Notes

__/__/____

Pandas

- pandas is a python package providing fast, flexible, and expressive data structures designed to make working with relational or labeled data both easy and intutive

- Data structures in pandas
  - ↳ Series (similar to 1 dimensional numpy array)
  - ↳ DataFrame (similar to 2 dim. numpy array)

- Installation command
- ! pip install pandas (Jupyter Notebook)
- pip install Pandas (command prompt)

- Importing pandas

- impost pandas as pd

Series (ndarray-like)

```
s = pd.Series(data, index = index)
```

list → data = [1,2,3,4] ← python list

dictionary → data = {'a': 1, 'b': 2} ← dictionary

numpy array → data = np.array([1,2,3]) ← numpy array

Scalar → data = 5, index = ['a', 'b', 'c']

__/__/____

- Homogenious
- One dimensional labeled array
- Holding any data type (int, str, float, date, objects etc)
- Row labels = index
- Value-mutable (the values they contain can be altered) but not Size-mutable (immutable)

Series

properties / Attributes

- Data Type : S.dtype
- Shape : S.shape
- Values : S.values
- Array : S.array

Methods

- Head : S.head( )
- Tail : S.tail( )
- Info : S.info( )
- TO numpy : S.to_numpy( )

```
S = pd.Series(data=    , index=    , dtype=    , name=    )
```

Accessing data

Indexing

- Single Index

```
S[2]   ← using index value
S['a'] ← usin row label
```

- Multiple Indexes

```
S[[1,4]] (or) S[['a','c']]
```

↳ list of indexes

slicing

```
S[start : stop : step]
```

TO slice the original string object from each row in a Series.

```
S.str.slice(start=, stop=, step=)
(OR)
S.str[start, stop, step]
```

__/__/____

DataFrame

- Heterogeneous typed columns
- Value-mutable as well as Size-mutable.
- Each column in df is Series.
- Working with Tabular data. (spread sheets stored in database also)

```
df = pd.DataFrame(data, index = idxs, columns = cols)
```

create data frame

using dictionary

```
data = {'Name' : ['Ravi', 'Shaheer'],
        'Age' : [26, 23] }
```

using Tuple

```
data = (('Ravi', 26), ('Shaheer', 23)),
columns = ['Name', 'Age']
```

using List

```
data = (['Ravi', 26], ['Shaheer', 23]), columns = ['Name', 'Age']
```

using Numpy array

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

__/__/____

DataFrame

Attributes / properties

- Shape of df : df.shape
- columns : df.columns
- data types : df.dtypes
- Axes : df.axes
- values : df.values

Methods

- Head : df.head( )
- Tail : df.tail( )
- Info : df.info( )

- Read data to pandas

- pd.read_* (* = csv, excel, sql, json, parquet etc ....)

Eg:- pd.read_csv('path')

- Export / save from pandas

- df.to_* (* = csv, excel, sql, json, parquet etc ...)

Eg:- df.to_excel('path', sheet_name='name', index=False)

Note: In pandas default Axis = 0

Axis = 0 → columnwise

Axis = 1 → Rowwise.

__/__/____

Non-visual Data Analysis (statistical)

- min `df.min()`
- max `df.max()`
- mean `df.mean()`
- median `df.median()`
- var `df.var()`
- std `df.std()`
- skew `df.skew()`
- kurt `df.kurt()`
- sum `df.sum()`
- count `df.count()`

- count `df.count()`
- nunique `df.nunique()`
- unique `df.unique()`
- value_counts `df.value_counts()`

- describe → `df.describe(include = ['object'])` ← categorical
  `(include = ['numbers'])` ← Numerical
  `(include = 'all')` ← All.

- `num_df = df.select_dtypes(include = ['int32', 'float64'])`
- `cat_df = df.select_dtypes(include = ['object'])`

- Aggregation :- `df['col-1'].agg(['min', 'max', 'mean', 'median', 'std'])`

```
df.agg({'col-1': ['min', 'max', 'mean', 'median'],
        'col-2': ['count', 'nunique', 'unique', 'value_counts']})
```

Indexing and Slicing.

