DATE: ☐☐ ☐☐ ☐☐☐☐

Global Transformation of Series by using map() method:-

Syntax: Series.map(arg, na_action=None)

eg: s=pd.Series(['cat', 'dog', np.nan, 'rat'])
s1 = s.map({'cat': 'kitten', 'dog': 'puppy'})
print(s1)

→ Iterating Elements of the Series:- Iterating means getting element one by one.

s=pd.Series([10,20,30,40])
for i,v in s.iteritems():
    print(i, '--->', v)

Chapter-2
Data Frame

Data-Frame is a labeled 2-D array. It Contains multiple rows & Columns.

eg: eno = [10,20,30,40]
ename = ['A', 'b', 'c', 'd']
esal = [1000, 2000, 3000, 4000]
eaddrs = ['Hyd', 'bang', 'Chen', 'ker']
df = pd.DataFrame({'eno': eno, 'ename': ename, 'esal': esal, 'eaddrs': eaddrs})
print(df)
print(type(df))
print(df.ndim) print(df.shape)

DATE: □□/□□/□□□□

Conclusion:

① Series is a labeled 1-D array where as DataFrame is labeled 2D array.
② Each column of Dataframe acts as Series. Hence we can consider DataFrame is nothing but collection of Series objects.
③ Series object holds only homogeneous values. But DataFrame contains multiple columns & each column values may be of different types. Hence DataFrame holds heterogenous data.

Syntax: pd.DataFrame(data=None, index=None, columns=None, dtype=None, copy=None)

Data-Frame is a Two-dimensional, size-Mutable, potentially heterogeneous tabular data.

Various ways to Create dataframe object:

① From dict of List objects. (Common way of creating data-frames)
② From dict of Tuple objects.
③ From dict of Series objects.
④ From dict of dicts.
⑤ from Numpy ndarray.

Note: 'dtype' attribute only applicable for Series object but not for dataframe. We have to use 'dtypes' attribute for dataframe object

DATE: ☐☐☐☐☐☐☐☐

→ Creating Dataframe from numpy ndarray :-

```
df = pd.DataFrame(nparray([1,2,3],[4,5,6]),
                  columns=['a','b','c'])
print(df)
```

→ Common Methods & Attributes of Series & DataFrame :-

Most of methods & attributes of Series are applicable for DataFrames also.

eg: ① df.head() ③ df.shape() ⑤ df.index() ⑦ df.dtypes
② df.tail() ④ df.ndim() ⑥ df.values() ⑧ df.dtypes.value_counts()
↓
Returns the no & columns with each data type.

Attributes & Methods applicable only for DataFrame but not for Series:-

① info() method :- It generates summary information of DataFrame.

syntax: DataFrame.info(verbose=None, buf=None, max_cols=None, memory_usage=None, show_counts=None, null_counts=None)

② Sample() Method :- We can use sample() method to get a random records from the DataFrame.

Syntax:- DataFrame.sample(n=None, frac=None, replace=False, weights=None, random_state=None, axis=None, ignore_index=False)

Returns a random sample of elements (or) items from a axis object.

eg: print(S.sample())
print(S.sample(n=2))
print(S.sample(frac=0.1))

DATE: □□ □□ □□□□

③ Columns attribute :- Returns column labels of the DataFrame.
eg: print(df.columns) → It is Index object contain column names.

④ axes attribute :- Returns a list representing the axes of the DataFrame.
eg: print(df.axes)

→ How to select only one required column of Data-Frame

1st way → df.Column_name
2nd way → df['column_name'] → 2nd way is recommended.

eg: print(s['Names'])
print(s['Marks'])
print(s[['Names','Marks']]) → Returns DataFrame

→ Arithmetic operations for DataFrame :-

We cannot apply airthmetic operations directly on Dataframe object because it contains multiple columns.

We can perform arithmetic operations by selecting a particular column of the DataFrame, because each column of a DataFrame is a Series object.

eg: print(s['Marks']+500) → 500 will be broadcasted to every value of the Series.
↳ (or) print(s['Marks'].add(500))

DATE: □□/□□/□□□□

→ Adding New Columns to the DataFrame

1st way : df['Column'] = value → It will be added at last.

