632
CH Gayle         4560
MS Dhoni         4477
RV Uthappa       4446
AB de Villiers   4428
G Gambhir        4228
Name: batsman_runs, dtype: int64
```

 # Find the batsman with max no of sixes

Code:

```
import pandas as pd
sixes = ipl[ipl['batsman_runs'] == 6]
sixes.groupby('batsman')['batsman'].count().sort_values
(ascending=False).head(1).index[0]
```

Output:

'CH Gayle'

 # Find batsman with most number of 4's and 6's in last 5 overs

Code:

```
import pandas as pd
t_df = ipl[ipl['over'] > 15]
t_df[(t_df['batsman_runs'] == 4) | (t_df['batsman_runs'] == 6)]
t_df.groupby('batsman')['batsman'].count().sort_values
(ascending=False).head(1).index[0]
```

Output:

```
batsman
MS Dhoni        1548
Name: batsman, dtype: int64
```

 # Find V Kohli's record against all teams

Code:

```
import pandas as pd
t _df = ipl[ipl['batsman'] == 'V Kohli']
t_df.groupby('bowling_team')['batsman_runs'].sum().reset_index()
```

Output:

	bowling_team	batsman_runs
0	Chennai Super Kings	749
1	Deccan Chargers	306
2	Delhi Capitals	66
3	Delhi Daredevils	763
4	Gujarat Lions	283
5	Kings XI Punjab	636
6	Kochi Tuskers Kerala	50
7	Kolkata Knight Riders	675
8	Mumbai Indians	628
9	Pune Warriors	128
10	Rajasthan Royals	370
11	Rising Pune Supergiants	83
12	Rising Pune Supergiants	188
13	Sunrisers Hyderabad	509

 # Create a function that can return the highest score of any batsman

Code:

```
import pandas as pd
def highestScore(batsman):
    t_df = ipl[ipl['batsman'] == batsman]
    return t_df.groupby('match_id')['batsman_runs'].sum()
            .sort_values (ascending=False).head(1).values[0]
highestScore('DA Warner')
```

Output:

```
batsman
MS Dhoni        1548
Name: batsman, dtype: int64
```

MERGING, JOINING AND CONCATENATION

 # Datasets

Code:

```
import pandas as pd
courses = pd.read_csv('courses.csv')
students = pd.read_csv('students.csv')
nov = pd.read_csv('reg-month1.csv')
dec = pd.read_csv('reg-month2.csv')
matches = pd.read_csv('matches.csv')
delivery = pd.read_csv('deliveries.csv')
```

pd.concat() method:

- Used to concatenate pandas objects such as DataFrames and Series.
- We can pass various parameters to change the behavior of the concatenation operation.
- **Syntax:** pd.concat(objs, axis, join, ignore_index, keys, levels, names, verify_integrity, sort, copy)

Parameters:

objs:	Series or DataFrame objects
axis:	axis to concatenate along; default = 0
join:	way to handle indexes on other axis; default = 'outer'
ignore_index:	if True, do not use the index values along the concatenation axis; default = False
keys:	sequence to add an identifier to the result indexes; default = None
levels:	specific levels (unique values) to use for constructing a MultiIndex; default = None
names:	names for the levels in the resulting hierarchical index; default = None
verify_integrity:	check whether the new concatenated axis contains duplicates; default = False
sort:	sort non-concatenation axis if it is not already aligned when join is 'outer'; default = False
copy:	if False, do not copy data unnecessarily; default = True

 # Concat the columns vertically(default)

Code:

```
import pandas as pd
registered = pd.concat([nov,dec], ignore_index=True)
registered
```

Output:

	student_id	course_id
0	23	1
1	15	5
2	18	6
3	23	4
4	16	9

Code:

```
import pandas as pd
d1 = {"Name": ["Pankaj", "Lisa"], "ID": [1, 2]}
d2 = {"Name": "David", "ID": 3}
df1 = pd.DataFrame(d1, index=[1, 2])
df2 = pd.DataFrame(d2, index=[3])
df3 = pd.concat([df1, df2])
print(df3)
```

Output:

	Name	ID
1	Pankaj	1
2	Lisa	2
3	David	3

pd.append() method:

- Used to append rows of other data frames to the end of the given data frame, returning a new data frame object.
- Columns not in the original data frames are added as new columns and the new cells are populated with NaN value.
- **Syntax:** DataFrame.append(other, ignore_index=False, verify_integrity=False, sort)

Parameters:

other:	DataFrame or Series/dict-like object, or list of these The data to append.
ignore_index:	If True, do not use the index labels.
verify_integrity:	If True, raise ValueError on creating an index with duplicates.
sortPandas:	default False, Sort columns if the columns of self and other are not aligned.

NOTE:

Append method is deprecated and will be removed from pandas in a future version.

 # Example

Code:

```
import pandas as pd
print(nov.append(dec, ignore_index=True).head())
```

Output:

	student_id	course_id
0	23	1
1	15	5
2	18	6
3	23	4
4	16	9

 # Multiindex dataframe (keep original index as it is)

Code:

```
import pandas as pd
multi = pd.concat([nov, dec], keys=['Nov', 'Dec'])
print(multi)
# accessing each months
# multi.loc['Nov']
# multi.loc['Dec']
# accessing the items
print(multi.loc[('Nov',0)])
```

Output:

		student_id	course_id
Nov	0	23	1
	1	15	5
	2	18	6
	3	23	4
	4	16	9

```
student_id      23
course_id        1
Name: (Nov, 0), dtype: int64
```

 # Concat dataframe horizontally

Code:

```
import pandas as pd
pd.concat([nov,dec], axis=1)
```

Output:

	student_id	course_id	student_id	course_id
0	23.0	1.0	3	5
1	15.0	5.0	16	7
2	18.0	6.0	12	10
3	23.0	4.0	12	1
4	16.0	9.0	14	9

merge() method:

- Used to merge two DataFrame objects with a database-style join operation.
- The joining is performed on columns or indexes.
- If the joining is done on columns, indexes are ignored.
- This function returns a new DataFrame and the source DataFrame objects are unchanged.
- **Syntax:** DataFrame.merge(self, right, how='inner', on, left_on, right_on, left_index=False, right_index=False, sort=False, suffixes=('_x', '_y'))

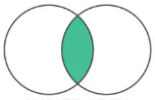
OR

Alternate syntax for merge: Ex. students.merge(regs)

DIFFERENT JOINS

Inner Join:

- An inner join returns only the rows where there is a match in both DataFrames' specified columns.
- It retains only the common elements from both DataFrames.
- Use the pd.merge() function with the how='inner' parameter or the .merge() method with the how='inner' argument



 # Datasets

Code:

```
import pandas as pd
print(students.head())
print (registered.head())
```

Output:

	student_id	name	partner
0	1	Kailash Harjo	23
1	2	Esha Butala	1
2	3	Parveen Bhalla	3
3	4	Marlo Dugal	14
4	5	Kusum Bahri	6

	student_id	course_id
0	23	1
1	15	5
2	18	6
3	23	4
4	16	9

 # inner join

Code:

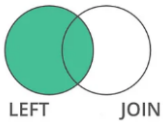
```
import pandas as pd
students.merge(registered, how='inner', on='student_id').head()
```

Output:

	student_id	name	partner	course_id
0	1	Kailash Harjo	23	1
1	1	Kailash Harjo	23	6
2	1	Kailash Harjo	23	10
3	1	Kailash Harjo	23	9
4	2	Esha Butala	1	5

Left Join:

- A left join returns all the rows from the left DataFrame and the matching rows from the right DataFrame.
- If there's no match in the right DataFrame, NaN values are filled in for columns from the right DataFrame.
- Use the pd.merge() function with the how='left' parameter or the .merge() method with the how='left' argument.



 # left join

Code:

```
import pandas as pd
print(courses.head())
# courses : left DataFrame
# registered : Right DataFrame
# join courses and registered dataset
courses.merge(registered, how='left', on='course_id').tail()
```

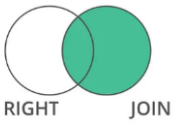
Output:

	course_id	course_name	price
0	1	python	2499
1	2	sql	3499
2	3	data analysis	4999
3	4	machine learning	9999
4	5	tableau	2499

	course_id	course_name	price	student_id
50	10	pyspark	2499	17.0
51	10	pyspark	2499	1.0
52	10	pyspark	2499	11.0
53	11	Numpy	699	NaN
54	12	C++	1299	NaN

Right Join:

- A right join is the opposite of a left join.
- It returns all the rows from the right DataFrame and the matching rows from the left DataFrame.
- If there's no match in the left DataFrame, NaN values are filled in for columns from the left DataFrame.
- Use the pd.merge() function with the how='right' parameter or the .merge() method with the how='right' argument.



 # right join

Code:

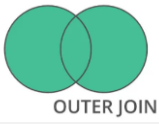
```
import pandas as pd
temp_df = pd.DataFrame({
        'student_id':[26, 27, 28],
        'name':['Akash','Vikas','Rahul'],
        'partner':[28,26,17]
})
students = pd.concat([students, temp_df], ignore_index=True)
students.merge(registered, how='right', on='student_id').tail(3)
```

Output:

	student_id	name	partner	course_id
50	42	NaN	NaN	9
51	50	NaN	NaN	8
52	38	NaN	NaN	1

Outer Join (full outer join):

- An outer join returns all the rows when there is a match in either the left or the right DataFrame.
- If there's no match in one of the DataFrames, NaN values are filled in for the corresponding columns.
- Use the pd.merge() function with the how='outer' parameter or the .merge() method with the how='outer' argument.



 # outer join

Code:

```
import pandas as pd
students.merge(registered, how='outer', on='student_id').tail(7)
```

Output:

student_id	name	partner	course_id

56	25	Shashank D'Alia	2.0	10.0
57	26	Akash	28.0	NaN
58	27	Vikas	26.0	NaN
59	28	Rahul	17.0	NaN
60	42	NaN	NaN	9.0
61	50	NaN	NaN	8.0
62	38	NaN	NaN	1.0

 # 1. Find total revenue generated

Code:

```
import pandas as pd
registered.merge(courses, how='inner', on='course_id')['price']
.sum()
```

Output:

154247

 # 2. Find month by month revenue

Code:

```
import pandas as pd
t_df = pd.concat([nov,dec], keys=['Nov','Dec']).reset_index()
t_df.merge(courses, on='course_id').groupby('level_0')['price']
.sum()
```

Output:

```
level_0
Dec   65072
Nov   89175
Name: price, dtype: int64
```

 # 3. Print the registration table

 # cols -> name -> course -> price

Code:

```
import pandas as pd
stu = registered.merge(students, on='student_id')
stu.merge(courses, on='course_id')[['name','course_name','price']]
.head()
```

Output:

	name	course_name	price
0	Chhavi Lachman	python	2499
1	Preet Sha	python	2499
2	Fardeen Mahabir	python	2499
3	Kailash Harjo	python	2499
4	Seema Kota	python	2499

 # 4.Find students who enrolled in both the monthsnov

Code:

```
import pandas as pd
common_student_id = np.intersect1d(nov['student_id'],
                                   dec['student_id'])
print(common_student_id)
students[students['student_id'].isin(common_student_id)]
```

Output:

```
array([ 1,  3,  7, 11, 16, 17, 18, 22, 23], dtype=int64)
```

	student_id	name	partner
0	1	Kailash Harjo	23
2	3	Parveen Bhalla	3
6	7	Tarun Thaker	9
10	11	David Mukhopadhyay	20
15	16	Elias Dodiya	25
16	17	Yasmin Palan	7
17	18	Fardeen Mahabir	13
21	22	Yash Sethi	21
22	23	Chhavi Lachman	18

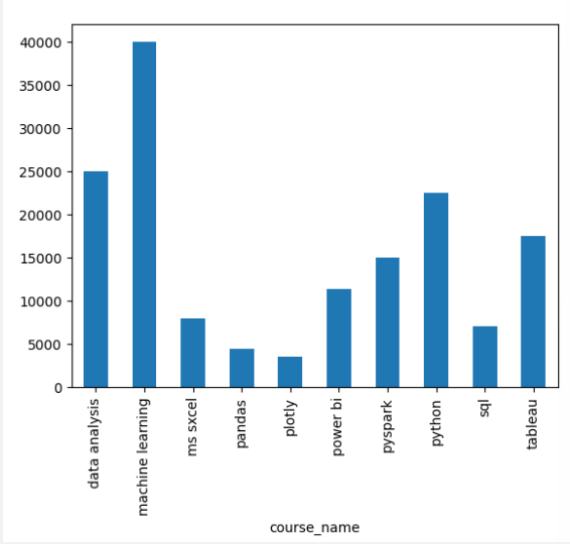
 # 5. Plot bar chart for revenue/course

Code:

```
import pandas as pd
registered.merge(courses,on='course_id').groupby('course_name')
['price'].sum().plot(kind='bar')
```

Output:

```
<AxesSubplot: xlabel='course_name'>
```



 # 6. Find course that got no enrollment

courses['course_id']

regs['course_id']

Code:

```
import pandas as pd
course_list = np.setdiff1d(courses['course_id'],
                           registered['course_id'])
courses[courses['course_id'].isin(course_list)]
```

Output:

	course_id	course_name	price
10	11	Numpy	699
11	12	C++	1299

Self Join:

- A self-join in pandas is a way to combine rows from a single DataFrame by creating a relationship between columns within that DataFrame.
- This is useful when you have hierarchical or relational data stored within a single DataFrame.
- You can achieve a self-join by using the .merge() method or the .join() method.

 # 9. Find top 3 students who did most number enrollments

Code:

```
import pandas as pd
registered.merge(students, on='student_id').groupby
(['student_id','name'])['name'].count().sort_values
(ascending=False).head(3)
```

Output:

```
student_id      name
23                Chhavi Lachman     6
7                  Tarun Thaker          5
1                  Kailash Harjo         4
Name: name, dtype: int64
```

#10. Find top 3 students who spent most amount of money on courses

Code:

```
import pandas as pd
registered.merge(students, on='student_id').merge(courses,
on='course_id').groupby(['student_id','name'])['price'].sum()
.head(3)
```

Output:

```
student_id   name
1                Kailash Harjo       7596
2                Esha Butala          2499
3                Parveen Bhalla      7498
Name: price, dtype: int64
```

 # IPL Problems

 # Find top 3 stadiums with highest sixes/match ratio

Code:

```
import pandas as pd
temp_df = delivery.merge(matches, left_on='match_id',
                         right_on='id')
sixes_df = temp_df[temp_df['batsman_runs'] == 6]
number_of_sixes = sixes_df.groupby('venue')['venue'].count()
number_of_matches = matches['venue'].value_counts()
(number_of_sixes/number_of_matches).sort_values
(ascending=False).head(3)
```

Output:

```
Holkar Cricket Stadium     17.600000
M Chinnaswamy Stadium      13.227273
Sharjah Cricket Stadium    12.666667
Name: venue, dtype: float64
```

 # Find orange cap holder of all the seasons

Code:

```
import pandas as pd
temp_df = delivery.merge(matches, left_on='match_id',
                                  right_on='id')
temp_df.groupby(['season','batsman'])['batsman_runs'].sum()
.reset_index() .sort_values('batsman_runs',ascending=False)
.drop_duplicates(subset=['season'], keep='first')
.sort_values('season')
```

Output:

	season	batsman	batsman_runs
115	2008	SE Marsh	616
229	2009	ML Hayden	572
446	2010	SR Tendulkar	618
502	2011	CH Gayle	608
684	2012	CH Gayle	733
910	2013	MEK Hussey	733
1088	2014	RV Uthappa	660
1148	2015	DA Warner 562	
1383	2016	V Kohli	973
1422	2017	DA Warner 641	

MULTIINDEX SERIES AND DATAFRAMES

MULTIINDEX SERIES

```
# can we have multiple index ? Let's try
index_val = [('cse',2019),('cse',2020),
             ('cse',2021),('cse',2022),
             ('ece',2019),('ece',2020),
             ('ece',2021),('ece',2022)]
a = pd.Series([1,2,3,4,5,6,7,8], index=index_val)
a
```

```
(cse, 2019)    1
(cse, 2020)    2
(cse, 2021)    3
(cse, 2022)    4
(ece, 2019)    5
(ece, 2020)    6
(ece, 2021)    7
(ece, 2022)    8
dtype: int64
```

PROBLEM IN MULTIINDEX SERIES !

- What if we want to fetch values based on specific branch.
- Here branch and year is not independent then we cannot access specific branch, so it will raise KeyError.

```
# KeyError: 'cse'
a['cse']
```

MULTIINDEX SERIES (HIERARCHICAL INDEXING)

- In pandas, a MultiIndex, also known as hierarchical indexing, allows you to have multiple index levels within single axis.
- This is particularly useful when you're working with higher-dimensional data that can be naturally represented as a hierarchical structure.
- MultiIndex can be applied to both rows (index) and columns, but we'll focus on row-based MultiIndexing here pd.MultiIndex is a class in the pandas library in Python that represents a multi-level or hierarchical index.
- It allows you to have multiple levels of indices on one axis, either for rows or columns in a DataFrame.
- syntax: pd.MultiIndex(levels=[level_values1, level_values2, ...], codes=[code_values1, code_values2, ...],names=[name1, name2, ...])

parameters:

levels:	A list of arrays containing the unique values for each level of the MultiIndex.
codes:	A list of arrays containing the integer codes that represent the labels for each level. The codes are indices into the corresponding levels.
names:	A list of names for each level. Names are optional and are used to provide meaningful labels to the levels.

CREATING A MULTIINDEX (HIERARCHICAL INDEX) OBJECT

- We can create a MultiIndex in several ways. One common way is to pass a list of arrays or tuples as the index when creating a DataFrame
- Also you can create a MultiIndex by using the pd.MultiIndex constructor or A MultiIndex can be created from a list of arrays (using MultiIndex.from_arrays()), an array of tuples (using MultiIndex.from_tuples()), a crossed set of iterables (using MultiIndex.from_product()), or a DataFrame (using MultiIndex.from_frame()).
- The Index constructor will attempt to return a MultiIndex when it is passed a list of tuples.