- Filtering Single column vs Multiple columns :-

Syntax
- single column : df['col-name'] ← Returns Series.
  df[['col-name']] ← Returns DataFrame.
- multiple columns : df[['col-1', 'col-2', 'col-3']]

- Filtering Rows :-
- Way-1 : df[starting-row-index : ending-row-index : step]
  Eg : df[1:6]
- Way-2 : df[condition]

  Eg : df[df["max temp"] > 95]

  ** df[df['date'].isin(['10-5-2016', '10-4-2016'])]

  (OR)

  df[(df['date'] == '10-5-2016') | (df['date'] == '10-4-2016')]

- Filtering Specific Rows & columns from df :-

Syntax :

df.loc[row-label, column-label]

df.iloc[row-index, column-index]

Eg:-

df.loc[100, ['date', 'snowfall']]

df.iloc[100, [0, 5]]

__/__/____

```
df.loc[10:15, 'max temp' : 'precipitation']   (or)
df.iloc[10:15,   1 : 5]
```

- Accessing rows based on condition :

```
df.loc[condition , column_labels]
```

Eg :- `df.loc[df['max temp'] > 95 , 'date']`

- Accessing rows based on multiple conditions :

```
df.loc[(cond-1) & (cond-2) | (cond-3), column_labels]
```

Eg :- `df.loc[(df['max temp'] > 95) & (df['min temp'] > 80), 'date']`

__/__/____

Renaming columns :

Syntax → `df.rename (index = None, columns = None)`

Eg: →
```
df = df.rename (columns = {'Description' : 'product discription',
                           'customes Id' : 'cust id'})
```
↑ previous name ↑ New Name.

A very common Renaming Strategy :-

1. Strip Extra spaces
2. convert to lowercase
3. Remove all the special characters including spaces.

```
col_names = [col.strip().lower().replace(' ','_').
             replace('-','_') for col in df.columns]

df.columns = col_names
```

(or)

```
df.columns = df.columns.str.strip().str.replace(' ','_').
             str.replace('-','_').
```

Note :

By doing the above column renaming we can Access the columns like properties of the data frame like `df.columns`, `df.shape` ....

`df.col_name`.

Modifying columns Datatype :

using df.astype( )

```
df['quantity'] = df['quantity'].astype(int)
```

(or)

```
df = df.astype({'quantity': int,
                'country': str})
```

using df.apply( )

```
df['stock'] = df['stock'].apply(pd.to_numeric)
```

(or)

pd.to_datetime (or)
pd.to_timedelta

Creating a Derived column :

```
df['amount'] = df['quantity'] * df['unit_price']
```

creating columns using apply( ) function :

data frame
```
df.apply(function, axis = )
```

Series
```
s.apply(function, axis = )
```

(or)

using Anonimous function
```
df['amount'] = df.apply(lambda row: row['quantity'] *
                        row['unitprice'], axis = 1)
```

__/__/____

- Handling TimeSeries data:

pandas has Great Support for time series and has an extensive set of tools for working with dates, times and time-indexed etc.

datetime

properties

- day - df['date'].dt.day
- month - .dt.month
- year - .dt.year
- day_of_week - .dt.day_of_week
- day_of_year - .dt.day_of_year
- is_leap_year - .dt.is_leap_year
- week - .dt.week

methods

- day name - .dt.day_name()
- month name - .dt.month_name()

timedelta → difference blw two dates (or) times

properties

- days .dt.days
- hours .dt.hours
- minutes .dt.minutes
- Seconds .dt.seconds
- components .dt.components

methods

- total seconds .dt.total_seconds()

__/__/____

- pd.to_datetime()

This function helps us to convert the 'date' column to its actual data type datetime.

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

- using this function, we endup with many warnings because of format.
- These warnings are generated for a reason. Since dates can be specified in various formats for eg: DD/MM/YYYY (OR) YYYY/MM/DD (OR) MM/DD/YYYY etc.

- Two ways to get rid of warnings:

Way-1 add parameter dayfirst = True
Way-2 add parameter format = "%d/%m/%Y"

Note: Best Suggested way to convert 'date' to it's actual data type → (Recommended: Kanav sir)

df['date'] = df['date'].apply(pd.datetime())

*** ✓ df['date'] = pd.to_datetime(df['date'], dayfirst = True)
(OR)
format = "%d/%m/%Y

- While Reading / Loading data

** ✓ df = pd.read_csv(PATH, parse_dates = ['cols'])

__/__/____

```
today = pd.to_datetime('today')
df['age'] = today.year - df['DOB'].dt.year
```

+ Improve performance by Setting date column as the Index:

```
df = df.set_index(['date'])
# modifying the index inplace
df.set_index(['date'], inplace = True)
```

Select data with a specific year & perform aggregation:

```
df.loc['2018']          → Select data with specific year
df.loc['2018-5-1']      →          specific date
df.loc['2018-5-1':'2018-5-4']  →   slicing
df.loc['2018-5-1':'2018-5-5', ['sales']].mean()  →
                                   agg. with slice
```

⊛ sort_index :

```
df.sort_index(ascending = False)
```

⊛ sort_values :

```
df.sort_values(by = 'col_name')
```

⊛ Reset_index :

```
df.reset_index()
```

Intermediate pandas

__/__/____

Note: groupby( ) and pivot_table( ) are very powerful in analyzing and summarizing the data.