2nd way : df.insert(1, 'Country', 'India') → It will be added at specified alon

↳ Syntax → DataFrame.insert(loc, column, value, allow_duplicates=False)

eg: ① S.insert(1, 'ESAL', 999)

② S.insert(1, 'Country', ['A', 'B', 'C', 'D'])

→ How to drop DataFrame rows with missing values/null values?

We have to use dropna() method

Syntax: DataFrame.dropna(axis=0, how='any', thresh=None, subset=None, inplace=False)

Remove missing values. Removes rows from the dataFrame where atleast one missing value.

eg: s=pd.read_csv('emp.csv')

S1 = S.dropna() s=s.dropna(how='all') default = any

Print(S1)

any → If any NA values are present, drop that row (or) column

all → If all values are NA, drop that —/—

Subset → To delete rows if a particular Column has missing values.

S1=S.dropna(subset=['esal'])

axis → To delete column which have miseing values.

DATE: □□/□□/□□□□

→ How to fill missing/Null values with our required values in DataFrame.
we have to use fillna() method for this requirement.

Syntax :- DataFrame.fillna(value=None, method=None, axis=None, inplace=False, limit=None, downcast=None)

eg: df=S.fillna(0)
print(df) → Replacing missing values of string with '0' is not meaning full. So for different data types we have to fill the default value.

df['Ename']=S['Ename'].fillna('Suhas') → df['Esal']=df['Esal'].fillna(0)

If Column with different types :-
we can provide different replacement values for different Columns based on Column data type. We have to use astype() method.
For this we have to dropna() → to remove null values.

eg: S=pd.read_csv('emp.csv')
print(S)
S.dropna(inplace=True)
S['Esal']=S['Esal'].astype('int')
print(S)

→ How to Sort values of the DataFrame :-
We can use Sort_values() method.

Syntax :- DataFrame.sort_values(by, axis=0, ascending=True, inplace=False, key=None, kind='quicksort', na_position='last', ignore_index=False)

DATE: ☐☐ ☐☐ ☐☐☐☐

We can use 'by' parameter to specify column based on which sorting has to be done.

eg:- S.sort_values(by='Ename', inplace=True)
print(s)

eg:2 :- S.sort_values(by='EName', inplace=True, ascending=False, na_position='first')

eg:-3 :- S.sort_values(by=['Ename', 'Esal',], ascending=[False, True])

→ How to Sort DataFrame by index:

Syntax: DataFrame.sort_index(axis=0, level=None, ascending=True, inplace=False, key=None, kind='quicksort', na_position='last', ignore_index=False

eg:- S.sort_index(inplace=True, ascending=False)

→ How to rank Series of values by using rank() method :-

Syntax: Series.rank(axis=0, method='average', ascending=True)

```
df=pd.read_csv('emp.csv').dropna(how='all')
df['Esal']=df['Esal'].fillna(0).astype('int')
print(df)
df['Salary Rank']=df['Esal'].rank(ascending=False).astype('int')
print(df)
df.sort_values(by='Salary Rank', inplace=True)
print(df)
```

→ Default way to remove na & fill esal of na to 0.

DATE: □□/□□/□□□□

→ How to filter data from Dataframe :-

- We can filter data based on some condition, ie by using subset of rows based on some selection criteria. Internally we can use Boolean masking for this filtering purpose.

Syntax: df[boolean masked array]

Case-1: Select employees belong to bangalore

print(df['Eaddrs']) → (df['Eaddrs']=='Bangalore') → print(df[df['Eaddrs']=='Bangalore'])

Case-2 :- Select salary >5000

print(df[df['Esal'] > 5000]) similarly we can use <, <=, >=,

→ How to filter data from DataFrame based on Multiple conditions :-

We can use & and | operation to combined multiple conditions together.

Syntax: df[C1 & C2]

df[C1 | C2]

Case 1: To Select employee belongs to banglore & Salary > 5000.

C1 = df['Eaddrs'] == Bangalore

C2 = df['Esal'] > 5000

print(df[C1 & C2])

Case-2 :- Select employee who should be either bangalore base who salary > 3000 (or) mumbai based employee.

- C1 = df['Eaddrs'] == 'Banglore' C2 = df[Esal'] > 300

C3 = df['Eaddrs'] == 'Mubai'

print(df[(C1 & C2) | C3])

DATE: □□/□□/□□□□

→ How to filter data from Dataframe :-

- We can filter data based on some condition, i.e by using subset of rows based on some selection criteria. Internally we can use Boolean masking for this filtering purpose.

