INTRODUCTION TO PANDA

Pandas is a powerful and open-source Python library used for data manipulation and analysis.Pandas consist of data structures and functions to perform efficient operations on data.It is built on top of the NumPy library which means that a lot of the structures of NumPy are used or replicated in Pandas and the data produced by Pandas is often used as input for plotting functions in Matplotlib.

Pandas Usage

• Data set cleaning, merging, and joining.

• Easy handling of missing data (represented as NaN) in floating point as well as non-floating-point data.

• Columns can be inserted and deleted from DataFrame and higher-dimensional objects.

• Powerful group by functionality for performing split-apply-combine operations on data sets.

• Data Visualization.

• The Pandas library allows to work with tabular data with columns of different data types, such as that from Excel spreadsheets, CSV files from the internet, and SQL database tables,Time series data, either at fixed-frequency or not, other structured datasets, such as those coming from web data, like JSON files

The Panda module is generally imported as follows: import pandas as pd

Data Structures in Pandas Library

Pandas generally provide two data structures for manipulating data. They are: • **Series**: The Pandas Series structure, is a one-dimensional homogenous array. • **DataFrame**: The pandas DataFrame structure, is a twodimensional, mutable, and potentially heterogeneous structure.

Syntax to create a Series pandas.Series (data, index=idx (optional)) Where data may be python sequence (Lists), ndarray, scalar value or a python dictionary

How to create Series with nd array

```
import pandas as pd
import numpy as np
arr=np.array([10,15,18,22])
s = pd.Series(arr)
print(s)
```

```
0    10
1    15
2    18
```

```
3    22
dtype: int64
```

How to create Series with Mutable index

```
import pandas as pd
import numpy as np
arr=np.array(['a','b','c','d'])
s=pd.Series(arr, index=['first','second','third','fourth'])
print(s)
```

```
first     a
second    b
third     c
fourth    d
dtype: object
```

Creating a series from Scalar value

```
import pandas as pd
s = pd.Series(50, index=[0, 1, 2, 3, 4])
print(s)
```

```
0    50
1    50
2    50
3    50
4    50
dtype: int64
```

Mathematical Operations in Series

```
import pandas as pd

s1 = pd.Series([1, 2, 3, 4, 5], index=['a', 'b', 'c', 'd', 'e'])
s2 = pd.Series([10, 20, 30, 40, 50], index=['a', 'b', 'c', 'd', 'e'])
s3 = pd.Series([5, 14, 23, 32], index=['a', 'b', 'c', 'd'])

print('To Add Series1 & Series2')
print('-------------------------------------------')
print(s1 + s2)

print('To Add Series2 & Series3')
print('-------------------------------------------')
print(s2 + s3) #While adding two series, if Non-Matching Index is 
found in either of the Series, Then NaN will be printed corresponds to 
Non-Matching Index.

print('To Add Series2 & Series3 and Fill Non-Matching Index with 0')
```

```
print('---------------------------------------------')
print(s2.add(s3, fill_value=0))#If Non-Matching Index is found in either of the series, then this Non-Matching Index corresponding value of that series will be filled as 0.
```

```
To Add Series1 & Series2
---------------------------------------------
a    11
b    22
c    33
d    44
e    55
dtype: int64
To Add Series2 & Series3
---------------------------------------------
a    15.0
b    34.0
c    53.0
d    72.0
e     NaN
dtype: float64
To Add Series2 & Series3 and Fill Non-Matching Index with 0
---------------------------------------------
a    15.0
b    34.0
c    53.0
d    72.0
e    50.0
dtype: float64
```

Head and Tail Functions in Series head (): It is used to access the first 5 rows of a series.
Note :To access first 3 rows we can call series_name.head(3)

```
import pandas as pd
import numpy as np
arr = np.array([10, 15, 18, 22, 55, 77, 42, 48, 97])
s = pd.Series(arr)
print(s.head())
print(s.head(3))
```

```
0    10
1    15
2    18
3    22
4    55
dtype: int64
0    10
1    15
```

```
2    18
dtype: int64
```

tail(): It is used to access the last 5 rows of a series. Note :To access last 4 rows we can call series_name.tail (4)

```
import pandas as pd
import numpy as np
arr = np.array([10, 15, 18, 22, 55, 77, 42, 48, 97])
s = pd.Series(arr)
print(s.tail())
print(s.tail(4))
```

```
4    55
5    77
6    42
7    48
8    97
dtype: int64
5    77
6    42
7    48
8    97
dtype: int64
```

Selection in Series Series provides index label loc and iloc and [] to access rows and columns. 1. loc index label :- Syntax:-series_name.loc[StartRange: StopRange] 2. Selection Using iloc index label :- Syntax:-series_name.iloc[StartRange : StopRange] 3. Selection Using [] : Syntax:- series_name[StartRange> : StopRange] or series_name[index]

```
import pandas as pd
import numpy as np
arr = np.array([10, 15, 18, 22, 55, 77])
s = pd.Series(arr)
print(s)
print(s.loc[:2])
print(s.loc[3:4])
print(s.loc[2:3])
```

```
0    10
1    15
2    18
3    22
4    55
5    77
dtype: int64
0    10
1    15
2    18
```

```
dtype: int64
3    22
4    55
dtype: int64
2    18
3    22
dtype: int64
```

Indexing in Series Pandas provide index attribute to get or set the index of entries or values in series.

```
import pandas as pd
import numpy as np

# create a numpy array
arr = np.array(['a', 'b', 'c', 'd'])
s = pd.Series(arr, index=['first', 'second', 'third', 'fourth'])
print(s)
print(s.index)
```

```
first     a
second    b
third     c
fourth    d
dtype: object
Index(['first', 'second', 'third', 'fourth'], dtype='object')
```

