

Pandas

① Pandas:-

- ↳ Wes McKinney in 2008.
- ↳ Open source library - mainly work with relational/ labeled data easily.
- ↳ Various data store work with numeric data & time series.
- ↳ Fast, high performance & productivity.
- ↳ Function - Analyzing, cleaning, exploring & manipulating data.
- ↳ Data cleaning - Delete irrelevant rows / wrong values empty / NULL values.

② Why Pandas:-

- ↳ Fast & efficient data manipulation & analyzing.
- ↳ Easy load data.
- ↳ Flexible reshaping & pivoting data sets.
- ↳ Time series analysis functionality.

③ Uses of Pandas:-

- ↳ Data set cleaning, merging & joining.
- ↳ Easy handle missing data (NaN).
- ↳ Easy insert, delete col & records in DF.
- ↳ Groupby functionality.
- ↳ Split-apply-combine operations.
- ↳ Data visualization.

① Get started with Pandas:-

① Install Pandas - Use pip command:

```
pip install pandas
```

② Import Pandas - Use following import statement:

```
import pandas as pd
```

② Data Structures:-

1) Series:-

↳ 1D labeled array hold - int, str, float, object.

↳ Labels must be hashable (immutable) type.

↳ In real world create series from: existing storage, SQL database, CSV file/Excel file.

↳ Can create from - List, dict & scalar values.

```
>>> arr = np.array([10, 15, 20, 25])
```

```
>>> sr = pd.Series(arr)
```

O/P →	0	10
	1	15
	2	20
	3	25

• Series - Just like a column in table/excel sheet.

• Labels - Default, index start from 0.

↳ Used to access values in series.

- Create labels for series - with index arg, can give labels to the series.

```
>>> s = pd.Series(a, index = ['a', 'b'])
```

O/p →

a	1
b	2

len = series len.

dtype: int64

- Key/Value object - create series from Dict.

↳ keys become labels.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
```

```
>>> s = pd.Series(d)
```

O/p →

a	10
b	20
c	30

- Series with selected Dict keys - Use index arg, to specify items to select.

```
>>> d = {'a': 10, 'b': 20, 'c': 30}
```

```
>>> s = pd.Series(d, index = ['a', 'c'])
```

O/p →

a	10
c	30

————→ 'b' is not taken.

- Series Represented by -

<class 'pandas.core.series.Series'>

2) DataFrame :-

↳ 2D size-mutable, heterogeneous tabular data stru. with labeled axis (rows & cols).

↳ cols - Also call Feature, variable, field, dimension.

↳ rows - Records, values, ~~obj~~ observations, index.

• Applications of DataFrame:

① Work on data set.

② Analysis

③ Dropping

④ Processing

⑤ cleaning

⑥ Join multiple data (CSV, excel file format)

⑦ create Excel, CSV, JSON, binary files.

⑧ Math & statistical operaⁿ.

⑨ Use of Groupby.

• Represented by -

```
<class 'pandas.core.frame.DataFrame'>
```

• Creating DataFrame -

① From dict with List values -

```
>>> data = {'a': [1, 2], 'b': [11, 12]}
```

```
>>> df = pd.DataFrame(data)
```

O/p →

	a	b
0	1	11
1	2	12

② Same value to all rows:-

```
>>> data = {'name': 'A', 'b': [1, 2], 'c': [11, 12]}
```

```
>>> df = pd.DataFrame(data)
```

O/p →

	name	a	b
0	A	1	11
1	A	2	12

Same value to all the columns.

③ Dict of numpy arrays:-

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

```
>>> b = np.array([11, 12])
```

```
>>> d = {'a': a, 'b': b}
```

```
>>> df = pd.DataFrame(d)
```

O/p →

	a	b
0	1	11
1	2	12

④ List of Lists -

```
>>> lst = [['a', 'b'], [1, 2], [11, 12]]
```

```
>>> df = pd.DataFrame(dict(zip(lst[0], lst[1:])))
```

O/p →

	a	b
0	1	11
1	2	12

⑤ Named Indexes - Use index argument -

