Stuti Mahajan

Numpy: [Numerical Python]

- It is a powerful opensource library used for numerical computing in python. It is a general purpose array processing package. It provides a high-performance multidimensional array object & tools for working with these arrays.

- Array - Data structures that stores values of same data type.

	List	Array
1.	Can store elements of different datatypes.	Store the element of same datatype only.
2.	Less memory-efficient	More memory efficient.
3.	Built in python data structure	Requires external module like array or numpy.
4.	Supports indexing & slicing.	Supports indexing & slicing (with type restrictions)
5.	Best for general-purpose collections with mixed types.	Best for numerical or scientific computing.
6.	Slower for numerical computation due to type-checking.	faster for numerical computation due to optimized operation.

- import numpy as np

- mylist = [1, 2, 3, 4]
  arr = np.array(mylist)

- MyArray = np.array([1, 2.5, 3])

array can be created from python lists, tuples or from any sequence like objects using np.array().

- type(arr) or type(MyArray)
  numpy.ndarray type is array.

* Using functions:

1. array = np.~~a~~arange(1, 10, 2)
   print(array) # [1, 3, 5, 7, 9]

np.arange(start, stop, step) creates range of numbers. If only start stop → all el except stop.

2. array = np.linspace(0, 1, 5)
   print(array) # [0, 0.25, 0.5, 0.75, 1]

np.linspace(start, stop, num) generates evenly spaced numbers b/w start & stop.

3.
```
array = np.zeros((2, 3))
print(array)    # [[0, 0, 0]
                   [0, 0, 0]]
```
np.zeros(shape) creates an array filled with 0

4.
```
array = np.ones((3, 2))
print(array)    # [[1, 1]
                   [1, 1]
                   [1, 1]]
```
np.ones(shape) creates an array filled with 1's

5.
```
array = np.empty((2,2))
print(array)
```
np.empty(shape) creates an array with random values

6.
```
array = np.full((2, 3), 7)
print(array)    # [[7, 7, 7]
                   [7, 7, 7]]
```
np.full((shape) fill_value)) creates an array filled with specified value.

★ Array datatype:

- common datatypes: int, float, bool, str, ...
- Numpy automatically assigns a datatype but it can also be explicitly specified.

```
array = np.array([1,2,3])
print(array.dtype)   #int64
```

BUT if we specify

```
                                dtype=
array = np.array([1, 2, 3], 'float')
print(array.dtype)  #float64
```

* Multidimensional Array:

-
  ```
  list1 = [1, 2, 3, 4]
  list2 = [5, 6, 7, 8]
  arr = np.array([list1, list2])
  print(arr)   #([[1  2  3  4]
                  [5  6  7  8]])
  ```

optional.

```
print (arr.shape)    # (2,4)
print (arr.size)     # 8
```

-
  ```
  array3D = np.array([[[1,2],[3,4]], [[5,6],
  [7,8]]])
  print(array3D)  # [[[1 2]
  array3D.shape        [3 4]]
   # (2,2,2)
  array3D.size #8     [[5 6]
                       [7 8]]]
  ```

- shape returns the dimensions of an array
- size returns total no. of element in array.

```
arr.ndim   # 2
array3D.ndim  #3
```

- ndim returns no. of dimensions in array.

• for previous arr :
Shape was (2, 4)
means 2 x 4 = 8 elements.

arr. reshape (shape) returns the elements present in another shape. But, the pdt of dimensions that you provide should be always equal to no. of elements present.
eg:

```
arr. reshape (4, 2)  # ([[1  2]
                         [3  4]
                         [5 6]
                         [7 8]])

arr . reshape (1, 8)
([[ 1  2  3  4  5  6  7  8]])

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

* Accessing array elements :

(Indexing)

- array = np.array([1, 2, 3, 4, 5])
  print (array[3]) # 4
  print (array[[0, 2, 4]]) # [1 3 5]
- array = np.array([[1, 2, 3], [4, 5, 6]])
  array[0, 2] # 3

$$\overset{\uparrow}{R}\ \overset{\uparrow}{C}$$

$$* \begin{array}{cccc} 0C & 1C & 2C & \\ \downarrow & \downarrow & \downarrow & \\ ([[1 & 2 & 3] & \leftarrow 0R \\ [4 & 5 & 6]]) & & & \leftarrow 1R \end{array}$$

- array = np.array([[1, 2, 3, 4, 5], [6, 7, 8, 9, 10], [11, 12, 13, 14, 15]])
  array[:, :] → gives all elements.