DATAFRAME-It is a two-dimensional object that is useful in representing data in the form of rows and columns. It is similar to a spreadsheet or an SQL table. This is the most commonly used pandas object. Once we store the data into the Dataframe, we can perform various operations that are useful in analyzing and understanding the data. A Dataframe has axes (indices)- ➢ Row index (axis=0) ➢ Column index (axes=1) 2. It is similar to a spreadsheet , whose row index is called index and column index is called column name. 3. A Dataframe contains Heterogeneous data. 4. A Dataframe Size is Mutable. 5. A Dataframe Data is Mutable. 5. The data can also contain missing data, as represented by the NaN (not a number) values. A data frame can be created using any of the following- 1. Series 2. Lists 3. Dictionary 4. A numpy 2D array

How to create Dataframe From Series

```
import pandas as pd
s = pd.Series(['a','b','c','d'])
df=pd.DataFrame(s)
print(df)
```

```
   0
0  a
1  b
```

```
2   c
3   d
```

DataFrame from Dictionary of Series

```
import pandas as pd
name = pd.Series(['Hardik', 'Virat'])
team = pd.Series(['MI', 'RCB'])
dic = {'Name': name, 'Team': team}
df = pd.DataFrame(dic)
print(df)
```

```
     Name Team
0  Hardik   MI
1   Virat  RCB
```

DataFrame from List of Dictionaries

```
import pandas as pd

dicl = [
    {'FirstName': 'Sachin', 'LastName': 'Bhardwaj'},
    {'FirstName': 'Vinod', 'LastName': 'Verma'},
    {'FirstName': 'Rajesh', 'LastName': 'Mishra'}
]
df1 = pd.DataFrame(dicl)
print(df1)
```

```
  FirstName  LastName
0    Sachin  Bhardwaj
1     Vinod     Verma
2    Rajesh    Mishra
```

Iteration on Rows and Columns If we want to access record or data from a data frame row wise or column wise then iteration is used. Pandas provide 2 functions to perform iterations- 1. iterrows (): It is used to access the data row wise. 2. items (): It is used to access data coulumn wise

```
import pandas as pd

dicl = [
    {'FirstName': 'Sachin', 'LastName': 'Bhardwaj'},
    {'FirstName': 'Vinod', 'LastName': 'Verma'},
    {'FirstName': 'Rajesh', 'LastName': 'Mishra'}
]
df1 = pd.DataFrame(dicl)
print(df1)
for (row_index, row_value) in df1.iterrows():
    print("\nRow index is ::", row_index)
```

```
    print("Row Value is ::")
    print(row_value)
```

```
  FirstName  LastName
0    Sachin  Bhardwaj
1     Vinod     Verma
2    Rajesh    Mishra

Row index is :: 0
Row Value is ::
FirstName      Sachin
LastName     Bhardwaj
Name: 0, dtype: object

Row index is :: 1
Row Value is ::
FirstName    Vinod
LastName     Verma
Name: 1, dtype: object

Row index is :: 2
Row Value is ::
FirstName    Rajesh
LastName     Mishra
Name: 2, dtype: object
```

```
import pandas as pd

dicl = [
    {'FirstName': 'Sachin', 'LastName': 'Bhardwaj'},
    {'FirstName': 'Vinod', 'LastName': 'Verma'},
    {'FirstName': 'Rajesh', 'LastName': 'Mishra'}
]
df1 = pd.DataFrame(dicl)
print(df1)
for (coulumn_name, coulumn_value) in df1.items():
    print("\n Coulumn name is ::", coulumn_name)
    print("Coulumn Value is ::")
    print(coulumn_value)
```

```
  FirstName  LastName
0    Sachin  Bhardwaj
1     Vinod     Verma
2    Rajesh    Mishra

 Coulumn name is :: FirstName
Coulumn Value is ::
0    Sachin
1     Vinod
2    Rajesh
```

```
Name: FirstName, dtype: object

 Coulumn name is :: LastName
Coulumn Value is ::
0    Bhardwaj
1       Verma
2      Mishra
Name: LastName, dtype: object
```

Select operation in data frame To access the column data ,we can mention the column name as subscript. e.g. - df[empid]. This can also be done by using df.empid. To access multiple columns we can write as df[[col1, col2,---]]

```
import pandas as pd

empdata = {
    'empid': [101, 102, 103, 104, 105, 106],
    'ename': ['Sachin', 'Vinod', 'Lakhbir', 'Anil', 'Devinder',
'UmaSelvi'],
    'Doj': ['12-01-2012', '15-01-2012', '05-09-2007', '17-01-2012',
'05-09-2007', '16-01-2012']
}

df = pd.DataFrame(empdata)
print(df)

   empid     ename         Doj
0    101    Sachin  12-01-2012
1    102     Vinod  15-01-2012
2    103   Lakhbir  05-09-2007
3    104      Anil  17-01-2012
4    105  Devinder  05-09-2007
5    106  UmaSelvi  16-01-2012

df.empid

0    101
1    102
2    103
3    104
4    105
5    106
Name: empid, dtype: int64

df['empid']

0    101
1    102
2    103
3    104
```

```
4    105
5    106
Name: empid, dtype: int64
```

```
df[['empid','ename']]
```

```
   empid     ename
0    101    Sachin
1    102     Vinod
2    103   Lakhbir
3    104      Anil
4    105  Devinder
5    106  UmaSelvi
```

To Add & Rename a column in data frame

```
import pandas as pd
s = pd.Series([10,15,18,22])
df=pd.DataFrame(s)
df.columns=['List1'] #To Rename the default column of Data Frame as 
List1
df['List2']=20 #To create a new column List2 with all values as 20
df['List3']=df['List1']+df['List2'] #Add Column1 and Column2 and store 
in New column List3 print(df)
print(df)
```

```
   List1  List2  List3
0     10     20     30
1     15     20     35
2     18     20     38
3     22     20     42
```