```
>>> df = pd.DataFrame(d, index=['id1', 'id2'])
```

O/p →

	a	b
id1	1	11
id2	2	12

• Importing & Exporting Dataframes -

① CSV File - comma-separated values.

`df.to_csv('file.csv')`

`df = pd.read_csv('file.csv')`

② Excel File -

`df.to_excel('file.xlsx')`

`df = pd.read_excel('file.xlsx')`

③ JSON - same format like Python Dict.

`df.to_json('file.json')`

`df = pd.read_json('file.json')`

④ HTML - Export to HTML `<table>` element.

`df.to_html('file.html')`

`df = pd.read_html('file.html')`

• DataFrame Functions -

1) checking Size & Index:

① size - Size of DataFrame = rows * cols.

Ex. [545 rows * 13 columns]

7085

② index - Range of index from start to end.

O/p - RangeIndex (start=0, stop=544, step=1)

2) Get columns of DF -

① columns - Get names of columns.

O/p - `Index ([.....], dtype = 'object')`
List of columns.

② axes - Range of index & names of columns.

O/p - `[RangeIndex (start=0, stop=544, step=1),
Index ([.....], dtype = 'object')]`

● ① Getting DF Information:

① df.info() - Get info of overall DF.

`>>> df.info()`

O/p → `<class 'pandas.core.frame.DataFrame'>`

Range Index: 545 entries, 0 to 544

Data columns (total ____ columns):

#	Column	Non-Null count	Dtype
---	---	---	---

dtypes: int64(), object(),

Memory usage: 55.5+ KB

None

② df.describe() - Return all stat funⁿ values, of numeric cols only.

↳ Not work for str(object) type.

>>> df.describe()

O/P →	col names			
count	-			
mean	-			
std	-			
min	-			
25%	-			
50%	-			
75%	-			
max	-			

③ max_rows - Define num of rows returned.

↳ Found at: pd.options.display.max_rows.

↳ Default value is ~~60~~ 60.

↳ So, for DF with rows > 60, when print() return header, 1st & last 5 rows.

• change value -

pd.options.display.max_rows = 9999

Set other value.

① Viewing Data from DF:-

① df.to_string() - Print entire DF.

② df.head() - Return top columns.

↳ Default 5, but can give num.

df.head() → 5 rows

df.head(10) → 10 rows

③ df.tail() - Return bottom columns.

↳ Default 5, but can give num.

df.tail() → 5 rows

df.tail(10) → 10 rows

④ df.isna() - show all NULL values in the DF.

↳ Return DF with bools:

- True: NULL values

- False: Non-NULL values

```
>>> df -
```

	a	b
0	10	15
1	20	np.nan

```
>>> df.isna()
```

O/p →

	a	b
0	T	T
1	T	F

```
>>> df.isna().sum()
```

O/p →

a	0	}	Return count of null values in each columns.
b	1		

dtype: int64

• Transpose of DataFrame -

↳ Convert rows to cols & cols to rows.

`>>> df.transpose()`
`>>> df.T` } Both serve same.

	a	b
0	1	11
1	2	12

(Transpose) \Rightarrow

	a	b
0	1	11
1	2	12

• dropna() Method:-

```
df.dropna(axis=0, how='any',
          thresh=None, subset=None,
          inplace=False)
```

• axis - 0 ('index') / 1 ('columns').

• how - 'any' - Drop if any 1 null present.

 'all' - Drop only if all null present.

• thresh - Min count of non-null values.

• subset - Limit to passed list of rows/cols.

• inplace - Permanent to original, if True.

`>>> df.dropna(axis=0) \Rightarrow Drop na rows.`

`>>> df.dropna(axis=1) \Rightarrow Drop na columns.`

`>>> subset = ['col-1', 'col-2']`

\Rightarrow consider na of these columns only.

⇒ thresh = 3 : Drop rows with less than 3 non-null values.

Non-null values should be greater than thresh values, otherwise drop.

• fillna() method - Replace the null with other values.

```
df.fillna(value, method, axis, limit,
          downcast, inplace=False)
```

- value - Value to replace with - num, str, dict, Series, DataFrame.

- method - Method to use when replacing. backfill, bfill, pad, ffill, None.

- axis - 0 ('index') / 1 ('columns')

- inplace - True - Permanent to original DF.

- limit - Max. num. of null values to fill.

- downcast - Dict of values to fill for specific data types.

- Example -