```
# pd.MultiIndex.from_tuples():
index_val = [('cse',2019),('cse',2020),('cse',2021),
             ('ece',2019),('ece',2020),('ece',2021)]
multiindex = pd.MultiIndex.from_tuples(index_val)
multiindex
```

```
MultiIndex([('cse', 2019),
            ('cse', 2020),
            ('cse', 2021),
            ('ece', 2019),
            ('ece', 2020),
            ('ece', 2021)],
           )
```

```
# pd.MultiIndex.from_product()
# it's like cartesion product
pd.MultiIndex.from_product([['cse','ece'],
                            [2019,2020,2021,2022]])
```

```
MultiIndex([('cse', 2019),
            ('cse', 2020),
            ('cse', 2021),
            ('cse', 2022),
            ('ece', 2019),
            ('ece', 2020),
            ('ece', 2021),
            ('ece', 2022)],
           )
```

levels attribute:

```
multiindex.levels
```

```
FrozenList([['cse', 'ece'], [2019, 2020, 2021]])
```

```
multiindex.levels[0]
```

```
Index(['cse', 'ece'], dtype='object')
```

```
multiindex.levels[1]
```

```
Int64Index([2019, 2020, 2021], dtype='int64')
```

CREATING A SERIES WITH MULTIINDEX OBJECTS

```
#Creating MultiIndex
multi_i = pd.MultiIndex.from_product([
    ['cse','ece'],
    [2019,2020,2021]])
# Creating a Series with the MultiIndex
ms = pd.Series([1,2,3,4,5,6], index=multi_i)
ms
```

```
cse  2019    1
     2020    2
     2021    3
ece  2019    4
     2020    5
     2021    6
dtype: int64
```

```
# how to fetch items from such a series
ms['cse']
ms['ece']
```

```
2019    4
2020    5
2021    6
dtype: int64
```

unstack() function :

- Convert the multiindex series into Dataframe

```
temp = ms.unstack()
temp
```

	2019	2020	2021
cse	1	2	3
ece	4	5	6

stack() function :

- Convert the dataframe into multiindex series

```
temp.stack()
```

```
cse  2019    1
     2020    2
     2021    3
ece  2019    4
     2020    5
     2021    6
dtype: int64
```

MAIN PURPOSE OF MULTIINDEXING OBJECTS

To representation of high dimensional data into lower dimensions like 1D(Series) and 2D(DataFrame)

MULTIINDEX DATAFRAME

- A multi-index DataFrame in pandas refers to a DataFrame that has multiple levels of indexing for both rows and columns.
- It is a way of handling higher-dimensional data by creating a hierarchical index structure.

```
branch_df1 = pd.DataFrame(
    [
        [1,2],[3,4],
        [5,6],[7,8],
        [9,10],[11,12],
    ],
    index = multiindex,
    columns = ['avg_package','students'])
branch_df1
```

		avg_package	students
cse	**2019**	1	2
	2020	3	4
	2021	5	6
ece	**2019**	7	8
	2020	9	10
	2021	11	12

ARE COLUMNS REALLY DIFFERENT FROM INDEX?

- In pandas, if you transpose a DataFrame using the T attribute or the transpose() method, the columns effectively become the index and vice versa.
- After transposing, what were originally columns become the index, and what were originally the index becomes the columns.

MULTIINDEX DATAFRAME BASED ON COLUMNS

The MultiIndex has two levels: the first level consists of city names ('delhi' and 'mumbai'), and the second level consists of attributes ('avg_package' and 'students').

```
# multiindex in columns
branch_df2 = pd.DataFrame(
    [
        [1,2,0,0],
        [3,4,0,0],
        [5,6,0,0],
    ],
    index = [2019,2020,2021],
    columns = pd.MultiIndex.from_product([
        ['delhi','mumbai'],['avg_package','students']
    ])
)
branch_df2
```

	delhi		mumbai	
	avg_package	**students**	**avg_package**	**students**
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0

```
branch_df2['mumbai']
```

	avg_package	students
2019	0	0
2020	0	0
2021	0	0

```
branch_df2['delhi']['avg_package']
```

```
2019    1
2020    3
2021    5
Name: avg_package, dtype: int64
```

```
branch_df2.loc[2020]
```

```
delhi   avg_package    3
        students       4
mumbai  avg_package    0
        students       0
Name: 2020, dtype: int64
```

unstack() method:

- In simple words, unstack() method specified index labels becomes new columns and return new dataframe.
- It's used to pivot specified levels of the index labels into new columns, returning a new DataFrame.
- **syntax:** DataFrame.unstack(level=1, fill_value=None)

parameters:

level:	(default is 1) Specifies the level(s) of the index to unstack. If you have a MultiIndex, you can choose which level(s) you want to move to columns.
fill_value:	If there are missing values after unstacking, you can specify a value to replace those missing values. By default, missing values are filled with NaN.

```
branch_df1.unstack(level=1)
```

	avg_package			students		
	2019	**2020**	**2021**	**2019**	**2020**	**2021**
cse	1	3	5	2	4	6
ece	7	9	11	8	10	12

```
branch_df1.unstack(level=0)
```

	avg_package		students	
	cse	**ece**	**cse**	**ece**
2019	1	7	2	8
2020	3	9	4	10
2021	5	11	6	12

stack() function :

- In simple words, Convert the dataframe into multiindex series

- Used to reshape a DataFrame by moving or pivoting specified levels of columns to become inner-most levels of the index.
- This operation results in a new DataFrame or Series with a multi-level index.

Here's a simpler explanation:

- Imagine you have a table (DataFrame) where some information is stored in both rows and columns.
- The stack() method allows you to take information from the columns and move it to the rows, creating a new structure.
- If your columns have only one level (like a regular DataFrame), using stack() will give you a Series.
- If your columns have multiple levels, you can choose which levels to move to the index, and the result will be a new DataFrame with a multi-level index.
- In essence, stack() helps you transform data by rearranging it from a wide format (with information in columns) to a long format (with information in rows).
- This can be useful for certain types of analyses or when you need the data in a different structure.

```
branch_df1.unstack()
```

	avg_package			students		
	2019	2020	2021	2019	2020	2021
cse	1	3	5	2	4	6
ece	7	9	11	8	10	12

```
# Here most inner columns becomes row
branch_df1.unstack().stack()
```

		avg_package	students
cse	2019	1	2
	2020	3	4
	2021	5	6
ece	2019	7	8
	2020	9	10
	2021	11	12

WORKING WITH MULTIINDEX DATAFRAME

We can use pandas DataFrame methods, functions, and attributes on a MultiIndex DataFrame just like you would on a regular DataFrame.

```
# examples
branch_df3.head()
branch_df3.shape
branch_df3.info()
branch_df3.duplicated()
branch_df3.isnull()
```

EXTRACTING ROWS AND COLUMNS

To extract rows and columns from a MultiIndex DataFrame, you can use various methods, including .loc[], .iloc[], and other DataFrame indexing techniques. Here are some examples:

```
# single row
branch_df3.loc[('cse',2019)]
```

```
delhi   avg_package    1
        students       2
mumbai  avg_package    0
        students       0
Name: (cse, 2019), dtype: int64
```

```
# multiple row (similar to indexing)
branch_df3.loc[('cse',2019):('ece',2020):2]
```

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2021	5	6	0	0
ece	2020	9	10	0	0

```
branch_df3.iloc[0:5:3]
```

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
ece	2019	7	8	0	0

```
# Extracting column
branch_df3['delhi']['students']
```

```
cse  2019     2
     2020     4
     2021     6
ece  2019     8
     2020    10
     2021    12
Name: students, dtype: int64
```

```
# multiple columns
branch_df3.iloc[:,1:3]
```

		delhi	mumbai
		students	avg_package
cse	2019	2	0
	2020	4	0
	2021	6	0
ece	2019	8	0
	2020	10	0
	2021	12	0

```
# Extracting both
branch_df3.iloc[[0,4],[1,2]]
```

		delhi	mumbai
		students	avg_package
cse	2019	2	0
ece	2020	10	0

SORTING INDEX IN MULTIINDEX

Sorting the index in a MultiIndex DataFrame can be done using the sort_index() method.

```
# default
branch_df3.sort_index()
```

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
ece	2019	7	8	0	0
	2020	9	10	0	0
	2021	11	12	0	0

```
# both -> descending sorting
branch_df3.sort_index(ascending=False)
```

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2021	11	12	0	0
	2020	9	10	0	0
	2019	7	8	0	0
cse	2021	5	6	0	0
	2020	3	4	0	0
	2019	1	2	0	0

```
# Sorting on level (0) and level (1)
branch_df3.sort_index(ascending=[False,True])
```

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2019	7	8	0	0
	2020	9	10	0	0
	2021	11	12	0	0
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0

```
# sorting on specific level
branch_df3.sort_index(level=1, ascending=False)
```

		delhi		mumbai	
		avg_package	students	avg_package	students
ece	2021	11	12	0	0
cse	2021	5	6	0	0
ece	2020	9	10	0	0
cse	2020	3	4	0	0
ece	2019	7	8	0	0
cse	2019	1	2	0	0

```
branch_df1.unstack().stack().stack()
```

```
cse  2019  avg_package     1
           students        2
     2020  avg_package     3
           students        4
     2021  avg_package     5
           students        6
ece  2019  avg_package     7
           students        8
     2020  avg_package     9
           students       10
     2021  avg_package    11
           students       12
dtype: int64
```

```
# Example
branch_df2
```

	delhi		mumbai	
	avg_package	students	avg_package	students
2019	1	2	0	0
2020	3	4	0	0
2021	5	6	0	0

```
# Most-inner columns becomes row in stack() method
branch_df2.stack()
```

		delhi	mumbai
2019	avg_package	1	0
	students	2	0
2020	avg_package	3	0
	students	4	0
2021	avg_package	5	0
	students	6	0

```
# stacking one more level
branch_df2.stack().stack()
```

```
2019  avg_package  delhi     1
                   mumbai    0
      students     delhi     2
                   mumbai    0
2020  avg_package  delhi     3
                   mumbai    0
      students     delhi     4
                   mumbai    0
2021  avg_package  delhi     5
                   mumbai    0
      students     delhi     6
                   mumbai    0
dtype: int64
```

TRANSPOSE MULTIINDEX DATAFRAME

transpose() method:

- Used to transpose the rows and columns of a DataFrame.
- It switches the rows and columns, effectively converting the rows into columns and vice versa.
- Keep in mind that after transposing, the index becomes the columns, and the columns become the index.
- If your original DataFrame had column names, they would become the MultiIndex after transposing.