Syntax: df[boolean masked array]

Case-1: Select employees belong to bangalore

print(df['Eaddrs']) → (df['Eaddrs']=='Bangalore') → print(df[df['Eaddrs']=='Bangalore'])

Case-2 : Select salary >5000

print(df[df['Esal'] > 5000]) Similarly we can use <, <=, >=,

→ How to filter data from DataFrame based on Multiple conditions :-

We can use & and | operation to combined multiple conditions together.

Syntax : df[C1 & C2]

df[C1 | C2]

Case 1 : To Select employee belongs to banglore & Salary > 5000.

C1 = df['Eaddrs'] == Bangalore

C2 = df['Esal'] > 5000

print(df[C1 & C2])

Case-2 : Select employee who should be either bangalore base who salary > 3000 (or) mumbai based employee.

C1 = df['Eaddrs'] == 'Banglore' C2 = df[Esal'] > 300

C3 = df['Eaddrs'] == 'Mubai'

print(df[(C1 & C2) | C3])

DATE: ☐☐/☐☐/☐☐☐☐

→ How to check inclusion by using isin() method :-

Eg: To select employees who are belongs to Hydrabad, Mumbai & Delhi?

Without using isin method

```
C1 = df['Eaddrs'] == 'Hydrabad'
C2 =  —  /  —     'Mubai'
C3 =  —  /  —     'Bangalore'
print(df[C1 | C2 | C3])
```

with using isin() method

```
C1 = df['Eaddrs'].isin(['Hyd', 'Mum', 'Ban'])
print(df[C1])
```

Syntax df.isin(values)

Note: To ignore Case :- `C1 = df['Eaddrs'].str.lower().isin(['Hyd', 'Ban'])`

→ How to check for inclusion by using between() method :-

Without using between() method

```
C1 = df['Esal'] >= 2500
C2 = df['Esal'] <= 10,000
print(df[C1 & C2])
```

With using between() method

```
C = df['Esal'].between(2000, 5000)
```

case-2: inclusive : { 'both, neither, left, right'

```
C = df['Esal'].between(2000, 5000, inclusive = 'both')
```

→ How to Check missing & non-missing Values by using isnull() & notnull() methods :-

Series.isnull() → Detect missing values.

Series.notnull() → Detect existing (non-missing) values.

DATE: □□ □□ □□□□

eg: C=df['Ename'].isnull()
print(df[C])

C=df['Ename'].notnull()
print(df[C])

→ How to check duplicate dataframe rows by using duplicated() method :-

eg: C=df['Ename'].duplicated()
print(df[C])

eg2:- C=df['Ename'].duplicated(keep=False)

eg3:- C=~df['Ename'].duplicated(keep=False)

→ How to drop duplicate rows in DataFrame by using drop_duplicates() :-

Syntax:- df.drop_duplicates(subset=None, keep='first' inplace=False, ignore_index=False)

eg: df.drop_duplicates(subset=['Ename'], keep=False, inplace=True)

→ How to identify & count uniane values in DataFrame :-

S.uniane() → returns uniane values.

S.nuniane() → number of values uniane by ignoring Nan.

S.nuniane(dropna=False) → number of uniane values without ignoring Nan.

eg: df['Ename'].uniane()

→ Extracting Values from the DataFrame :-

① How to customize column as index

1st way :- df=pd.read_csv('emp.csv', index_col='Ename')

2nd way :- df.set_index(by='Ename', inplace=True)

DATE: ☐☐ ☐☐ ☐☐☐☐

→ How to reset index from custom index to default numerical index :-
df.reset_index(inplace=True)

→ How to select rows by index label with loc[ ] indexer :-
Syntax- df.loc[index_label] ∴ df = pd.read_csv('emp.csv', index_col='Ename')
① df.loc['suhas'] → only one row matched & returns in the form of Series.
if multiple rows matched returns Dataframe
if specified index not matched we get keyerror.
② Using slice operator :- df.loc[index_label1 : index_label2 : step]
Here both Begin & end index labels are inclusive.
eg: print(df.loc['Suhas' : 'Ramesh' : 2])
③ With multiple index labels of df.loc[[index1, index2, index3]]