```
df['col-1'].fillna(value, inplace=True)
```


• Accessing DataFrame :-

① By Column Name -

df.col-name

df['col-name']

} Return whole col data.

>>> df.a / df['a']

O/p →

0	1
1	2

 → Data

index

Name: a, Length: 2, dtype: int64

• Give index value in [] -

>>> d.a[1] OR >>> d['a'][1]

- Return value at '1' index of col 'a'.

O/p - 2

③ Locate Rows - loc attr return 1/more rows.

↳ Also access val at index & col.

↳ Can get record at index:

df.loc[index]

df.loc[start:end]

df.loc[start:end:step]

>>> df.loc[1] → data at index '1'

O/p →

a
b

 → columns.

2

12

>>> df.loc[0:1]

O/p →

	a	b
0	1	11
1	2	12

Name: 1, dtype: int64

• loc[] with condition:

`df.loc[condition]`

```
>>> df.loc[df['a'] % 2 == 0]
```

O/p -

	a	b
[1]	2	12

• loc[] with multiple conditions:

`df.loc[(condi1) & (condi2) & (condi3)]`

```
>>> df.loc[(df['a'] == 2) & (df['b'] == 12)]
```

	a	b
1	2	12

• Access multiple columns: write column names in list

`df[[col-names-list]]`

```
>>> df[['a', 'b']]
```

③ iloc[] — Pass axes numbers for index & cols.

`df.iloc[row-index, col-index]`

```
>>> df.iloc[1:5, ]
```

1 to 4 rows & all columns.

```
>>> df.iloc[1:5, :3]
```

1 to 4 rows & first 3 cols.

• Basic Data cleaning :-

→ Fixing bad data in the dataset;

- ① Empty cells
- ② Wrong format data
- ③ Wrong data
- ④ Duplicates

~~1) Empty cells - No effect, remove some rows/large data.
df.dropna(inplace=True)~~

1) Empty cells - Result in wrong analysis. es

① Remove - Remove some rows in large data sets.

```
df.dropna(inplace=True)
```

② Fill another's value - May be mean, median, mode.

```
df['col-name'].fillna(val, inplace=True)
```

③ Replace with mean, median, mode -

```
df['col'].fillna(df['col'].mean(), inplace=True)
```

2) wrong Format data -

Ex: '2020/12/01', '10 Dec 2020', '2020-12-01'

① convert to correct format -

```
df['date'] = pd.to_datetime(df['date'])
```

convert to: 2020-12-01 format

NaT - Not a Date, for NULL values.

② Remove NULL Data -

```
df.dropna(subset=['Date', inplace=True])
```

↳ check for NULL values in 'Date' col only.

3) Fixing wrong Data -

↳ May be data not in default range.

↳ Can be outliers.

① Replace with other values -

```
df.loc[index, col] = other_value
```

↳ Replacing one value possible for few values.

② Remove rows -

```
df.drop(index, inplace=True)
```

• For large data set with many values -

```
for i in df.index:
```

```
if df.loc[i, col] condition True:
```

↳ check for condition of wrong data.

```
df.loc[i, col] = other
```

OR

~~df.loc~~

```
df.drop(i, inplace=True)
```

4) Removing Duplicates: Duplicates just ↑ size of data.

↳ Use uplicated() method for duplicate rows.

↳ Return bool for each row -

- True - Duplicate

- False - Original

↳ Remove automatically: remove_duplicates().

```
>>> df.duplicated()
```

O/p →

0	False
1	False
2	True
⋮	
n	False

dtype: bool

→ List of cols to consider

df.duplicated(subset, keep) → 'first, last, False (Delete all)

```
>>> df.drop_duplicates(inplace = True)
```

df.drop_duplicates(subset, keep, ignore_index, inplace = False)

- subset - consider these list of cols only.

- keep - 'first', 'last', False (delete all).

- ignore_index — If True: Return continuous index start from 0.

Otherwise return original, may not continuous.

- Apply filter on the Data:

1) apply() Method - `df.apply()` take funⁿ arg & apply to all values in series/col.

`df.apply(func, convert_dtype = True, args = ())`

- func - Funcⁿ to apply on data in series.
- convert_dtype - True, convert type as per operⁿ.
- args - Additional args to the funⁿ.

```
>>> d = {  
    'name': ['A', 'B', 'C', 'D', 'E'],  
    'Marks': [56, 70, 40, 80, 90]}
```

```
>>> }
```

```
>>> df = pd.DataFrame(d)
```

```
>>> def func(num):
```

```
>>>     if num > 6:
```

```
>>>         return 'Pass'
```

```
>>>     else:
```

```
>>>         return 'Fail'
```

} Funⁿ to apply on the individual data.