```
branch_df3
```

		delhi		mumbai	
		avg_package	students	avg_package	students
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
ece	2019	7	8	0	0
	2020	9	10	0	0
	2021	11	12	0	0

```
branch_df3.transpose()
```

		cse			ece		
		2019	2020	2021	2019	2020	2021
delhi	avg_package	1	3	5	7	9	11
	students	2	4	6	8	10	12
mumbai	avg_package	0	0	0	0	0	0
	students	0	0	0	0	0	0

swaplevel() method:

- The swaplevel() method in pandas is used to swap levels of a MultiIndex in a DataFrame.
- This can be particularly useful when you want to interchange the order of levels in a MultiIndex DataFrame.
- The method is applied to the DataFrame's index and can be useful for reorganizing or reshaping the data.
- Default is to swap the two innermost levels of the index.
- **Syntax:** DataFrame.swaplevel(i=-2, j=-1, axis=0)
- Swap levels i and j in a MultiIndex.
- Default is to swap the two innermost levels of the index.

Parameters:

i, j:	(int or str) Levels of the indices to be swapped. Can pass level name as string.

axis:	{0 or 'index', 1 or 'columns'}, default 0 The axis to swap levels on. 0 or 'index' for row-wise, 1 or 'columns' for column-wise

```
branch_df3.swaplevel()
```

		delhi		mumbai	
		avg_package	students	avg_package	students
2019	cse	1	2	0	0
2020	cse	3	4	0	0
2021	cse	5	6	0	0
2019	ece	7	8	0	0
2020	ece	9	10	0	0
2021	ece	11	12	0	0

```
# swaplevel with default and column
branch_df3.swaplevel(0,axis=1)
```

		avg_package	students	avg_package	students
		delhi	delhi	mumbai	mumbai
cse	2019	1	2	0	0
	2020	3	4	0	0
	2021	5	6	0	0
ece	2019	7	8	0	0
	2020	9	10	0	0
	2021	11	12	0	0

LONG FORMAT VS WIDE FORMAT

Long format: (tidy data):

- Long format is where, for each data point we have as many rows as the number of attributes and each row contains the value of a particular attribute for a given data point.
- A long format contains values that do repeat in the first column.
- Long format data common sources of obtain data is:
  Surveys and Questionnaires
  Time seires data
  Sensor data
  Clinical Trials and Medical Studies
- Long format:

Name	Attribute	Value
John	Height	160
John	Weight	67
wick	Height	182
wick	Weight	78

Wide format:

- Wide format is where we have a single row for every data point with multiple columns to hold the values of various attributes.
- A wide format contains values that do not repeat in the first column.
- Wide format data common sources of obtain data is:
  Government Databases
  CSV Files from Statistical Agencies
  Machine Learning Datasets
- Wide format:

Name	Height	Weight
John	160	67
wick	182	78

WHAT DIFFERENCE BETWEEN THEM?

Both are similar, but he choice between long and wide formats in data storage and analysis is often dependent on the problem statement, the nature of the data, and the specific analysis or tasks you plan to perform.

Long format:

Name	Attribute	Value
John	Height	160
John	Weight	67
wick	Height	182
wick	Weight	78

Wide format:

Name	Height	Weight
John	160	67
wick	182	78

melt() function:

- In pandas, melt() function was used to transform the dataset from a Wide format into a Long format.
- **Syntax:** pd.melt(frame, id_vars, value_vars, var_name, value_name='value', col_level)

Parameters:

Frame:	DataFrame
value_vars:	[tuple, list, or ndarray, optional] Column(s) to unpivot. If not specified, uses all columns that are not set as id_vars.
id_vars:	[tuple, list, or ndarray, optional] Column(s) to use as identifier variables.
var_name:	[scalar] Name to use for the 'variable' column. If None it uses frame.columns.name or 'variable'.
value_name:	[scalar, default 'value'] Name to use for the 'value' column.
col_level:	[int or string, optional] If columns are a MultiIndex then use this level to melt.

PIVOT TABLE

The pivot table takes simple column-wise data as input and group the entries into a two-dimenational table that provide a multidimensional summarization of the data.

```
import seaborn as sns
df = sns.load_dataset('tips')
df.head()
```

	total_bill	tip	sex	smoker	day	time	size
0	16.99	1.01	Female	No	Sun	Dinner	2
1	10.34	1.66	Male	No	Sun	Dinner	3
2	21.01	3.50	Male	No	Sun	Dinner	3
3	23.68	3.31	Male	No	Sun	Dinner	2
4	24.59	3.61	Female	No	Sun	Dinner	4

```
# according to gender what is total average bill
df.groupby('sex')[['total_bill']].mean()
```

	total_bill
sex	
Male	20.744076
Female	18.056897

```
new_df = df.groupby(['sex','smoker'])[['total_bill']]
new_df.mean().unstack()
```

	total_bill	
smoker	Yes	No
sex		
Male	22.284500	19.791237
Female	17.977879	18.105185

pivot_table() function:

- Used to create pivot tables from a DataFrame.
- It provides a flexible and powerful way to reshape and summarize data.
- **Syntax:** df.pivot_table(values=None, index=None, columns=None, aggfunc='mean', fill_value=None, margins=False, dropna=True, margins_name='All')

Parameters:

values	This is the column that you want to aggregate. You can specify multiple columns if needed
index	The column whose unique values will become the index of the pivot table
columns	The column whose unique values will become the columns of the pivot table
aggfunc	The aggregation function to apply. It could be 'mean', 'sum', 'count', 'min', 'max', etc. You can also pass a dictionary to apply different aggregation functions to different columns
fill_value	A scalar value to replace missing values.
margins	If True, it adds all row/column margins (subtotals)
dropna	If True, it excludes NA/null values

margins_name	Name of the row/column that will contain the totals when margins is True

```
df.pivot_table(index='sex',
               columns='smoker',
               values='total_bill')
```

smoker	Yes	No
sex		
Male	22.284500	19.791237
Female	17.977879	18.105185

```
# aggfunc parameter
df.pivot_table(index='sex',
               columns='smoker',
               values='total_bill',
               aggfunc='sum')
```

smoker	Yes	No
sex		
Male	1337.07	1919.75
Female	593.27	977.68

```
# columns together
df.pivot_table(index='sex', columns='smoker')
```

	size		tip		total_bill	
smoker	Yes	No	Yes	No	Yes	No
sex						
Male	2.500000	2.711340	3.051167	3.113402	22.284500	19.791237
Female	2.242424	2.592593	2.931515	2.773519	17.977879	18.105185

```
# Multidimentinal
df.pivot_table(index=['sex','smoker'],
               columns=['day','time'],
               values='total_bill')
```

	day	Thur		Fri		Sat	Sun
	time	Lunch	Dinner	Lunch	Dinner	Dinner	Dinner
sex	smoker						
Male	Yes	19.171000	NaN	11.386667	25.892	21.837778	26.141333
	No	18.486500	NaN	NaN	17.475	19.929063	20.403256
Female	Yes	19.218571	NaN	13.260000	12.200	20.266667	16.540000
	No	15.899167	18.78	15.980000	22.750	19.003846	20.824286

```
# mergins parameter
df.pivot_table(index='sex',
               columns='smoker',
               values='total_bill',
               aggfunc='sum',
              margins=True)
```

smoker	Yes	No	All
sex			
Male	1337.07	1919.75	3256.82
Female	593.27	977.68	1570.95
All	1930.34	2897.43	4827.77

PLOTTING GRAPHS USING PIVOT TABLE

```
df = pd.read_csv('DATASETS/S21/expense_data.csv')
#df.head()
```

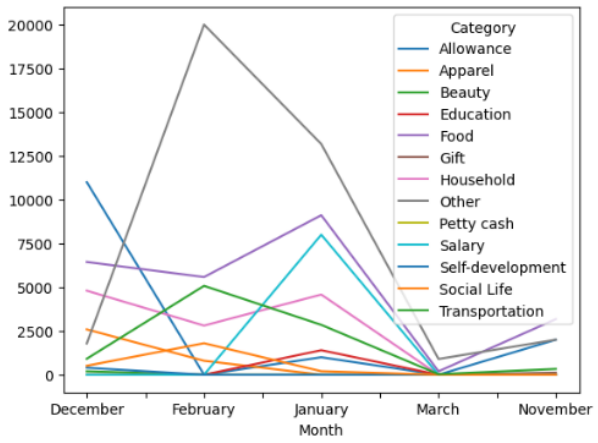
```
# Converting to date column type object to datatime64
df['Date'] = pd.to_datetime(df['Date'])
# add month column
df['Month'] = df['Date'].dt.month_name()
```