To Delete a Column in data frameWe can delete the column from a data frame by using any of the the following – 1. del 2. pop() 3. drop()

```
import pandas as pd
s = pd.Series([10,15,18,22])
df=pd.DataFrame(s)
df.columns=['List1'] #To Rename the default column of Data Frame as 
List1
df['List2']=20 #To create a new column List2 with all values as 20
df['List3']=df['List1']+df['List2'] #Add Column1 and Column2 and store 
in New column List3 print(df)
print(df)
del df['List3']
print(df)
```

```
   List1  List2  List3
0     10     20     30
```

```
1     15     20     35
2     18     20     38
3     22     20     42
   List1  List2
0     10     20
1     15     20
2     18     20
3     22     20
```

```
df.pop('List2')
print(df)
```

```
   List1
0     10
1     15
2     18
3     22
```

```
import pandas as pd
s= pd.Series([10,20,30,40])
df=pd.DataFrame(s)
df.columns=['List1']
df['List2']=40
print(df)
df1=df.drop('List2',axis=1) #(axis=1) means to delete Data column wise

df2=df.drop(index=[2,3],axis=0) #(axis=0) means to delete data row
wise with given index print(df)
print(" After deletion::")
print(df1)
print (" After row deletion::")
print(df2)
```

```
   List1  List2
0     10     40
1     20     40
2     30     40
3     40     40
 After deletion::
   List1
0     10
1     20
2     30
3     40
 After row deletion::
   List1  List2
0     10     40
1     20     40
```

Accessing the data frame through loc() and iloc() method or indexing using Labels Pandas provide loc() and iloc() methods to access the subset from a data frame using row/column. *Accessing the data frame through loc()* It is used to access a group of rows and columns. Syntax- Df.loc[StartRow : EndRow, StartColumn : EndColumn]

```
import pandas as pd

Runs = {
    'TCS': {'Qtr1': 2500, 'Qtr2': 2000, 'Qtr3': 3000, 'Qtr4': 2000},
    'WIPRO': {'Qtr1': 2800, 'Qtr2': 2400, 'Qtr3': 3600, 'Qtr4': 2400},
    'L&T': {'Qtr1': 2100, 'Qtr2': 5700, 'Qtr3': 35000, 'Qtr4': 2100}
}

df = pd.DataFrame(Runs)
print(df)

# Select only Qtr3 row
print(df.loc['Qtr3', :])

# Select rows from Qtr1 to Qtr3
print(df.loc['Qtr1':'Qtr3', :])
```

```
       TCS  WIPRO    L&T
Qtr1  2500   2800   2100
Qtr2  2000   2400   5700
Qtr3  3000   3600  35000
Qtr4  2000   2400   2100
TCS       3000
WIPRO     3600
L&T      35000
Name: Qtr3, dtype: int64
       TCS  WIPRO    L&T
Qtr1  2500   2800   2100
Qtr2  2000   2400   5700
Qtr3  3000   3600  35000
```

```
print(df.loc[:,'TCS'])#To access single column
print(df.loc[:,'TCS':'WIPRO'])#To access multiple column
```

```
Qtr1    2500
Qtr2    2000
Qtr3    3000
Qtr4    2000
Name: TCS, dtype: int64
       TCS  WIPRO
Qtr1  2500   2800
Qtr2  2000   2400
Qtr3  3000   3600
Qtr4  2000   2400
```

```
import pandas as pd

# Data dictionary
empdata = {
    'empid': [101, 102, 103, 104, 105, 106],
    'ename': ['Sachin', 'Vinod', 'Lakhbir', 'Anil', 'Devinder',
'UmaSelvi'],
    'Doj': ['12-01-2012', '15-01-2012', '05-09-2007', '17-01-2012',
'05-09-2007', '16-01-2012']
}
df = pd.DataFrame(empdata)
print(df)
# Access first row using .loc
print(df.loc[0])
# Access first three rows using .loc
print(df.loc[0:2])
```

```
   empid     ename         Doj
0    101    Sachin  12-01-2012
1    102     Vinod  15-01-2012
2    103   Lakhbir  05-09-2007
3    104      Anil  17-01-2012
4    105  Devinder  05-09-2007
5    106  UmaSelvi  16-01-2012
empid           101
ename        Sachin
Doj      12-01-2012
Name: 0, dtype: object
   empid    ename         Doj
0    101   Sachin  12-01-2012
1    102    Vinod  15-01-2012
2    103  Lakhbir  05-09-2007
```

Accessing the data frame through iloc() It is used to access a group of rows and columns based on numeric index value. Syntax- Df.loc[StartRowindexs : EndRowindex, StartColumnindex : EndColumnindex]

```
import pandas as pd

# Data dictionary for company earnings
Runs = {
    'TCS': {'Qtr1': '2500', 'Qtr2': '2000', 'Qtr3': '3000', 'Qtr4':
'2000'},
    'WIPRO': {'Qtr1': '2800', 'Qtr2': '2400', 'Qtr3': '3600', 'Qtr4':
'2400'},
    'L&T': {'Qtr1': '2100', 'Qtr2': '5700', 'Qtr3': '35000', 'Qtr4':
'2100'}
}
```

```
df = pd.DataFrame(Runs)
print(df)
# Accessing specific rows and columns using iloc
print(df.iloc[0:2, 1:2])  # First 2 rows, second column
print(df.iloc[:, 0:2])    # All rows, first 2 columns
```

```
       TCS  WIPRO    L&T
Qtr1  2500   2800   2100
Qtr2  2000   2400   5700
Qtr3  3000   3600  35000
Qtr4  2000   2400   2100
     WIPRO
Qtr1  2800
Qtr2  2400
       TCS  WIPRO
Qtr1  2500   2800
Qtr2  2000   2400
Qtr3  3000   3600
Qtr4  2000   2400
```