- Groupby :

```
grouped_df = df.groupby('category')
```
← splits based on category.

```
grouped_df = df.groupby(['category', 'sub-category'])
```
It splits the df based on 'category' first and then 'sub-category'.

* Groups :

```
grouped_df.groups
```

- The "groups" attribute is a dictionary whose keys are the computed unique Groups and corresponding values are the axis labels belonging to each group.
- Here we can apply dictionary Methods.

```
grouped_df.groups.keys()
```
→ Returns unique Group keys.

Filter data on the basis of Group keys :

```
grouped_df.get_group("Group-key-name")
```
→ Grouped on category only.

```
grouped_df.get_group(("Group-category", "Group-sub-category"))
```
→ Grouped based on category and Sub-category.

__/__/____

+ Returning First row, Last row and nth row for each group

- First row of each group :

grouped_df.first()

- Last row of each group :

grouped_df.last()

- nth row of each group :

grouped_df.nth(n) n = 1, 2, 3, 4, 5, ------

Note : The groupby method is used to support this type of operations.
This fits in the more general,
Split - apply - combine

- Split the data into groups.
- apply a function to each group independetly.
- combine the results into a data structure.

(***) - In the apply step, we might wish to do one of the following :

- Aggregation : Compute a Summary statistic for each group

__/__/____

- **Filteration :** Discard some groups, according to a group-wise computation that evaluates to True or False.

- **Transformation :** perform some group-specific computations and return a like-indexed object

Aggregation :

Built-in Aggregation methods

Eg: grouped_df['sales'].min()

any( ) – compute whether any of the values in the groups are truthy.
all( ) – compute whether all of the values in the groups are truthy.
count( ) – computes the non-NA values in the groups.

cov( )* – compute the co-variance of the Groups
first( ) – compute the first occuring value in the Group
last( ) – compute the last occuring value in each Group
nth( ) – compute the nth occuring value in each Group

idxmax( )* – compute the index of maximum value in each group
idxmin( )* – index of minimum value in each group.
min( ) – minimum value in Each Group
max( ) – maximum value in Each Group
mean( ) – mean of Each Group.
median( ) – median of Each Group.

nunique ( )	– No. of unique values in Each Group.
prod ( )	– product of values in Each Group.
quantile ( )	– a Given quantile of the values in each group.
Sem ( )	– The standard error of the mean of the values in each group.
Size ( )	– The no. of values in each group.
Skew ( )*	– The skew of the values in each group.
Sum ( )	– The Sum of the values in each group.
Var ( )	– The variance of the values in each group.
Std ( )	– The Std of the values in each group.

User-defined Aggregation

Aggregation on Single column:

```
grouped_df['sales'].agg(['min', 'max', 'mean'])
```

Different Aggregations on different columns:

```
grouped_df.agg({'sales': ['mean', 'std'],
                'order_date': ['min', 'max']})
```

__/__/____

Filteration:

Built-in Filteration

head( ) - Select the top row(s) of each group
tail( ) - bottom row(s)
nth( ) - nth row(s)

Eg: grouped_df.head( )

User-defined Filteration

grouped_df.filter(lambda group: group['Sales'].mean > 20)

Transformation:

unlike aggregations, the groupings that are used to split the Original object are not included in the result.

Built-in Transformation

bfill( ) - Back fill NA values with in each group.
cumcount( ) - The cumulative count with in each group.
cummax( ) - The cumulative max with in each group.
cummin( ) - The cumulative min with in each group.
cumprod( ) - The cumulative product with in each group.
cumsum( ) - The cumulative sum with in each group.
diff( ) - The difference b/w adjacent values with in each group.
ffill( ) - Forward fill NA values with in each group
fillna( ) - Fill NA values with in each group.

__/__/____

pct_change( ) – compute the percent change b/w adjacent values with in each group.

rank( ) – compute the rank of each value within each group.

shift( ) – shift values up (or) down with in each group.