```
df.pivot_table(index='Month',columns='Category',
               values='INR',aggfunc='sum',fill_value=0)
```

Category	Allowance	Apparel	Beauty	Education	Food	Gift	Hous
Month							
December	11000	2590	196	0	6440.72	0	
February	0	798	0	0	5579.85	0	
January	1000	0	0	1400	9112.51	0	
March	0	0	0	0	195.00	0	
November	2000	0	0	0	3174.40	115	

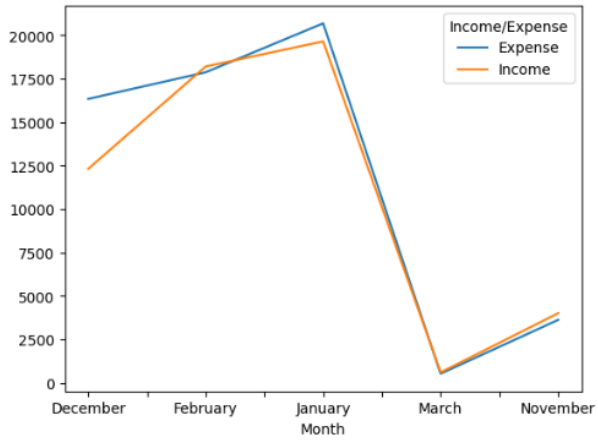
```
df.pivot_table(index='Month',columns='Category',
               values='INR',aggfunc='sum',fill_value=0
              ).plot()
```

```
<AxesSubplot: xlabel='Month'>
```




```
df.pivot_table(index='Month',columns='Income/Expense',
               values='INR',aggfunc='sum',fill_value=0
              ).plot()
```

```
<AxesSubplot: xlabel='Month'>
```



VECTORIZED STRING OPERATIONS

WHERE WE CAN APPLY THIS TOPIC?

Apply on textual dataset like movies description datasets, customer's reviews datasets, WhatsApp chat analysis review etc.

WHAT ARE VECTORIZED OPERATIONS?

- Here, a is a NumPy array with the elements [1, 2, 3, 4].
- The operation a * 4 is a vectorized operation. It multiplies each element of the array a by 4.
- The result is a new NumPy array where each element is the product of the corresponding element in the original array a and the scalar value is 4.

```
import numpy as np
import pandas as pd
# Example of vectorized operation
a = np.array([1,2,3,4])
a * 4
```

```
array([ 4,  8, 12, 16])
```

PROBLEMS IN VECTORIZED OPERATION IN VANILLA PYTHON !

vanilla python:

- It refers to the core, basic, or standard implementation of the Python programming language without any additional libraries or frameworks.
- It is the pure, unmodified form of Python, as defined by the Python Software Foundation.
- If we have None, missing values, and null values in datasets, in this scenario, Python functionality cannot handle.
- Vanilla Python may not be optimized for handling large datasets efficiently.

```
# try to apply vectories operation on string
s = ['cat','mat',None,'rat']
[i.startswith('c') for i in s]
```

```
---------------------------------------------------------------------------
AttributeError                            Traceback (most recent call last)
Cell In [96], line 3
      1 # try to apply vectories operation on string
      2 s = ['cat','mat',None,'rat']
----> 3 [i.startswith('c') for i in s]

Cell In [96], line 3, in <listcomp>(.0)
      1 # try to apply vectories operation on string
      2 s = ['cat','mat',None,'rat']
----> 3 [i.startswith('c') for i in s]

AttributeError: 'NoneType' object has no attribute 'startswith'
```

HOW PANDAS SOLVE THIS ISSUE?

- To solve this above issue is use .str accessor
- In pandas, the .str accessor is used to perform vectorized string operations on a Pandas Series containing strings.
- This accessor provides a collection of methods that allow you to manipulate strings efficiently without using explicit loops.
- The .str accessor simplifies and speeds up the process of working with string data in Pandas DataFrames, making it a powerful tool for data cleaning and manipulation.

```
# apply vecorized string operation
s = pd.Series(['cat','mat',None,'rat'])
# string accessor
s.str.startswith('c')
```

```
0     True
1    False
2     None
3    False
dtype: object
```

APPLY VECTORIZED STRING OPERATIONS IN TITANIC DATASET

```
df = pd.read_csv('DATASETS/S22/titanic.csv')
df.head(3)
```

	PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch
0	1	0	3	Braund, Mr. Owen Harris	male	22.0	1	0
1	2	1	1	Cumings, Mrs. John Bradley (Florence Briggs Th...	female	38.0	1	0
2	3	1	3	Heikkinen, Miss. Laina	female	26.0	0	0

```
# Choose 'Name' column for operations
df['Name']
```

```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
                           ...
886                              Montvila, Rev. Juozas
887                       Graham, Miss. Margaret Edith
888           Johnston, Miss. Catherine Helen "Carrie"
889                              Behr, Mr. Karl Howell
890                                Dooley, Mr. Patrick
Name: Name, Length: 891, dtype: object
```

COMMON .STR METHODS/FUNCTION

Nearly all Python's built-in string methods are mirrored by a Pandas vectorized string method.

.str.lower() / .str.upper() / .str.capitalize() / .str.title() / .str.len()

```
df['Name'].str.lower().tail(3)
```

```
888     johnston, miss. catherine helen "carrie"
889                        behr, mr. karl howell
890                          dooley, mr. patrick
Name: Name, dtype: object
```

```
df['Name'].str.upper().tail(3)
```

```
888     JOHNSTON, MISS. CATHERINE HELEN "CARRIE"
889                        BEHR, MR. KARL HOWELL
890                          DOOLEY, MR. PATRICK
Name: Name, dtype: object
```

```
df['Name'].str.capitalize().tail(3)
```

```
888     Johnston, miss. catherine helen "carrie"
889                        Behr, mr. karl howell
890                          Dooley, mr. patrick
Name: Name, dtype: object
```

```
df['Name'].str.title().tail(3)
```

```
888     Johnston, Miss. Catherine Helen "Carrie"
889                        Behr, Mr. Karl Howell
890                          Dooley, Mr. Patrick
Name: Name, dtype: object
```

```
df['Name'].str.len().head()
```

```
0    23
1    51
2    22
3    44
4    24
Name: Name, dtype: int64
```

```
# find longest name
df['Name'][df['Name'].str.len() == 82].values[0]
```

```
'Penasco y Castellana, Mrs. Victor de Satode (Maria Josefa Perez de Soto y Vallejo)'
```

.str.strip() method:

- The .str.strip() method removes leading and trailing whitespace from each string in the Series.
- For the element with None and empty string, it becomes NaN after stripping.
- Useful in nlp projects.

```
df['Name'].str.strip().tail()
```

```
886                              Montvila, Rev. Juozas
887                       Graham, Miss. Margaret Edith
888           Johnston, Miss. Catherine Helen "Carrie"
889                              Behr, Mr. Karl Howell
890                                Dooley, Mr. Patrick
Name: Name, dtype: object
```

.str.split() method

- The .str.split() method splits each string in the Series into a list of substrings.
- If you want to split the string based on a specific delimiter, you can provide that delimiter as an argument to .str.split(). For example, to split based on a comma, you can use s.str.split(',').
- **Syntax:** .str.split(pat=None, n=-1, expand=False)

Parameters:

n:	Specifies the maximum number of splits to perform. By default, it is set to -1, which means there is no limit to the number of splits.
Expand:	Controls whether to expand the result into a DataFrame. If expand is set to True, the result will be a DataFrame with one column per split.

	If False (the default), the result is returned as a Series of lists.

.str.get() method:

- The .str.get() method in pandas is used to get the element at a specified position for each string in a Pandas Series of strings.
- It is a vectorized string method, meaning it operates on each element of the Series without requiring explicit loops.

```
df['Name'].head()
```

```
0                              Braund, Mr. Owen Harris
1    Cumings, Mrs. John Bradley (Florence Briggs Th...
2                               Heikkinen, Miss. Laina
3         Futrelle, Mrs. Jacques Heath (Lily May Peel)
4                             Allen, Mr. William Henry
Name: Name, dtype: object
```

```
# Create lastname columns
df['Lastname'] = df['Name'].str.split(',').str.get(0)
df.head(3)
```

Parch	Ticket	Fare	Cabin	Embarked	Lastname	Firstname	Title
0	A/5 21171	7.2500	NaN	S	Braund	Mr. Owen Harris	Mr.
0	PC 17599	71.2833	C85	C	Cumings	Mrs. John Bradley (Florence Briggs Thayer)	Mrs.
0	STON/O2. 3101282	7.9250	NaN	S	Heikkinen	Miss. Laina	Miss.