head() and tail() Method The method head() gives the first 5 rows and the method tail() returns the last 5 rows. To display first 2 rows we can use head(2) and to returns last2 rows we can use tail(2) and to return 3rd to 4th row we can write df[2:5].

```
import pandas as pd
empdata={ 'Doj':['12-01-2012','15-01-2012','05-09-2007', '17-01-
2012','05-09-2007','16-01-2012'], 'empid':[101,102,103,104,105,106],
'ename':['Sachin','Vinod','Lakhbir','Anil','Devinder','UmaSelvi'] }
df=pd.DataFrame(empdata)
print(df)
print(df.head())
print(df.tail())
```

```
          Doj  empid     ename
0  12-01-2012    101    Sachin
1  15-01-2012    102     Vinod
2  05-09-2007    103   Lakhbir
3  17-01-2012    104      Anil
4  05-09-2007    105  Devinder
5  16-01-2012    106  UmaSelvi
          Doj  empid     ename
0  12-01-2012    101    Sachin
1  15-01-2012    102     Vinod
2  05-09-2007    103   Lakhbir
3  17-01-2012    104      Anil
4  05-09-2007    105  Devinder
          Doj  empid     ename
1  15-01-2012    102     Vinod
2  05-09-2007    103   Lakhbir
```

```
3  17-01-2012    104      Anil
4  05-09-2007    105  Devinder
5  16-01-2012    106  UmaSelvi
```

Accessing a Specific DataFrame Cell by Row and Column

```python
import pandas as pd

# Data for students' marks in different tests
marks = pd.DataFrame({
    'Wally' : [87, 89, 93, 87, 92],
    'Eva' : [95, 99, 87, 88, 84],
    'Sam' : [88, 94, 85, 89, 95],
    'Katie' : [87, 92, 95, 84, 85],
    'Bob' : [83, 93, 86, 83, 87]
}, index = ['Test01', 'Test02', 'Test03', 'Test04', 'Test05'])

# Printing all marks
print('All marks:')
print(marks)

# Accessing Eva's Test01 marks using different methods
print('\nAccessing Eva\'s Test01 marks: ')
print('Method 01:', marks['Eva']['Test01'])  # Column name then Index name
print('Method 02:', marks.Eva.Test01)        # Column name then Index name
print('Method 03:', marks.at['Test01', 'Eva'])  # Row name then Column name
print('Method 04:', marks.iat[0, 1])  # Row index then Column index
```

```
All marks:
        Wally  Eva  Sam  Katie  Bob
Test01     87   95   88     87   83
Test02     89   99   94     92   93
Test03     93   87   85     95   86
Test04     87   88   89     84   83
Test05     92   84   95     85   87

Accessing Eva's Test01 marks:
Method 01: 95
Method 02: 95
Method 03: 95
Method 04: 95
```

Boolean Indexing

```python
import pandas as pd
```

```python
# Data for students' marks in different tests
marks = pd.DataFrame({
    'Wally' : [87, 89, 93, 87, 92],
    'Eva' : [95, 99, 87, 88, 84],
    'Sam' : [88, 94, 85, 89, 95],
    'Katie' : [87, 92, 95, 84, 85],
    'Bob' : [83, 93, 86, 83, 87]
}, index = ['Test01', 'Test02', 'Test03', 'Test04', 'Test05'])

# Displaying 0 grade students
print('0 grade students:')
print(marks[marks >= 90])  # Filter students with marks >= 90

# Displaying A grade students
print('\nA grade students:')
print(marks[(marks >= 80) & (marks < 90)])  # Filter students with
marks between 80 and 90
```

```
0 grade students:
        Wally   Eva   Sam  Katie   Bob
Test01    NaN  95.0   NaN    NaN   NaN
Test02    NaN  99.0  94.0   92.0  93.0
Test03   93.0   NaN   NaN   95.0   NaN
Test04    NaN   NaN   NaN    NaN   NaN
Test05   92.0   NaN  95.0    NaN   NaN

A grade students:
        Wally   Eva   Sam  Katie   Bob
Test01   87.0   NaN  88.0   87.0  83.0
Test02   89.0   NaN   NaN    NaN   NaN
Test03    NaN  87.0  85.0    NaN  86.0
Test04   87.0  88.0  89.0   84.0  83.0
Test05    NaN  84.0   NaN   85.0  87.0
```

Transposing the DataFrame with the T Attribute

```python
import pandas as pd

# Create a DataFrame with student marks
marks = pd.DataFrame({
    'Wally': [87, 89, 93, 87, 92],
    'Eva': [95, 99, 87, 88, 84],
    'Sam': [88, 94, 85, 89, 95]
}, index=['Test01', 'Test02', 'Test03', 'Test04', 'Test05'])

# Transpose the DataFrame
marksTransposed = marks.T

# Display results
print('All marks:')
```

```python
print(marks)

print('\nAll marks Transposed:')
print(marksTransposed)
```

```
All marks:
        Wally  Eva  Sam
Test01     87   95   88
Test02     89   99   94
Test03     93   87   85
Test04     87   88   89
Test05     92   84   95

All marks Transposed:
       Test01  Test02  Test03  Test04  Test05
Wally      87      89      93      87      92
Eva        95      99      87      88      84
Sam        88      94      85      89      95
```