→ How to Access Rows of DataFrame by using index position :-
We have use by iloc[ ] indexer.
① df = pd.read_csv('emp.csv', index_col='Ename')
print(df.iloc[0])
② by using slice operator :
df.iloc[2:5:2]
③ with multiple indexpositions :- df.iloc[[index_1, index_2, index]]

→ How to get values of the required cells :-
Syntax :- df.loc[index_label, column_name] → eg: print(df.loc['suhas', 'Esal'])
df.iloc[index_position, column_name]

DATE: ☐☐ ☐☐ ☐☐☐☐

Case study: ① df.loc['sunny', 'Esal'].iloc[0]

→ How to set new value of the specified cells:

Syntax: df.loc[index_label, col-name] = new value

df.iloc[index_position, col_name] = new value

Eg: ① df.loc['Sidhas', 'Esal'] = 777 ② df.loc[['A', 'B', 'c'], 'Eaddrs'] = Bang

print(df) or print(df.loc['Sidhas'])

eg: 3: Replace every occurance of Hyderabad to Cyberabad

C = df['Eaddrs'] == 'Hyderabad'

df.loc[C, 'Eaddrs'] = 'cyberabad'

→ How to rename index labels & column names in a DataFrame:-

By using rename() method For index labels (rows) → axis 0

For column names → axis 1

Renaming index labels in Two ways: ① By using mapper & axis Parameters

② By using index parameter.

Syntax: DataFrame.rename(mapper=None, index=None, columns=None, axis=None, copy=True, inplace=False)

eg: ① df.rename(mapper={'Sunny': 'leone', 'Bunny': 'Arjun'}, axis=0, inplace=True)

② df.rename(index={'Sunny': 'leone', 'Bunny': 'Arjun'}, inplace=True)

DATE: ☐☐☐☐☐☐☐☐

→ Renaming Column Names :- ① By using mapper & axis parameters
② By using Column parameter

① eg: df.rename(mapper={'Esal':'salary', 'Eaddrs':'city'}, axis=1, inplace=True)

② df.rename(Columns={ ——— , ——— }, inplace=True)

→ How to delete rows & Columns from the DataFrame :-
By using drop() method.

Syntax: df.drop(labels=None, axis=0, index=None, Column=None, inplace=False)

eg: df.drop(labels='Sunny', inplace=True) → For rows default axis=0
df.drop(labels='Esal', axis=1, inplace=True) → To delete columns.

→ How to filter DataFrame with query() method :-
We can use query() method to filter rows from the DataFrame. It is the most commonly used method in pandas.

eg: df.query(expr, inplace=False, **kwargs)

While using query() method the following 2 points are important :-
① expr parameter value should be a valid string.
② The Column names should not Contain spaces.

eg: ① df.query("Eaddrs == 'Hyderabad'", inplace=True)
print(df)

② df.query("Esal > 1500", inplace=True)

DATE: ☐☐☐☐☐☐

③ To handle multiple conditions :- (we can use Error!)

df.query("Eaddrs == 'Bangalore' and Esal > 1500", inplace=True)

④ in & not in operators in query() method :-

df.query("Eaddrs in ['Hydrabad', 'Bangalore']", inplace=True)
(not in)

Note: If column names contain space then query() method won't work.

How to resolve this ? → Replace space with underscore (_) symbol & then apply query() method.

df.columns = [col_name.replace(' ', '_') for col_name in df.columns]

df.query("Total_Salary > 1500", inplace=True)

print(df)

→ How to use apply() method for DataFrame rows :-

It is exactly same as Series Class apply() method except that it is applicable row wise.

df.apply(func, axis=0, raw=False, result_type=None, *args, **kwargs)

Eg: Double the Salary of each employee:

df['Esal'].apply(lambda x: 2*x, inplace=True)

DATE: □□□□□□□□

→ How to get random sample of rows from dataFrame() :-
By using sample() method.

Df.sample(n=None, frac=None, replace=False)

Eg: ① print(df.sample(2)) ② print(df.sample(n=2, axis=1))

→ How to use nlargest() method of the dataframe

Eg: df.nlargest(n=3, columns=['Esal'])

→ How to use nsmallest() method:

eg: df.nsmallest(n=3, columns=['Esal']))

→ How to filter DataFrame with where() method :-

Syntax: df.where(Cond, other=nan, inplace=False, axis=None)

Replaces values where Condition is False., i.e it returns Full dataFrame where original data will be considered if Condition is True & Nan will be considered if Condition is False.

eg:- To select rows where Eaddrs is Hyderabad?

df = pd.read_csv('emp.csv', index_col='Ename')

C = df['Eaddrs'] == 'Banglore'

print(df.where(C)) (or) print(df.where(C, other='Not usefull'))

Note: Usage of where() method on the Series object is more meaningful than DataFrame.