```
df['Firstname'] = df['Name'].str.split(',').str.get(1)
df.head(2)
```

Sp	Parch	Ticket	Fare	Cabin	Embarked	Lastname	Firstname	Title
1	0	A/5 21171	7.2500	NaN	S	Braund	Mr. Owen Harris	Mr.
1	0	PC 17599	71.2833	C85	C	Cumings	Mrs. John Bradley (Florence Briggs Thayer)	Mrs.

.str.replace() method:

```
# Count titles
df['Title'].value_counts().head(10)
```

```
Mr.        517
Miss.      185
Mrs.       125
Master.     40
Dr.          7
Rev.         6
Major.       2
Col.         2
Don.         1
Mme.         1
Name: Title, dtype: int64
```

```
# replace the title 'Ms' and 'Mlle' with 'Miss'
df['Title'] = df['Title'].str.replace('Ms','Miss')
df['Title'] = df['Title'].str.replace('Mlle','Miss')
```

```
df['Title'].value_counts().head(10)
```

```
Mr.        517
Miss.      185
Mrs.       125
Master.     40
Dr.          7
Rev.         6
Major.       2
Col.         2
Don.         1
Mme.         1
Name: Title, dtype: int64
```

FILTERING IN VECTORIZED STRING OPERATIONS

In pandas, vectorized string operations can be combined with boolean indexing to filter data based on string conditions efficiently.

.str.statswith() / .str.endswith() / .str.isalpha() / .str.isdigit

```
# Find Firstname starts with 'A'
bool_df = df['Firstname'].str.startswith('A')
df[bool_df]['Firstname'].head()
```

```
13       Anders Johan
22       Anna "Annie"
35    Alexander Oskar
38      Augusta Maria
61             Amelie
Name: Firstname, dtype: object
```

```
# Find Firstname ends with 'A'
bool_df = df['Firstname'].str.endswith('A')
df[bool_df]['Firstname'].head()
```

```
64     Albert A
303      Nora A
Name: Firstname, dtype: object
```

```
# Find the name it include digits ?
bool_df = df['Firstname'].str.isdigit()
df[bool_df] # digit not found
```

PassengerId	Survived	Pclass	Name	Sex	Age	SibSp	Parch	Ticke

ADVANCED FILTERING WITH REGEX (with .str.contains() method)

.str.contains() method:

- The .str.contains() method in pandas is used to check if each element in a Pandas Series of strings contains a specified substring or matches a regular expression pattern.
- This method returns a boolean mask indicating whether the condition is met for each element in the Series.
- Syntax: .str.contains(pat, case=True, flags=0, na=nan, regex=True)

Parameters:

pat:	The substring or regular expression pattern to search for.
case:	If True, the matching is case-sensitive (default is True).
flags:	Additional flags for controlling the behavior of the regex (see the re module for options).
na:	The value to use for missing values (default is nan).
regex:	If True, treats the pat parameter as a regular expression (default is True).

```
# Find the john name
bool_df = df['Firstname'].str.contains('john',case=False)
df[bool_df]['Firstname'].head()
```

```
1           John Bradley (Florence Briggs Thayer)
41     William John Robert (Dorothy Ann Wonnacott)
45                                    William John
98                          John T (Ada Julia Bone)
112                                     David John
Name: Firstname, dtype: object
```

```
# Find Lastname with start and end char is vowel
pattern = '^[aeiouAEIOU].+[aeiouAEIOU]$'
bool_df = df['Lastname'].str.contains(pattern,case=False)
df[bool_df]['Lastname']
```

```
30          Uruchurtu
49     Arnold-Franchi
207          Albimona
210               Ali
353    Arnold-Franchi
493       Artagaveytia
518             Angle
784               Ali
840           Alhomaki
Name: Lastname, dtype: object
```

SLICING IN VECTORIZED STRING OPERATIONS

- In pandas, vectorized string operations allow you to perform slicing on each element of a Pandas Series of strings efficiently.
- You can use the .str accessor along with indexing or slicing to extract substrings based on position or conditions.

```
# basic slicing like pythons string
df['Name'].str[1:4].head()
```

```
0    rau
1    umi
2    eik
3    utr
4    lle
Name: Name, dtype: object
```

DATETIME

TIMESTAMP OBJECT

WHAT IS TIMESTAMP?

- Timestamp refers as particular moments in time (e.g., Oct 24th. 2022 at 7:00pm)
- Pandas library provide different datatype to store that datatype

WHAT IS TIMESTAMP OBJECT?

- Timestamp object is part of the pandas library and is particularly useful when working with time series data.
- Patient health metrics, stock price changes, weather records, economic indicators, servers, networks, sensors, and applications performance monitoring are examples of time-series data.
- We can also called vectorized datetime operations or vectorized datetime functions.

CREATING TIMESTAMP OBJECT

- Create a Timestamp object using the pd.Timestamp constructor.
- It can handle various input formats, including strings, datetime objects, or even numeric values representing timestamps.

NOTE: Always try to follow this date format: 'YYYY/MM/DD'

```
import numpy as np
import pandas as pd
# Creating a timestamp
pd.Timestamp('2023/10/18')
```

```
Timestamp('2023-10-18 00:00:00')
```

```
# diffrent variation in timestamp
pd.Timestamp('2023-10-18')
pd.Timestamp('2023, 10, 18')
```

```
Timestamp('2023-10-18 00:00:00')
```

```
# only year
pd.Timestamp('2023')
```

```
Timestamp('2023-01-01 00:00:00')
```

```
# use text
pd.Timestamp('18th oct 2023')
```

```
Timestamp('2023-10-18 00:00:00')
```

```
# Providing time as well
pd.Timestamp('2023, 10, 18, 8:12PM')
```

```
Timestamp('2023-10-18 20:12:00')
```

```
# creating timestamp using python datetime object
import datetime as dt
dt_object = dt.datetime(2023, 10, 18, 8, 18, 56)
t = pd.Timestamp(dt_object)
t
```

```
Timestamp('2023-10-18 08:18:56')
```

```
# fetching attributes like year,month,year etc
print(t.year)
print(t.month)
print(t.day)
print(t.hour)
print(t.minute)
print(t.second)
```

```
2023
10
18
8
18
56
```

(INTERVIEW QUESTION)

why separate objects to handle data and time when python already has datetime functionality?

- syntax wise datetime is very convenient
- But the performance takes a hit while working with huge data. List vs Numpy Array
- The weaknesses of Python's datetime format inspired the NumPy team to add a set of native time series data type to NumPy.
- The datetime64 dtype encodes dates as 64-bit integers, and thus allows arrays of dates to be represented very compactly.

```
import numpy as np
date = np.array('2015-07-04', dtype=np.datetime64)
date
```

```
array('2015-07-04', dtype='datetime64[D]')
```

```
# suporting vectorized operations
date + np.arange(6)
```

```
array(['2015-07-04', '2015-07-05', '2015-07-06', '2015-07-07',
       '2015-07-08', '2015-07-09'], dtype='datetime64[D]')
```

- Because of the uniform type in NumPy datetime64 arrays, this type of operation can be accomplished much more quickly than if we were working directly with Python's datetime objects, especially as arrays get large
- Pandas Timestamp object combines the ease-of-use of python datetime with the efficient storage and vectorized interface of numpy.datetime64
- From a group of these Timestamp objects, Pandas can construct a DatetimeIndex that can be used to index data in a Series or DataFrame

DATETIMEINDEX OBJECTS

- In simple words, A collection of pandas timestamp
- Pandas DatetimeIndex makes it easier to work with Date and Time data in our DataFrame.
- DatetimeIndex() can contain metadata related to date and timestamp and is a great way to deal with DateTime related data and do the calculations on data and time.
- **Syntax:** pd.DatetimeIndex(data, freq=_NoDefault.no_default, tz, normalize=False, closed, ambiguous='raise', dayfirst=False, yearfirst=False, dtype, copy=False, name)

Parameters:

data:	The data to be converted to datetime. It can be a list, array, or Series containing datetime-like objects, or a single datetime-like object.
freq:	The frequency of the datetime values if the data is not already a time series. It can be a string (e.g., 'D' for daily, 'H' for hourly) or a Timedelta object.
tz:	Timezone for the datetime values.
normalize:	If True, normalize the datetime values (set time to midnight).
closed:	Specify whether the interval is left-closed ('left'), right-closed ('right'), both closed ('both'), or neither closed ('neither').
ambiguoius:	How to handle daylight savings time ambiguities. Default is 'raise', but it can also be set to 'infer' or 'NaT'.
dayfirst:	If True, parse dates with the day first (e.g., '10/12/2023' is October 12, 2023).
yearfirst:	If True, parse dates with the year first (e.g., '2023-10-12' is October 12, 2023).
dtype:	The dtype of the datetime values.
copy:	If True, ensure that the input data is copied. Default is False.
name:	Name to be assigned to the resulting DatetimeIndex.