Sorting by Rows and Columns by Their Indices (axis=0 for rows and axis = 1 for columns)

```python
import pandas as pd

# Create a DataFrame with student marks
marks = pd.DataFrame({
    'Wally': [87, 89, 93],
    'Eva': [95, 99, 87],
    'Sam': [88, 94, 85],
    'Katie': [87, 92, 95],
    'Bob': [83, 93, 86]
}, index=['Test01', 'Test02', 'Test03'])

# Display the DataFrame
print('All marks:')
print(marks)

# Sort by row indices (descending order)
print('\nAll marks Sorted by Row indices:')
print(marks.sort_index(ascending=False))

# Sort by column indices (alphabetical order of names)
print('\nAll marks Sorted by Column indices:')
print(marks.sort_index(axis=1))
```

```
All marks:
        Wally  Eva  Sam  Katie  Bob
Test01     87   95   88     87   83
Test02     89   99   94     92   93
Test03     93   87   85     95   86
```

```
All marks Sorted by Row indices:
        Wally  Eva  Sam  Katie  Bob
Test03     93   87   85     95   86
Test02     89   99   94     92   93
Test01     87   95   88     87   83

All marks Sorted by Column indices:
        Bob  Eva  Katie  Sam  Wally
Test01   83   95     87   88     87
Test02   93   99     92   94     89
Test03   86   87     95   85     93
```

Sorting by Column Values (axis = 1 for columns)

```
import pandas as pd

# Create a DataFrame with student marks
marks = pd.DataFrame({
    'Wally': [87, 89, 93],
    'Eva': [95, 99, 87],
    'Sam': [88, 94, 85],
    'Katie': [87, 92, 95],
    'Bob': [83, 93, 86]
}, index=['Test01', 'Test02', 'Test03'])

# Display original marks
print('All marks:')
print(marks)

# Sort columns by values of Test01 and Test02 (descending order)
print('\nAll marks Sorted by Column values:')
print(marks.sort_values(by='Test01', axis=1, ascending=False))
print(marks.sort_values(by='Test02', axis=1, ascending=False))

# Transpose and sort by Test01 row values (descending order)
print('\nTranspose and sort:')
print(marks.T.sort_values(by='Test01', ascending=False))

# Select Test01 row and sort values (descending order)
print('\nSelect and sort:')
print(marks.loc['Test01'].sort_values(ascending=False))

All marks:
        Wally  Eva  Sam  Katie  Bob
Test01     87   95   88     87   83
Test02     89   99   94     92   93
Test03     93   87   85     95   86

All marks Sorted by Column values:
        Eva  Sam  Wally  Katie  Bob
```

```
Test01   95   88     87     87   83
Test02   99   94     89     92   93
Test03   87   85     93     95   86
        Eva  Sam  Bob  Katie  Wally
Test01   95   88   83     87     87
Test02   99   94   93     92     89
Test03   87   85   86     95     93

Transpose and sort:
      Test01  Test02  Test03
Eva       95      99      87
Sam       88      94      85
Wally     87      89      93
Katie     87      92      95
Bob       83      93      86

Select and sort:
Eva      95
Sam      88
Wally    87
Katie    87
Bob      83
Name: Test01, dtype: int64
```

Creating CSV file

```
import pandas as pd

# Create a sample dictionary of data
data = {
    'Name': ['Alice', 'Bob', 'Charlie', 'David'],
    'Age': [24, 27, 22, 32],
    'City': ['New York', 'Los Angeles', 'Chicago', 'Houston']
}

# Convert dictionary to DataFrame
df = pd.DataFrame(data)

# Save DataFrame to CSV file
df.to_csv('Data.csv', index=False)

print("CSV file 'Data.csv' created successfully!")

CSV file 'Data.csv' created successfully!

df2 = pd.read_csv('Data.csv')
print(df2)

      Name  Age         City
0    Alice   24     New York
```

```
1      Bob   27  Los Angeles
2  Charlie   22      Chicago
3    David   32      Houston
```

Downloading a csv data file directly from the web

```
import pandas as pd

# Define column names from UCI Auto MPG dataset
column_names = [
    "mpg", "cylinders", "displacement", "horsepower", "weight",
    "acceleration", "model_year", "origin", "car_name"
]

# URL of Auto MPG dataset
url = "https://archive.ics.uci.edu/ml/machine-learning-databases/auto-
mpg/auto-mpg.data"

# Load dataset
df = pd.read_csv(
    url,
    sep='\s+',   # data separated by spaces
    names=column_names,      # assign column names
    na_values="?"            # missing values marked as '?'
)

# Save to local CSV file
df.to_csv("auto_mpg.csv", index=False)

print("� Auto MPG dataset saved as 'auto_mpg.csv'")
print(df.head())

� Auto MPG dataset saved as 'auto_mpg.csv'
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5

   model_year  origin                   car_name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino

df.head(10)
```

```
    mpg  cylinders  displacement  horsepower  weight  acceleration  \
0  18.0          8         307.0       130.0  3504.0          12.0
1  15.0          8         350.0       165.0  3693.0          11.5
2  18.0          8         318.0       150.0  3436.0          11.0
3  16.0          8         304.0       150.0  3433.0          12.0
4  17.0          8         302.0       140.0  3449.0          10.5
5  15.0          8         429.0       198.0  4341.0          10.0
6  14.0          8         454.0       220.0  4354.0           9.0
7  14.0          8         440.0       215.0  4312.0           8.5
8  14.0          8         455.0       225.0  4425.0          10.0
9  15.0          8         390.0       190.0  3850.0           8.5

   model_year  origin                   car_name
0          70       1  chevrolet chevelle malibu
1          70       1          buick skylark 320
2          70       1         plymouth satellite
3          70       1              amc rebel sst
4          70       1                ford torino
5          70       1           ford galaxie 500
6          70       1           chevrolet impala
7          70       1          plymouth fury iii
8          70       1           pontiac catalina
9          70       1         amc ambassador dpl
```

```
df.tail(10)
```

```
      mpg  cylinders  displacement  horsepower  weight
acceleration  \
388  26.0          4         156.0        92.0  2585.0          14.5

389  22.0          6         232.0       112.0  2835.0          14.7

390  32.0          4         144.0        96.0  2665.0          13.9

391  36.0          4         135.0        84.0  2370.0          13.0

392  27.0          4         151.0        90.0  2950.0          17.3

393  27.0          4         140.0        86.0  2790.0          15.6

394  44.0          4          97.0        52.0  2130.0          24.6

395  32.0          4         135.0        84.0  2295.0          11.6

396  28.0          4         120.0        79.0  2625.0          18.6

397  31.0          4         119.0        82.0  2720.0          19.4


     model_year  origin                   car_name
388          82       1  chrysler lebaron medallion
```

```
389          82       1              ford granada l
390          82       3            toyota celica gt
391          82       1           dodge charger 2.2
392          82       1            chevrolet camaro
393          82       1             ford mustang gl
394          82       2                   vw pickup
395          82       1               dodge rampage
396          82       1                 ford ranger
397          82       1                  chevy s-10
```