User-defined Transformation:

grouped_df.transform(lambda x : x+1)

__/__/____

Reshaping Layout of Tables :

Q: How to reshape the Layout of tables?

A:- Change the structure of your data table in multiple ways.

- you can melt() your data table from wide to long/tidy form (or) pivot_table() from long to wide format.

+ pivot_table() - used to Summarize and Analyse data.

① df.pivot_table(values = "Sales",
index = ["region"],
aggfunc = ['sum', 'count'])

② df.pivot_table(values = 'sales',
index = ['region'],
aggfunc = ['sum'],
columns = ['category']).apply(lambda val:
val * 100 / sum(val))

category	color	Region	units	sales
Tech	Red	west	1	$11
Tech	Blue	South	8	$96
fancy	Green	West	2	$26
Tech	Blue	North	7	$84
fancy	Green	North	8	$104
fancy	Red	South	2	$22
Tech	Blue	East	5	$60
Tech	silver	East	8	$96

	Category	
Region	Tech	fancy
west	$11	$26
south	$86	$22
North	$84	$104
East	$156	0

pivot-table

__/__/____

Delete column(s) in a DataFrame :

Syntax-1 : Dropping columns by using column name:

dropping two columns by passing column names.
inplace = True parameter performs the operation saves the result back to the data frame.

```
df.drop(['col1','col3'], axis=1, inplace=True)
```

Syntax-2 : Removing columns by using columns name usig .loc[ ]:

Removing columns blw 'col2 and col4

```
df.drop(df.loc[:,'col2':'col4'], inplace=True)
```

Syntax-3 : Removing columns based on index:

Remove three columns as index base.

```
df.drop(df.columns[[0,4,2]], axis=1, inplace=True)
```

Syntax-4 : Removing columns based on index using .iloc[ ]:

Removing two columns blw column index 1 to 3.

```
df.drop(df.iloc[:, 1:3], inplace=True)
```

Syntax-5 : DataFrame.pop( ) method :

using pop( ) we can delete single column at a time

```
df.pop('col4', inplace=True)
```

__/__/____

Adding / inserting Rows ↴

Syntax-1: Inserting a single Row

```
# create a new record using Dictionary.
new_record = pd.DataFrame({'day': '1/7/2017', 'temp.': 36,
                           'windSpeed': 4, 'event': 'sunny'})
# inserting row at the end ↴
df = pd.concat([df, new_record], ignore_index = True)
# inserting row at the top ↴
df = pd.concat([new_record, df], ignore_index = True)
```

Syntax-2: Inserting multiple rows (i.e; a batch of data)

```
# create a new record(s) using dictionary.
batch_records = pd.DataFrame([{'day': '1/8/2017', 'temp': 30,
                               'windSpeed': 3, 'event': 'Rain'},
                              {'day': '1/9/2017', 'temp': 27,
                               'windSpeed': 4, 'event': 'snow'}])
# inserting at the end ↴
df = pd.concat([df, batch_records], ignore_index = True)
# inserting at the top ↴
df = pd.concat([batch_records, df], ignore_index = True)
```

__/__/____

Syntax-3 : Inserting a row using df.loc[]

```
df.loc[len(df)] = ['1/12/2017', 28, 2, 'Rain']
```

Note : using df.iloc[] we can't Insert a row.
It Generates Error - you can't use .iloc[] to Enlarge the target object (i.e; iloc[] can't be used to add new records), it can be used for updating and overwriting the Existing row.

Syntax-4 : Inserting a row at Specific index

```
# adding at row label 8.5

df.loc[8.5] = ['1/11/2017', 30, 3, 'Rain']

# Sort & Reset Index

df = df.sort_index().reset_index(drop=True)
```

__/__/____

Combining data from Multiple tables using pd.merge() function:

Remember: ① Multiple tables can be concated both column-wise and row-wise using the concat function

② For data-base like merging/joining of tables, use the merge function.

Syntax: pd.merge(left, right, how='inner', on=None, left_on=None, right_on=None, left_index=False, right_index=False, sort=False)

Note:

pandas		SQL
* By default how = 'inner'	–	Inner join
'left'	–	left outer join
'right'	–	right outer join
'outer'	–	full outer join
'cross'	–	Cross join

⊛ cross-join don't need 'On' parameter

__/__/____

Brief primes on merge methods (relational Algebra):

- Expesienced usesrs of relational data bases like SQL will be familiers with the terminology used to describe join opesations b/w two SQL-table like structuses (data frames).
- There are several cases to consider which ase very important to understand:

① one-to-one joins : for example when joining two data frame objects on their indexes (which must contain unique values)

② many-to-one joins : for example when joining an index (unique) to one or more columns in a different data frame.