DATE: □□ □□ □□□□

→ How to get Seperate Copy of the DataFrame :-

To create Seperate independent isolated copy of dataFrame (or) Series, we have to use copy() method.

df1 = df.copy() → If we perform any changes to the original Object, then those changes won't be reflected to the copy.

→ How to handle text data with String methods :-

The most Commonly used data type in any programming language is text data (Str type). To perform Common requirement to handle text data like removing spaces, Converting into required Case etc.

Pandas library provides the following methods to handle text data :-

① lower() ② title() ③ replace (old-chr, new-chr) ④ strip() → lstrip() rstrip()
⑤ upper() ⑥ len() etc

While using these methods we should use 'str' prefix.

① ex : ① Convert all employees into uppercase :-

df['Ename'] = df['Ename'].str.upper()

② To remove $ Symbol in Esal :-

df['Esal'] = df['Esal'].str.replace('$', '')

③ To remove $ & , Symbols in Esal & then covert values into float type :-

df['Esal'] = df['Esal'].str.replace('$', '').str.replace(',', '').astype(float)

④ To add new column 'Number of Characters' which is evaluated from Ename value :-

df['Numer of Char'] = df['Ename'].str.len()

DATE: ☐☐/☐☐/☐☐☐☐

Case study: Convert Esal into proper numeric values:

Esal
15K
20K
30K
40L
3C

eg: df = pd.read_csv('emp2.csv')

Convertion_rate = {'k': 1000, 'L': 100000, 'C': 10000000}

def Convert(value):

unit = value[-1] → i(5k) (0 1 2)

numeric = value[1:-1] $200K (0 1 2 3 4, -1)

updated_value = float(numeric) * Convertion_rate[unit]

return '$' + str(updated_value)

df['Esal'] = [Convert(sal) for sal in df['Esal']]

Print(df)

→ How to filter DataFrame rows with String Methods :-

Contains(), startswith(), endswith().

① Contains(sub_string) → Returns True if the string Contains provided sub_string irrespective of position.

② startswith(substring) :- Returns True if the string starts with provided sub_string

③ endswith(sub_string) :- —/— String ends with —/—

eg: ① To filter all rows where ename starts with 'A' ?

C = df['Ename'].str.startswith('A') → Case Sensitive.

C = df['Ename'].str.lower().str.startswith('a')

DATE: ☐☐ ☐☐ ☐☐☐☐

② To filter all rows where ename contains 'a'?

C = df['Ename'].str.lower().str.contains('s')

③ Ends with y :- C = df['Ename'].str.lower().str.endswith('y')

→ How to use String split() method :-

We can use split() method to split the given string based on some condition.

eg: Consider the Ename column of DataFrame & split every name to two names - First Name & Last Name.

```
df = pd.read_csv('emp2.csv')
df.dropna(inplace = True)
df.insert(loc = 1, column = 'First Name', value = df['Ename'].str.strip().str.split(expand=True).get(0).str.title())
df.insert(loc=2, column = 'Last Name', value = df['Ename'].str.strip().str.split(expand=True).get(1).str.title())
del df['Ename']
print(df)
```

eg: Ename = 'Sunny Leone Pande' (n parameter)

df['Ename'].str.split() → ['sunny', 'Leone', 'Pande']

df['Ename'].str.split(n=1) → ['sunny', 'Leone Pande']

DATE: ☐☐☐☐☐☐☐☐

→ How to Concatenate, merge & join multiple DataFrames into a SingleDataFrame.