Understanding data using df.info() The df.info() method is a quick way to look at the data types, missing values, and data size of a DataFrame. • show_counts = True: gives a few over the total nonmissing values in each column. • memory_usage = True: shows the total memory usage of the DataFrame elements. • verbose = True: prints the full summary from df.info()

```
df.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 398 entries, 0 to 397
Data columns (total 9 columns):
 #   Column        Non-Null Count  Dtype
---  ------        --------------  -----
 0   mpg           398 non-null    float64
 1   cylinders     398 non-null    int64
 2   displacement  398 non-null    float64
 3   horsepower    392 non-null    float64
 4   weight        398 non-null    float64
 5   acceleration  398 non-null    float64
 6   model_year    398 non-null    int64
 7   origin        398 non-null    int64
 8   car_name      398 non-null    object
dtypes: float64(5), int64(3), object(1)
memory usage: 28.1+ KB
```

Understanding data using df.describe() Prints the summary statistics of all numeric columns, such as • count, • mean, • standard deviation, • range, and • quartiles of numeric columns.

```
df.describe()

              mpg   cylinders  displacement  horsepower
weight  \
count  398.000000  398.000000    398.000000  392.000000   398.000000

mean    23.514573    5.454774    193.425879  104.469388  2970.424623

std      7.815984    1.701004    104.269838   38.491160   846.841774

min      9.000000    3.000000     68.000000   46.000000  1613.000000
```

```
25%     17.500000     4.000000    104.250000   75.000000  2223.750000

50%     23.000000     4.000000    148.500000   93.500000  2803.500000

75%     29.000000     8.000000    262.000000  126.000000  3608.000000

max     46.600000     8.000000    455.000000  230.000000  5140.000000


       acceleration  model_year      origin
count    398.000000  398.000000  398.000000
mean      15.568090   76.010050    1.572864
std        2.757689    3.697627    0.802055
min        8.000000   70.000000    1.000000
25%       13.825000   73.000000    1.000000
50%       15.500000   76.000000    1.000000
75%       17.175000   79.000000    2.000000
max       24.800000   82.000000    3.000000
```

Modifying the quartiles • You can also modify the quartiles using the percentiles argument. • Here, for example, we're looking at the 30%, 50%, and 70% percentiles of the numeric columns in DataFrame df.

```
df.describe(percentiles=[0.3,0.5,0.7])

              mpg   cylinders  displacement  horsepower
weight  \
count  398.000000  398.000000    398.000000  392.000000   398.000000

mean    23.514573    5.454774    193.425879  104.469388  2970.424623

std      7.815984    1.701004    104.269838   38.491160   846.841774

min      9.000000    3.000000     68.000000   46.000000  1613.000000

30%     18.000000    4.000000    112.000000   80.000000  2301.000000

50%     23.000000    4.000000    148.500000   93.500000  2803.500000

70%     27.490000    6.000000    250.000000  110.000000  3424.500000

max     46.600000    8.000000    455.000000  230.000000  5140.000000


       acceleration  model_year      origin
count    398.000000  398.000000  398.000000
mean      15.568090   76.010050    1.572864
std        2.757689    3.697627    0.802055
min        8.000000   70.000000    1.000000
30%       14.200000   73.000000    1.000000
```

```
50%         15.500000   76.000000    1.000000
70%         16.800000   78.000000    2.000000
max         24.800000   82.000000    3.000000
```

Categorical Data In case of categorical data the df.describe()method summarizes by • number of observations, • number of unique elements, • mode, and • frequency of the mode.

```
df['car_name'].describe()
```

```
count                 398
unique                305
top            ford pinto
freq                    6
Name: car_name, dtype: object
```

```
df.describe(include='all')
```

```
               mpg   cylinders  displacement  horsepower       weight
\
count   398.000000  398.000000    398.000000  392.000000   398.000000

unique         NaN         NaN           NaN         NaN          NaN

top            NaN         NaN           NaN         NaN          NaN

freq           NaN         NaN           NaN         NaN          NaN

mean     23.514573    5.454774    193.425879  104.469388  2970.424623

std       7.815984    1.701004    104.269838   38.491160   846.841774

min       9.000000    3.000000     68.000000   46.000000  1613.000000

25%      17.500000    4.000000    104.250000   75.000000  2223.750000

50%      23.000000    4.000000    148.500000   93.500000  2803.500000

75%      29.000000    8.000000    262.000000  126.000000  3608.000000

max      46.600000    8.000000    455.000000  230.000000  5140.000000


        acceleration  model_year      origin    car_name
count     398.000000  398.000000  398.000000         398
unique           NaN         NaN         NaN         305
top              NaN         NaN         NaN  ford pinto
freq             NaN         NaN         NaN           6
mean       15.568090   76.010050    1.572864         NaN
std         2.757689    3.697627    0.802055         NaN
min         8.000000   70.000000    1.000000         NaN
25%        13.825000   73.000000    1.000000         NaN
```

```
50%        15.500000   76.000000    1.000000         NaN
75%        17.175000   79.000000    2.000000         NaN
max        24.800000   82.000000    3.000000         NaN
```