```
# collection of timestaime object
d = pd.DatetimeIndex(['2023-10-18','2023-10-19'])
print(d)
```

```
DatetimeIndex(['2023-10-18', '2023-10-19'], dtype='datetim
e64[ns]', freq=None)
```

```
# checking type
print(type(d))
print(type(d[0]))
```

```
<class 'pandas.core.indexes.datetimes.DatetimeIndex'>
<class 'pandas._libs.tslibs.timestamps.Timestamp'>
```

```
# using python datetime object
dates = [dt.datetime(2023,10,18),dt.datetime(2023,10,19)]
pd.DatetimeIndex(dates)
```

```
DatetimeIndex(['2023-10-18', '2023-10-19'], dtype='datetim
e64[ns]', freq=None)
```

```
# using pd.Timestamp
dates = [pd.Timestamp(2023,10,18),pd.Timestamp(2023,10,19)]
dt_index = pd.DatetimeIndex(dates)
dt_index
```

```
DatetimeIndex(['2023-10-18', '2023-10-19'], dtype='datetim
e64[ns]', freq=None)
```

```
# using DatetimeIndex as Series
pd.Series([1,2], index=dt_index)
```

```
2023-10-18    1
2023-10-19    2
dtype: int64
```

date_range() function

- The pd.date_range() function in Pandas is used to generate a fixed-frequency DatetimeIndex.
- It's a convenient method for creating date sequences for time-based data, such as time series data.
- The function allows you to specify the start date, end date, and frequency of the date range.
- **Syntax:** pd.date_range(start, end, periods, freq, tz=None, normalize=False, name, closed, **kwargs)

Parameters:

start:	The start date of the range.
end:	The end date of the range.
periods:	The number of periods (int) to generate.
freq:	Frequency of the resulting date sequence. This can be a string representing a frequency alias (e.g., 'D' for day, 'H' for hour), a Timedelta object, or a custom frequency string.
tz:	Timezone for the datetime values.
normalize:	If True, normalize the datetime values (set time to midnight).
name:	Name to be assigned to the resulting DatetimeIndex.
closed:	Specify whether the interval is left-closed ('left'), right-closed ('right'), both closed ('both'), or neither closed ('neither').
**kwargs:	Additional keyword arguments that are passed to the underlying DatetimeIndex constructor.

```
# generate daily dates in given range
pd.date_range(start='2023/10/18',end='2023/10/21')
```

```
DatetimeIndex(['2023-10-18', '2023-10-19', '2023-10-20', '20
23-10-21'], dtype='datetime64[ns]', freq='D')
```

```
# alternative dates in given range
pd.date_range(start='2023/10/18',
              end='2023/10/25', freq='2D')
```

```
DatetimeIndex(['2023-10-18', '2023-10-20', '2023-10-22', '20
23-10-24'], dtype='datetime64[ns]', freq='2D')
```

```
# B --> business days
pd.date_range(start='2023/10/18',
              end='2023/10/25',freq='B')
```

```
DatetimeIndex(['2023-10-18', '2023-10-19', '2023-10-20', '20
23-10-23',
               '2023-10-24', '2023-10-25'],
              dtype='datetime64[ns]', freq='B')
```

```
# W --> One week per days
pd.date_range(start='2023/10/18',
              end='2023/10/28',freq='W')
```

```
DatetimeIndex(['2023-10-22'], dtype='datetime64[ns]', freq
='W-SUN')
```

```
# M -> Month end
pd.date_range(start='2023/10/18',
              end='2023/11/18',freq='M')
```

```
DatetimeIndex(['2023-10-31'], dtype='datetime64[ns]', freq
='M')
```

```
# MS -> Starting month
pd.date_range(start='2023/10/18',
              end='2023/11/18',freq='MS')
```

```
DatetimeIndex(['2023-11-01'], dtype='datetime64[ns]', freq
='MS')
```

```
# H -> Hourly data(factor)
pd.date_range(start='2023/10/18',
              end='2023/10/28',freq='12H')
```

```
DatetimeIndex(['2023-10-18 00:00:00', '2023-10-18 12:00:00',
               '2023-10-19 00:00:00', '2023-10-19 12:00:00',
               '2023-10-20 00:00:00', '2023-10-20 12:00:00',
               '2023-10-21 00:00:00', '2023-10-21 12:00:00',
               '2023-10-22 00:00:00', '2023-10-22 12:00:00',
               '2023-10-23 00:00:00', '2023-10-23 12:00:00',
               '2023-10-24 00:00:00', '2023-10-24 12:00:00',
               '2023-10-25 00:00:00', '2023-10-25 12:00:00',
               '2023-10-26 00:00:00', '2023-10-26 12:00:00',
               '2023-10-27 00:00:00', '2023-10-27 12:00:00',
               '2023-10-28 00:00:00'],
              dtype='datetime64[ns]', freq='12H')
```

```
# A -> Year end
pd.date_range(start='2023/10/18',
              end='2025/11/28',freq='A')
```

```
DatetimeIndex(['2023-12-31', '2024-12-31'], dtype='dateti
me64[ns]', freq='A-DEC')
```

```
# using periods(number of results)
pd.date_range(start='2023/10/18',periods=25,freq='M')
```

```
DatetimeIndex(['2023-10-31', '2023-11-30', '2023-12-31',
'2024-01-31',
               '2024-02-29', '2024-03-31', '2024-04-30',
'2024-05-31',
               '2024-06-30', '2024-07-31', '2024-08-31',
'2024-09-30',
               '2024-10-31', '2024-11-30', '2024-12-31',
'2025-01-31',
               '2025-02-28', '2025-03-31', '2025-04-30',
'2025-05-31',
               '2025-06-30', '2025-07-31', '2025-08-31',
'2025-09-30',
               '2025-10-31'],
              dtype='datetime64[ns]', freq='M')
```

to_datetme() function:

- The pd.to_datetime() function in Pandas is used to convert an object to a datetime.
- It can be used to convert a wide variety of input types, including strings, integers, floats, and other datetime-like objects, into Pandas datetime objects, such as Timestamp or DatetimeIndex.
- In simple words, Converts an existing objects to pandas timestamp/datetimeindex object
- **Syntax:** pd.to_datetime(arg, errors='raise', format, dayfirst=False, yearfirst=False, utc, format2, exact=True, unit, infer_datetime_format=False, origin='unix', cache=False)

Parameters:

arg:	The object to be converted to a datetime. It can be a single value or an iterables (e.g., list, array, or Series).
errors:	How to handle parsing errors. It can be set to 'raise' (default), 'coerce' (to force errors to NaT), or 'ignore' (to skip errors).

format:	A format string to specify the exact format of the input data.
dayfirst:	If True, parse dates with the day first (e.g., '10/12/2023' is October 12, 2023).
yearfirst:	If True, parse dates with the year first (e.g., '2023-10-12' is October 12, 2023).
utc:	If True, return UTC datetime objects.
unit:	The unit of the input data if the input is numeric (e.g., 's' for seconds or 'ns' for nanoseconds).
Infer_datetime_format:	If True, infer the datetime format of the input data (can improve parsing performance).
origin:	A reference date for numeric time data (default is 'unix' for Unix timestamps).
cache:	Whether to cache the datetime conversion results for performance improvement.

```
# simple example
s = pd.Series(['2015-07-04','2015-08-05','2015-09-06'])
# Series dtype is string
print(s)
```

```
0    2015-07-04
1    2015-08-05
2    2015-09-06
dtype: object
```

```
# convert into datatime object
d = pd.to_datetime(s)
print(d)
# access of year, month etc.
print(d.dt.year)
print(d.dt.month)
```

```
0   2015-07-04
1   2015-08-05
2   2015-09-06
dtype: datetime64[ns]
0    2015
1    2015
2    2015
dtype: int64
0    7
1    8
2    9
dtype: int64
```

```
# if date format is invalid and try to convert
s = pd.Series(['2015-07-04','2015-07-43','2015-17-06'])
pd.to_datetime(s)
```

```
ParserError: day is out of range for month: 2015-07-43
present at position 1
```

```
# Handling parsing errors using the errors parameter:
s = pd.Series(['2015-07-04','2015-07-43','2015-17-06'])
pd.to_datetime(s,errors='coerce')
```

```
0   2015-07-04
1          NaT
2          NaT
dtype: datetime64[ns]
```

```
df = pd.read_csv('DATASETS/S22/expense_data.csv')
df.head()
```

	Date	Account	Category	Subcategory	Note	INR	Incom
0	3/2/2022 10:11	CUB - online payment	Food	NaN	Brownie	50.0	
1	3/2/2022 10:11	CUB - online payment	Other	NaN	To lended people	300.0	
2	3/1/2022 19:50	CUB - online payment	Food	NaN	Dinner	78.0	
3	3/1/2022 18:56	CUB - online payment	Transportation	NaN	Metro	30.0	
4	3/1/2022 18:22	CUB - online payment	Food	NaN	Snacks	67.0	

```
df['Date'].info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 277 entries, 0 to 276
Series name: Date
Non-Null Count  Dtype
--------------  -----
277 non-null    object
dtypes: object(1)
memory usage: 2.3+ KB
```

```
# convert Date column dtype 'object' into 'datatime64'
df['Date'] = pd.to_datetime(df['Date'])
df['Date'].info()
```

```
<class 'pandas.core.series.Series'>
RangeIndex: 277 entries, 0 to 276
Series name: Date
Non-Null Count  Dtype
--------------  -----
277 non-null    datetime64[ns]
dtypes: datetime64[ns](1)
memory usage: 2.3 KB
```

.dt accsessor:

- In Pandas, the .dt accessor is used to access the datetime components of a Series or DataFrame.
- The .dt accessor provides a convenient way to work with date and time components like year, month, day, hour, minute, second, etc.

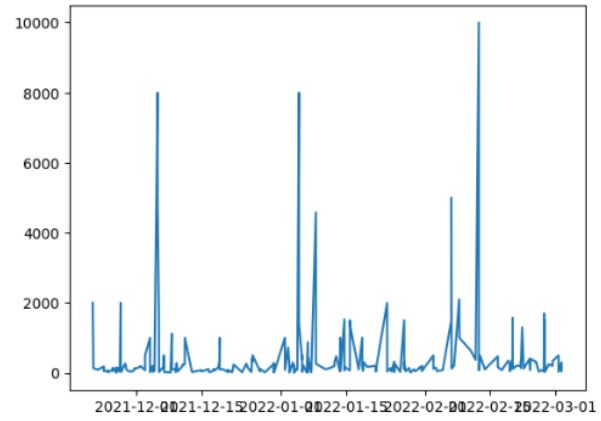
```
print(df['Date'].dt.year.head(3))
print(df['Date'].dt.month.head(3))
print(df['Date'].dt.day.head(3))
print(df['Date'].dt.day_name().head(3))
print(df['Date'].dt.month_name().head(3))
print(df['Date'].dt.is_month_end.head(3))
print(df['Date'].dt.is_quarter_end.head(3))
```