```
# ([[1  2  3  4  5],
    [6  7  8  9 10],
    [11 12 13 14 15]])
```

array[:, 3:] → from 3rd column.
  ↳ Not specified so all rows.

```
# ([[4  5 ],
    [9  10 ],
    [14 15 ]])
```

array[1:, 3:]

```
# ([[9   10]
    [14  15]])
```

- Boolean Idxing:
  print (array[array > 3]) #[4 5]

  print (array[array % 2 == 0])
  # [2 4 6]

```
arr = np.array([1, 2, 3, 4, 5, 6, 7, 8])
arr  # [1 2 3 4 5 6 7 8]
arr[3:] = 100
arr # [1 2 3 100 100 100 100 100]
```

The values are set to 100 from index 3.

no space so F hoga False nai

```
arr < 2 #([True False False F F F F F])
arr * 2  #([2 4 6 200 200 200 200 200])
arr / 2  #([0.5 1 1.5 50 50 50 50 50])
```

→ To specify just the element:

```
arr [arr<2]  # ([1])
```

* **Basic Arithmatic Operations**: +, -, /, *, **

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

print (array 1 + array 2) # [5 7 9]
print (array 1 - array 2) # [-3 -3 -3]
print (array 1 * array 2) # [4 10 18]
print (array 1 ** 2 ) # [1 4 9]
```

↑ power

→ arcsin → Inverse

```
print (np.sin(array1)) Trignometric
# [0.841 0.909 0.141]
print (np.log (array1)) logarithmic
# [0 0.6931 1.0986]
print (np.exp (array 1)) Exponential  e^x
# [2.7182 7.3890 20.0855]
```

M T W T F S S
Page No.:
Date:
YOUVA

★ array = np.array([1, 2, 3])
scalar = 10
print(array + scalar) # [11 12 13]

array_2D = np.array([[1, 2, 3], [4, 5, 6]])
print(array_2D + array)
# [[2 4 6]
   [5 7 9]]

★ **Aggregating functions:** sum(), mean(), median(), std() (sd), min(), max(), var()

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

In this format too (var also)

```
print(array.sum())  # 15
print(array.mean()) # 3
print(np.median(array)) # 3.0
print(array.std()) # 1.4142135623730951
print(array.min()) # 1
print(array.max()) # 5
```

★ arr = np.array([1 5 3 2 7 9 4])
print(np.sort(arr)) # [1 2 3 4 5 7 9]

sort() sorts the array in ascending order.

★ print(np.argmax(arr)) # 5
print(np.argmin(arr)) # 0

argmax () returns the index of maximum value.
argmin () returns the index of minimum value.
If duplicate in both, first appearance is considered.

★
```
arr = np. array ([[1, 2, 3], [4, 5, 6]])
print (array. flatten ()) # [1 2 3 4 5 6]
```
$\underbrace{\text{flatten ()}}_{\downarrow\ \text{ravel ()}}$

flatten returns a copy of the array flattened into 1D, Ravel returns a flattened view of array (does not create a copy).

★ Transposing Array (.T):

```
print (arr. T)  # [[1  4]
                   [2  5]
                   [3  6]]
```

★ np. concatenate (joinig the array):

```
arr1 = np.array ([1, 2, 3])
arr2 = np. array ([4, 5, 6])
print (np. concatenate ((arr1, arr2)))
 # [1 2 3 4 5 6]
```

★ np.split (splitting array):

⇒ array= np. array ([1, 2, 3, 4, 5, 6])
print (np. split (arr, 3))
↖ into how many sections.

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

* degree = np. array ([0, 90, 180])
radians = np. deg2rad (degrees)
print (radians) # [0 1.5707 3.141592]
* np. deg2rad - convert degree to radians
np. rad2deg - convert radian to degree.