Understanding your data using df.shape • The number of rows and columns of a DataFrame can be identified using the .shape attribute of the DataFrame. • It returns a tuple (row, column) and can be indexed to get only rows, and only columns count as output.

```
print('(Rows, Cols): ', df.shape)
print('Rows: ', df.shape[0])
print('Cols: ', df.shape[1])

(Rows, Cols):  (398, 9)
Rows:  398
Cols:  9
```

Get all columns and column names • Calling the df.columns attribute of a DataFrame object returns the column names in the form of an Index object. • As a reminder, a pandas index is the address/label of the row or column. • This can also be converted to a Python list object.

```
print(df.columns)
col_list=list(df.columns)

Index(['mpg', 'cylinders', 'displacement', 'horsepower', 'weight',
       'acceleration', 'model_year', 'origin', 'car_name'],
      dtype='object')
```

Handling Duplicates in a DataFrame **df.duplicated()**: Used to identify duplicate rows in a DataFrame. It returns a boolean Series where True indicates a row is a duplicate of a previous row,False indicates it is not. **df.drop_duplicates()**: will return a copy of your DataFrame, with duplicates removed. **df.drop_duplicates(inplace=True)**: will modify the DataFrame object in place, with duplicates removed **df.drop_duplicates(inplace=True, keep=first)**: Drop duplicates in place except for the first occurrence. **df.drop_duplicates(inplace=True, keep=last)**: Drop duplicates in place except for the last occurrence **df.drop_duplicates(inplace=True, keep=False)**: Drop all duplicates.

```
print(df.duplicated())

0      False
1      False
2      False
3      False
4      False
       ...
393    False
394    False
395    False
396    False
```

```
397    False
Length: 398, dtype: bool
```

```python
import pandas as pd

# Create a sample DataFrame with duplicate rows and values
data = {
    'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob', 'David'],
    'Age': [25, 30, 25, 35, 30, 40],
    'City': ['New York', 'London', 'New York', 'Paris', 'London',
'Tokyo']
}

df = pd.DataFrame(data)
print("=== Handling Duplicates in a DataFrame ===")

# 1. Display the original DataFrame
print("\n1. Original DataFrame:")
print(df)
```

```
=== Handling Duplicates in a DataFrame ===

1. Original DataFrame:
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
2    Alice   25  New York
3  Charlie   35     Paris
4      Bob   30    London
5    David   40     Tokyo
```

```python
print("\n2. Identify duplicates (duplicated()):")
print("Rows marked as duplicates (True means duplicate):")
print(df.duplicated())   # Checks for duplicate rows

print("\nDuplicate rows only:")
print(df[df.duplicated()])
```

```

2. Identify duplicates (duplicated()):
Rows marked as duplicates (True means duplicate):
0    False
1    False
2     True
3    False
4     True
5    False
dtype: bool

Duplicate rows only:
    Name  Age      City
```

```
2  Alice   25  New York
4    Bob   30    London

# 3. Drop duplicates using drop_duplicates() - keep first occurrence
print("\n3. Drop duplicates (keep='first'):")

df_first = df.drop_duplicates()
print(df_first)


3. Drop duplicates (keep='first'):
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
3  Charlie   35     Paris
5    David   40     Tokyo

# 4. Drop duplicates using drop_duplicates() - keep last occurrence
print("\n4. Drop duplicates (keep='last'):")

df_last = df.drop_duplicates(keep='last')
print(df_last)


4. Drop duplicates (keep='last'):
      Name  Age      City
2    Alice   25  New York
3  Charlie   35     Paris
4      Bob   30    London
5    David   40     Tokyo

# 5. Drop duplicates based on specific columns (e.g., 'Name' and
'Age')
print("\n5. Drop duplicates based on 'Name' and 'Age':")

df_subset = df.drop_duplicates(subset=['Name', 'Age'])
print(df_subset)


5. Drop duplicates based on 'Name' and 'Age':
      Name  Age      City
0    Alice   25  New York
1      Bob   30    London
3  Charlie   35     Paris
5    David   40     Tokyo

# 6. Drop duplicates and keep neither (drop all duplicates)
print("\n6. Drop duplicates (keep=False):")

df_none = df.drop_duplicates(keep=False)
print(df_none)
```

```
6. Drop duplicates (keep=False):
      Name  Age   City
3  Charlie   35  Paris
5    David   40  Tokyo

# 7. Count duplicates
print("\n7. Count duplicates:")

duplicate_count = df.duplicated().sum()
print(f"Number of duplicate rows: {duplicate_count}")


7. Count duplicates:
Number of duplicate rows: 2

# 8. Handling duplicates in a specific column (e.g., 'Name')
print("\n8. Unique values in 'Name' column:")

unique_names = df['Name'].drop_duplicates()
print(unique_names)


8. Unique values in 'Name' column:
0      Alice
1        Bob
3    Charlie
5      David
Name: Name, dtype: object
```