```
0    2022
1    2022
2    2022
Name: Date, dtype: int64
0    3
1    3
2    3
Name: Date, dtype: int64
0    2
1    2
2    1
Name: Date, dtype: int64
0    Wednesday
1    Wednesday
2      Tuesday
Name: Date, dtype: object
0    March
1    March
2    March
Name: Date, dtype: object
0    False
1    False
2    False
Name: Date, dtype: bool
0    False
1    False
2    False
Name: Date, dtype: bool
```

PLOTTING GRAPHS

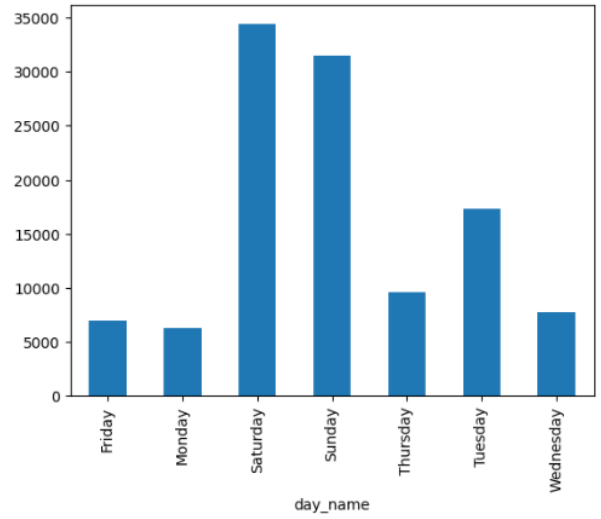
```
import matplotlib.pyplot as plt
plt.plot(df['Date'],df['INR'])
```

```
[<matplotlib.lines.Line2D at 0x25f4bc09600>]
```



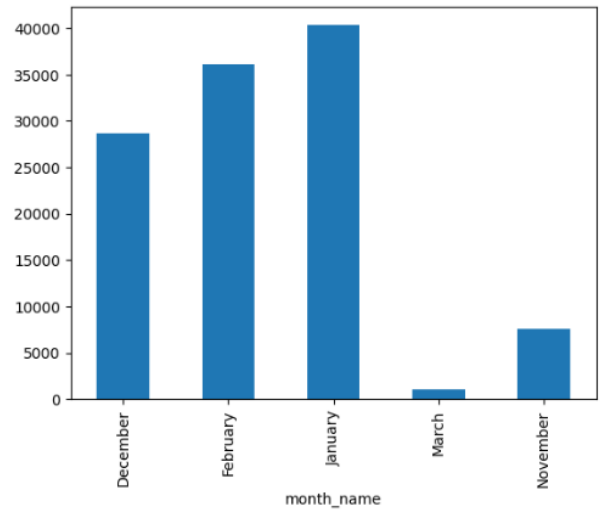

```
# daywise spending
df['day_name'] = df['Date'].dt.day_name()
df.groupby('day_name')['INR'].sum().plot(kind='bar')
```

```
<AxesSubplot: xlabel='day_name'>
```




```
# monthly spending
df['month_name'] = df['Date'].dt.month_name()
df.groupby('month_name')['INR'].sum().plot(kind='bar')
```

```
<AxesSubplot: xlabel='month_name'>
```



TIMEDELTA OBJECT (PANDAS)

- Represents a duration, the difference between two dates or times.
- In pandas, a Timedelta object represents a duration, which is the difference between two dates, times, or a combination of both.
- It is useful to work with time-based data and perform various time-related operations.
- We can create a Timedelta object using the pd.Timedelta constructor and provide the duration as an argument, which can be expressed in various units such as days, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds.

```
import pandas as pd
# Create Timedelta object using Timestamp object
td1 = pd.Timestamp('20th Oct 2023')
td2 = pd.Timestamp('20th Nov 2023')

# diff_td is Timedelta object
diff_td = td2 - td1
print(diff_td)
```

```
31 days 00:00:00
```

pd.Timedelta(value, unit):

- The pd.Timedelta() function in pandas is used to create a Timedelta object, which represents a duration or time difference.
- We can specify the duration as arguments to this function, using various units such as days, hours, minutes, seconds, milliseconds, microseconds, and nanoseconds.
- **Syntax:** pd.Timedelta(value, unit)

Parameters:

value:	The numerical value that represents the duration or time difference.
unit:	A string that specifies the unit of time. This can be one of the following: 'D' or 'days' for days 'H' or 'hours' for hours 'T' or 'minutes' for minutes 'S' or 'seconds' for seconds 'L' or 'milliseconds' for milliseconds 'U' or 'microseconds' for microseconds 'N' or 'nanoseconds' for nanoseconds

NOTE:
Timedelta is the pandas equivalent of python's datetime.timedelta and is interchangeable with it in most cases.

```
# standalone creation using pd.Timedelta()
pd.Timedelta(days=2,hours=10,minutes=35)
```

```
Timedelta('2 days 10:35:00')
```

```
# arithmetic operations
td = pd.Timedelta(days=10,hours=2)
new_date = pd.Timestamp('2023/08/20') + td
print(new_date)
```

```
2023-08-30 02:00:00
```

```
dates = pd.date_range(start='2023/10/18',
                      end='2023/10/25', freq='D')
td = pd.Timedelta(days=2,hours=10,minutes=35)
print(dates + td)
```

```
DatetimeIndex(['2023-10-20 10:35:00', '2023-10-21 1
0:35:00',
               '2023-10-22 10:35:00', '2023-10-23 1
0:35:00',
               '2023-10-24 10:35:00', '2023-10-25 1
0:35:00',
               '2023-10-26 10:35:00', '2023-10-27 1
0:35:00'],
              dtype='datetime64[ns]', freq='D')
```

```
# real life example (90's delivery dataset)
df = pd.read_csv('deliveries.csv')
df.head()
```

	order_date	delivery_date
0	5/24/98	2/5/99
1	4/22/92	3/6/98
2	2/10/91	8/26/92
3	7/21/92	11/20/97
4	9/2/93	6/10/98

```
# Calculating avg time period of delivery
df['order_date'] = pd.to_datetime(df['order_date'])
df['delivery_date'] = pd.to_datetime(df['delivery_date'])
# Timedelta
df['delivery_time'] = df['delivery_date'] - df['order_date']
df['delivery_time'].mean()
```

```
Timedelta('1217 days 22:53:53.532934128')
```

DateOffset() method:

- In Pandas, you can use the dateOffset class to represent various date offsets or date-related calculations. Date offsets allow you to perform operations like adding or subtracting a specific number of days, months, years, hours, minutes, or other time units from a date or time.
- The DateOffset class is part of the Pandas library and is found in the pandas.tseries.offsets module. You can use it to create date offsets and apply them to date or datetime objects.
- Here are some common date offsets in Pandas:

Day:	Represents a day offset.
BusinessDay:	Represents a business day (excluding weekends) offset.
Week:	Represents a week offset.
MonthEnd:	Represents the last day of a month.
YearEnd:	Represents the last day of a year.
Hour:	Represents an hour offset.
Minute:	Represents a minute offset.

asfreq() method:

- (Controlling the frequency of time series data)
- Used to resample time series data.
- Resampling is the process of changing the frequency of the data in a time series, which means converting it from one frequency (e.g., daily) to another frequency (e.g., monthly or quarterly).
- Particularly useful for time series data that's indexed with a DatetimeIndex or PeriodIndex.
- It allows you to specify the desired frequency (or frequency rule) to which you want to resample your data.
- It's a common tool for time series data analysis and manipulation.
- **Syntax:** DataFrame.asfreq(freq, method, how, normalize=False)

Parameters:

freq:	The frequency to which you want to resample the data, specified as a string (e.g., 'D' for daily, 'M' for monthly).
method:	This is an optional parameter that specifies how to handle missing data when resampling. It can take values like 'pad' or 'ffill' (forward fill), 'bfill' (backward fill), 'nearest', or None. This parameter determines how missing values are filled.
how:	This is an optional parameter, and it's mainly used for PeriodIndex. It allows you to specify whether the resampling should be done at the start or end of the period. It can take values like 'start' or 'end'.
normalize:	This is an optional parameter that, when set to True, ensures that the resulting DatetimeIndex is in a normalized form. For example, when resampling to a monthly frequency, if normalize is True, the resulting dates will always have day 1.

```
import pandas as pd
import numpy as np
# Create a sample time series
date_rng = pd.date_range(start='2023-01-01',
                         end='2023-01-10', freq='D')
data = np.arange(len(date_rng))
df = pd.DataFrame({'Date': date_rng, 'Data': data})
# Resample the data to a weekly frequency
weekly_data = df.set_index('Date').asfreq('W')
print(weekly_data)
```

```
            Data
Date
2023-01-01     0
2023-01-08     7
```