* generating random numbers:

a. Uniform Distribution: (np. random. rand())
print (np. random. rand ()) # (0.45)
print (np. random. rand (3,2)) #[[_ _]
[_ _]
[_ _]]

• generates random no. b/w 0 to 1.

b. np. random . randn ()
generates random no. with mean 0 & sd of 1. (Normal / gaussian distribution)

c. Random Integer (np. random. randint()).
→ Random integer b/w 1 - 9
print (np. random. randint (1, 10))
print (np. random. randint (1, 10, (2,3))
→ 2X3 array of random integer from 1-9
↑ low ↳ inclusive
↑ high ↳ exclusive
↑ size.

- Data manipulation
- Data Cleaning
- Data Transformation

Stuti Mahajan

Pandas: import pandas as pd

- It is a opensource python library providing high performance, easy to use data structures & data analysis tools for handling structured data. It offers powerful capabilities to work with tabular data, including operations like cleaning, transformations & analysis. (EDA)

- Data Frame: (more than 1 row/column)
→ 2D ~~[illegible]~~ tabular data structure in pandas with labelled rows & columns.
→ Supports heterogeneous datatypes
→ Rows & columns can be accessed & manipulated.

```
df = pd.dataframe(data)
print(df)
```

- Series: (1 column/ 1 row)
→ 1D array in pandas capable of holding any data type, with labelled indices.
→ Only homogeneous datatype.

```
S = pd.Series([10, 20, 30], index=['a', 'b'
'c'])
print(s) #
```

Index	Value
a	10
b	20
c	30

* To create own data,

1 eg:

df = pd. DataFrames (np. arange (0, 20) . resh-
ape (5, 4) , indix = ['R1', 'R2', 'R3', 'R4', 'R5'
], columns = ["C1", "C2", "C3", "C4"])

To load from somewhere:

- df = pd. read - csv ('data . csv')
- df = pd. read - excel ('data. xlsx')
- file-path = r' C:\ - - - - . ___'

  df = pd. read-file (file - path)

  ↳ excel / csv

→ if we want to specify a sheet:

df = pd. read - excel ('data . xlsx', sheet_
name = 'Sheet 1')

* Accessing the elements:

1. .loc

df. loc['R1'] → type (df. loc['R1'])

bcoz only 1 row ← pandas. core. series. Series ↵

2. .iloc

df. iloc [: , :]

- df. iloc[0: 3, 0: 2]

Inclusive ↑ (0), exclusive ↑ (3, 2)

→ type (df. iloc [0:3 , 0:2])

more than 1 row/column ← pands. core. frame . Dataframe

M T W T F S S
Page No.:
Date:
YOUVA

∘★ Pandas:
1. Handles large datasets
2. Data manipulation
3. Data analysis
4. Data cleaning
5. Data joining
6. Data structure : Series , Dataframe
7. Easier to handle tabular data.
8. Supports labelled indexing whereas numpy requires positional indexing.

★ Convert dataframes into arrays:

```
df.iloc[: ,1:].values

# ([[ 1, 2, 3],
    [ 5, 6, 7],
    [ 9, 10, 11],
    [ 13, 14, 15],
    [ 17, 18, 19]])
```

.values converting all values of data into an array.

```
∘ df.iloc[: ,1:].values.shape
# (5, 3)
```

★ df.isnull().sum()
To check total null values in each column.

```
# C1 0
  C2 0
  C3 0
  C4 0
```

M T W T F S S
Page No.:
Date:
YOUVA

* just for null values:

print (df. ~~isull~~ isnull ())
#gives O/P in tabular form with true false → True where null
False where not null.

- df['C1']. value_counts ()
#gives all element of C1 with no. of times they appes in C1.

- df ['C1']. unique()
#will give O/P in array form with only the elements of C1, if duplicate, will show only once. → Uniqueness.

- df [['column 3', 'column 4']]
- df ['column 3']
→ also the ways to access elements

* Accessing data in series:

- print (S [1]) by position
- print (s_dict ['b']) by label

- print (s[1:3]) position based slicing.