Handling NaN Values • Detect NaN Values (isna()): df.isna() returns a boolean DataFrame where True indicates a NaN value. • df.isna().sum() counts NaN values per column. • Drop Rows with Any NaN (dropna()): df.dropna() removes rows containing any NaN values. • Drop Rows Where All Values Are NaN: df.dropna(how='all') removes rows where all columns are NaN (none in this example). • Drop Rows with NaN in Specific Columns:df.dropna(subset=['Age', 'Salary']) removes rows where 'Age' or 'Salary' is NaN. • Fill NaN with Specific Values (fillna()): Replaces NaN with a specified value (e.g., 0 for numeric columns, 'Unknown' for strings). • Fill NaN with Column Mean: Uses df['column'].mean() to compute the mean of non- NaN values and fills NaN with that value. • Forward Fill (fillna(method='ffill')): Propagates the last valid value forward to fill NaN. • Interpolate NaN Values: df['column'].interpolate(method='linear') estimates NaN values by interpolating between neighboring values (works for numeric columns).

```
import pandas as pd
import numpy as np

# Create a sample DataFrame with NaN values
data = {
    'Name': ['Alice', 'Bob', 'Alice', 'Charlie', 'Bob', 'David'],
    'Age': [25, np.nan, 25, 35, np.nan, 40],
    'City': ['New York', 'London', np.nan, 'Paris', 'London', np.nan],
```

```python
    'Salary': [50000, 60000, np.nan, np.nan, 60000, 70000]
}

df = pd.DataFrame(data)

# --- Demonstration of Handling NaN Values ---
print("=== Handling NaN Values in a DataFrame ===")

# 1. Display the original DataFrame
print("\n1. Original DataFrame with NaN values:")
print(df)
```

```
=== Handling NaN Values in a DataFrame ===

1. Original DataFrame with NaN values:
      Name   Age      City   Salary
0    Alice  25.0  New York  50000.0
1      Bob   NaN    London  60000.0
2    Alice  25.0       NaN      NaN
3  Charlie  35.0     Paris      NaN
4      Bob   NaN    London  60000.0
5    David  40.0       NaN  70000.0
```

```python
# 2. Detect NaN values using isna()
print("\n2. Detect NaN values (isna()):")
print(df.isna())

print("\nCount of NaN values per column:")
print(df.isna().sum())
```

```
2. Detect NaN values (isna()):
    Name    Age   City  Salary
0  False  False  False   False
1  False   True  False   False
2  False  False   True    True
3  False  False  False    True
4  False   True  False   False
5  False  False   True   False

Count of NaN values per column:
Name      0
Age       2
City      2
Salary    2
dtype: int64
```

```python
# 3. Drop rows with any NaN values
print("\n3. Drop rows with any NaN (dropna()):")
df_drop_rows = df.dropna()
print(df_drop_rows)
```

```
3. Drop rows with any NaN (dropna()):
    Name   Age      City   Salary
0  Alice  25.0  New York  50000.0

# 4. Drop rows where all values are NaN (none in this case)
print("\n4. Drop rows where all values are NaN:")
df_drop_all = df.dropna(how='all')
print(df_drop_all)


4. Drop rows where all values are NaN:
      Name   Age      City   Salary
0    Alice  25.0  New York  50000.0
1      Bob   NaN    London  60000.0
2    Alice  25.0       NaN      NaN
3  Charlie  35.0     Paris      NaN
4      Bob   NaN    London  60000.0
5    David  40.0       NaN  70000.0

# 5. Drop rows where NaN appears in specific columns (e.g., 'Age' and
'Salary')
print("\n5. Drop rows with NaN in 'Age' or 'Salary':")
df_drop_subset = df.dropna(subset=['Age', 'Salary'])
print(df_drop_subset)


5. Drop rows with NaN in 'Age' or 'Salary':
    Name   Age      City   Salary
0  Alice  25.0  New York  50000.0
5  David  40.0       NaN  70000.0

# 6. Fill NaN values with a specific value (e.g., 0 for numeric,
'Unknown' for strings)
print("\n6. Fill NaN values with specific values:")
df_fill = df.copy()
df_fill['Age'] = df_fill['Age'].fillna(0)
df_fill['City'] = df_fill['City'].fillna('Unknown')
df_fill['Salary'] = df_fill['Salary'].fillna(0)
print(df_fill)


6. Fill NaN values with specific values:
      Name   Age      City   Salary
0    Alice  25.0  New York  50000.0
1      Bob   0.0    London  60000.0
2    Alice  25.0   Unknown      0.0
3  Charlie  35.0     Paris      0.0
4      Bob   0.0    London  60000.0
5    David  40.0   Unknown  70000.0
```

```
# 7. Fill NaN values with column mean (for numeric columns)
print("\n7. Fill NaN values with column mean (Age and Salary):")
df_fill_mean = df.copy()
df_fill_mean['Age'] =
df_fill_mean['Age'].fillna(df_fill_mean['Age'].mean())
df_fill_mean['Salary'] =
df_fill_mean['Salary'].fillna(df_fill_mean['Salary'].mean())
print(df_fill_mean)
```

```
7. Fill NaN values with column mean (Age and Salary):
      Name    Age      City   Salary
0    Alice  25.00  New York  50000.0
1      Bob  31.25    London  60000.0
2    Alice  25.00       NaN  60000.0
3  Charlie  35.00     Paris  60000.0
4      Bob  31.25    London  60000.0
5    David  40.00       NaN  70000.0
```

```
# 8. Forward fill NaN values (propagate previous value forward)
print("\n9. Forward fill NaN values (using obj.ffill):")
df_interpolate = df.copy()
df_interpolate['Age'] = df_interpolate['Age'].ffill()
df_interpolate['Salary'] = df_interpolate['Salary'].ffill()
print(df_interpolate)
```

```
9. Forward fill NaN values (using obj.ffill):
      Name   Age      City   Salary
0    Alice  25.0  New York  50000.0
1      Bob  25.0    London  60000.0
2    Alice  25.0       NaN  60000.0
3  Charlie  35.0     Paris  60000.0
4      Bob  35.0    London  60000.0
5    David  40.0       NaN  70000.0
```

```
# 9. Interpolate NaN values (linear interpolation for numeric columns)
print("\n9. Interpolate NaN values (linear):")
df_interpolate = df.copy()
df_interpolate['Age'] =
df_interpolate['Age'].interpolate(method='linear')
df_interpolate['Salary'] =
df_interpolate['Salary'].interpolate(method='linear')
print(df_interpolate)
```

```
9. Interpolate NaN values (linear):
      Name   Age      City   Salary
0    Alice  25.0  New York  50000.0
1      Bob  25.0    London  60000.0
2    Alice  25.0       NaN  60000.0
```

```
3  Charlie  35.0     Paris  60000.0
4      Bob  37.5    London  60000.0
5    David  40.0       NaN  70000.0
```