```
>>> df['Marks'].apply(func, convert_dtype = True)
```

<u>O/p</u> →	0	Fail
	1	Pass
	2	Fail
	3	Pass
	4	Pass

} convert marks to Pass/Fail strings.

2) map() Method:

↳ Transform values in series (col) using specified mapping data / funⁿ.

↳ Replace values from series with other series val / result of custom funⁿ.

↳ map funⁿ can take - dict, series / funⁿ.

`new-sr = old-sr.map(mapping)`

• old-sr: sr need to be transformed.

• new-sr: Dict / series / funⁿ defining transformⁿ.

① Using Dict mapping - Use dict to replace the values in series.

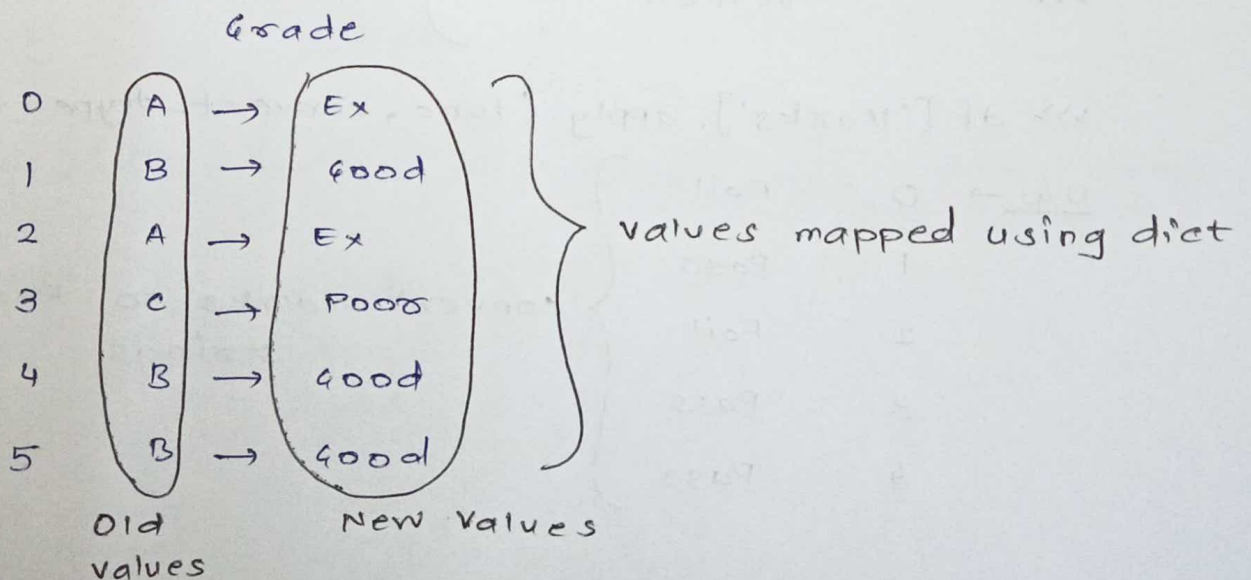
```
>>> data = {'Grade': ['A', 'B', 'A', 'C', 'B', 'B']}
```

```
>>> df = pd.DataFrame(data)
```

```
>>> grades = {'A': 'Ex', 'B': 'Good', 'C': 'Poor'}
```

↗ corresponding value to change.
↘ Keys define values in old series.

```
>>> df['Grade'] = df['Grade'].map(grades)
```



② Using Series Mapping - Use series to transform values.

→ Pass new values as data.
>>> sr = pd.Series(['Ex', 'Good', 'Poor'],
 index = ['A', 'B', 'C'])
→ Give old values to index

>>> df['Grade'] = df['Grade'].map(sr)

O/p →

	Grade
0	A → Ex
1	B → Good
2	A → Ex
3	C → Poor
4	B → Good
5	B → Good

③ Using Function for Mapping - serves same as apply().

→ Use custom funⁿ for transforming values in series.

```
>>> def transform(g):  
>>>     if g == 'A':  
>>>         return 'Ex'  
>>>     elif g == 'B':  
>>>         return 'Good'  
>>>     elif g == 'C':  
>>>         return 'Poor'
```

>>> df['Grade'] = df['Grade'].map(transform)

O/p →

	Grade
0	Ex
1	Good
2	Ex
3	Poor
4	Good
5	Good