- S1 = pd. series ([1, 2, 3])
S2 = pd. series ([4, 5, 6])
print (s1 + s2)
← (-, *, /, **)

* with null value handling we can use: axis, inplace.

* Handling Missing data in series

- S = pd. Series ([1, None, 3])

1. print (s. isnull()) # True where null
2. print (s. fillna(0)) # 0 where null
3. print (S. dropna()) # drops where null.
4. print (S. notnull()) # True where not null.

- print (df. columns) # column names
- print (df. dtypes) # datatypes of column.

* Boolean Indexing in DF's:

print (df[df['A'] > 1])
Rows where column A is greater than 1.

* Handling missing data in DF:

print (df. isnull()) # True where null
print (df. dropna()) # removes null values
print (df. fillna ({ — }) # replaces Nan with a specified value.

* Basic Operations:

1. Viewing data →

df. head () # first 5 entries if not speci
df. tail () # last 5 entries if not specified
df. info () # information: no. of column, column label, —‖— dtype etc.

2. Statistical Summary →
df. describe () # description of data: numerical data of each column, count of non-empty val etc.

To show duplicate rows.
df[df.duplicated()]

3. Removing duplicates:
df.drop-duplicates()

4. To select specific element:
df['a'][1]
$\hookrightarrow$ column $\longrightarrow$ element index

returns element of column a with index 1.

type(df['a'][1]) #returns type of that element.

$\hookrightarrow$ • df.query('col1>10 and col2<20')

• df[df['name'].str.contains('John')]
#selects rows where name contains john.

★ df1 = pd.Dataframe({'A':[1,2,3]}, index= ['a', 'b', 'c'])
df2 = pd.Dataframe({'A':[4,5,6]}, index= ['b', 'c', 'd'])
df3 = df1 + df2 # give addition of b & c and Nan for a & d.

★ df['A'] = df['A'] +5 # adds 5 to all ~~column~~ values in column A.

★ df.isnull().sum() # no. of missing values per column.

M T W T F S S
Page No.:
Date:
YOUVA

★ To replace specific values:

.replace()

- df.replace(10, 100, inplace = True) #replace 10 with 100.

- df.replace(10: 100, 20: 200, inplace = True) #replace multiple value 10 by 100 & 20 by 200.

★ To change datatype:

.astype()

- df['col1'] = df['col1'].astype(int)
  'category'
  float ... etc.

- df = df.astype({'col1': 'float64', 'col2': 'category'})

★ renaming column:

.rename()

- df.rename(columns = {'old-name': 'new-name'}, inplace = True)

- df.rename(columns = {'old_name1': 'new-name1' 'old_name2': 'new_name2'}, inplace = True)

* To rename all column [df.column]
  df.columns = ['col1', 'col2', 'col3']

* **Aggregation function:**
.sum() , .mean(), .count() , .min(), .max()

1. grouped['column'].sum() #sum of values in each grp.
   .mean()
   .min()
   .max()
   .count() #count of null values in each grp.

• **Grouping:**
grouped = df.groupby(col-name) #grpby single col
([‘col1’, ‘col2’]) #grp by multiple column.

• To apply multiple aggregation function:

~~grouple~~
grouped['col-name'].agg(['sum', 'mean', 'std'])

* **Concatenation (concat)**

• Used to combine multiple DFs along a particular axis (rows / column)
• default axis = 0 [rows]

```
df1 = pd.Dataframe({'A': [1,2], 'B': [3,4]
})
df2 = pd.DataFrame({'A': [5,6], 'B': [7,8]})

result = pd.concat([df1, df2])
print(result)

#      A    B
  0    1    3    0
  1    2    4    1
  0    5    7    2
  1    6    8    3
```

concat keeps the index. To reset use ignore_index = True

```
result 1 = pd.concat([df1, df2], axis = 1)
print (result 1)

#      A    B    A    B
  0    1    3    5    7
  1    2    4    6    8
```

* concat adds NaN where there is no matching column.

* apply () - used to apply a function to each element in a series. It can take any function as an argument, including lambda functions.
applymap() - used to apply a function to each element in a DF.