③ many-to-many joins : joining columns on columns.

Visual Data Analysis using pandas (plotting):

Syntax: df.plot(kind = "kind")

The kind of plot to produce:

① line : line plot (default 'kind')
② bar : vertical bar plot
③ barh : Horizontal bar plot
④ hist : Histogram
⑤ box : Box plot
⑥ kde : Kernel Density Estimation plot
⑦ density: Same as "kde"
⑧ area : area plot
⑨ pie : pie plot
⑩ scatter: scatter plot
⑪ hexbin: Hexbin plot.

Note: ① whenever we draw 'bar' plot / categorical column follow the syntax below ↓

df['species'].value_counts().plot(kind = "bar")

__/__/____

combining data from multiple tables using pd.merge()

Examples:

left

	key1	key2	A	B
0	K0	K0	A0	B0
1	K0	K1	A1	B1
2	K1	K0	A2	B2
3	K2	K1	A3	B3

right

	K_1	K_2	C	D
0	K0	K0	C0	D0
1	K1	K0	C1	D1
2	K_1	K0	C2	D_2
3	K2	K0	C3	D_3

pd.merge(left, right, how='inner', on=['key1','key2'])

Inner Join:

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D_2

Left Join: how='left'

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D1
3	K1	K0	A2	B2	C2	D_2
4	K2	K1	A3	B3	NaN	NaN

Right Join: how='right'

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K1	K0	A2	B2	C1	D1
2	K1	K0	A2	B2	C2	D2
3	K_2	K0	NaN	NaN	C3	D3

__/__/____

Outer Join : how = 'outer'

	key1	key2	A	B	C	D
0	K0	K0	A0	B0	C0	D0
1	K0	K1	A1	B1	NaN	NaN
2	K1	K0	A2	B2	C1	D_1
3	K1	K0	A2	B2	C2	D_2
4	K2	K1	A3	B3	NaN	NaN
5	K2	K0	NaN	NaN	C_3	D_3

Cross Join : how = 'cross'

Note : for cross join "on" parameter is not required

	key1_x	key2_x	A	B	key1_y	key2_y	C	D
0	K0	K0	A0	B0	K0	K0	C0	D0
1	K0	K0	A0	B0	K1	K0	C1	D1
2	K0	K0	A0	B0	K1	K0	C2	D2
3	K0	K0	A0	B0	K2	K0	C3	D3
4	K0	K1	A1	B1	K0	K0	C0	D0
5	K0	K1	A1	B1	K1	K0	C1	D1
6	K0	K1	A1	B1	K1	K0	C2	D2
7	K0	K1	A1	B1	K2	K0	C3	D3
8	K1	K0	A2	B2	K0	K0	C0	D0
9	K1	K0	A2	B2	K1	K0	C1	D1
10	K1	K0	A2	B2	K1	K0	C2	D2
11	K1	K0	A2	B2	K2	K0	C3	D3
12	K2	K1	A3	B3	K0	K0	C0	D0
13	K2	K1	A3	B3	K1	K0	C1	D1
14	K2	K1	A3	B3	K1	K0	C2	D2
15	K2	K1	A3	B3	K2	K0	C3	D3

__/__/____

pd.melt( ):

```
data = {'Name': ['Ravi', 'Shaheer'],
        'math': [90, 95],
        'English': [85, 88]}

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

Name	math	English
Ravi	90	85
Shaheer	95	88

```
melted_df = pd.melt(df, id_vars=['Name'],
                    var_name=['subject'],
                    value_name='Score')
print(melted_df)
```

	Name	Subject	Score
0	Ravi	Math	90
1	Shaheer	Math	95
2	Ravi	English	85
3	Shaheer	English	88

__/__/____

- pd.explode( ) :

```
data = {'Name': ['Ravi', 'Abhi'],
        'Subjects': [['Math', 'English'], ['Science', 'History']]}

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

Name	Subjects
Ravi	['Math', 'English']
Abhi	['Science', 'History']

```
exploded_df = df.explode('Subjects')
print(exploded_df)
```

	Name	Subjects
0	Ravi	Math
1	Ravi	English
2	Abhi	Science
3	Abhi	History.