Syntax: pd.concat(objs, axis=0, join='outer', sort=False)

eg: df1 = pd.read_csv('emp1.csv')
df2 = pd.read_csv('emp2.csv')
combined = pd.concat(objs=[df1, df2])
print(combined)

→ How to remove duplicate index labels :- Two ways to remove duplicate index

1st way: by using reset_index() method

df.reset_index(level=None, drop=False, inplace=False, col_fill='')

eg: combined.reset_index(inplace=True)
print(combined)

2nd way: Ignore_index parameter of concat() method

eg: combined = pd.concat(objs=[df1, df2], ignore_index=True)
print(combined)

→ Defining Multiindexing By using keys parameter.

eg: Combined = pd.concat(objs=[df1, df2], keys=['month1', 'month2'])

→ How to access data if we are using MultiIndex :

eg: print(combined.loc[('month1', 1)])

⇒ To enforce uniaue index labels →
By using verif_integrity=True

DATE: ☐☐ ☐☐ ☐☐☐☐

→ How to Concatenate Columns

```
Combined = pd.concat(objs=[df1, df2], axis=1)
print(combined)
```

→ Merge Method :- It is most powerful than concat() method. It's functionality is exactly same as SQL Join queries. Grouping can be done based on matched content.

Syntax :- `pd.merge(left, right, how='inner', on=None, right_on=None, sort=False)`

Eg:- To select all students who registered for the course in 1 month :-

```
df1 = pd.read_csv('month_1 reg.csv')
df2 = pd.read_csv('students.csv')
S = pd.merge(df1, df2)
print(S)
```

Eg: `S = pd.merge(df1, df2, on='student_id', suffixes=('_month_1', 'month_2'))`

Eg: If there is no Common Column name :-

If there is no common name, merge operation fails by default. To Overcome this problem, we have to use left_on & right_on parameters

Eg:- `df1 = pd.read_csv('month_1reg.csv')` → student id, Cource id

`df2 = pd.read_csv('courses.csv')` → CId, Cource Name, Fee

`S = pd.merge(df1, df2, left_on='Cource id', right_on='CID')`

`S.drop('CID', axis=1, inplace=True)` in df1

In df2

DATE: ☐☐ ☐☐ ☐☐☐☐

→ Merge Outer Join :- how = 'outer' → All keys are selected. similar to set union operation. If data present in both dataFrames is important & if we don't want to miss any data then we should go for outer Join.

eg:- df1 = pd.read_csv('month_1_reg.csv')
df2 = pd.read_csv('courses.csv')
s = pd.merge(df1, df2, how='outer') → default is 'inner Join'
print(s)

Indicator parameter :- indicator :- bool (or) str, default False
If True, adds a column to the output DataFrame called '_merge' with information on the source of each row.

s = pd.merge(df1, df2, how='outer', indicator=True)

How to find source count of rows:
print(s['_merge'].value_counts())

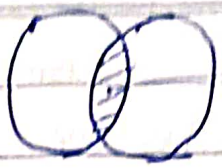
eg:- Select only rows present in either left (or) Right DataFrame but not both

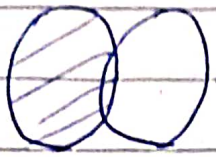
w = s['_merge'].isin(['right_only', 'left_only'])
print(s[w])

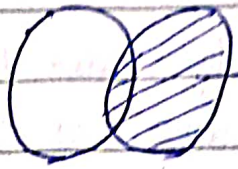
→ Left Join :- Data present in Left DataFrame is very important & we have to consider complete information, but right DataFrame is not important we have to consider only matched content, then we should go for left Join. eg:- df1 = pd.read_csv('month_1_reg.csv')
df2 = pd.read_csv('students.csv')
s = pd.merge(df1, df2, how='left')

DATE: ☐☐☐☐☐☐☐☐


→ Outer


→ Inner


→ left


→ Right.

Right Join:

```
df1 = pd.read_csv('month_1 reg.csv')
df2 = pd.read_csv('students.csv')
s = pd.merge(df1, df2, how='right')
s.to_csv('complete.csv') → To write new csv file.
```

Syntax:: df.merge(right, how='inner', on=None, right_on=None, left_on=None, left_index=False, right_index=False, suffixes=('_x','_y'), copy=True, indicator=False, validate=None)

On → Based on which column, merge has to perform.

left_on → Based on which column in left dataFrame, merge has to perform

right_on → ——— / ——— right ——— / ———

DATE: ☐☐ ☐☐ ☐☐☐☐

→ join() method: If we want to perform merge operation based on indexes of both DataFrame then we should use join() method as shortcut technique.

pd.merge(df1, df2, left_index=True, right_index=True)

We can rewrite this line as: df1.join(df2).

→ How to use Regular Expressions in Pandas:-

If we want to represent a group of characters according to a particular pattern then we should go for regular expressions.

predefined character classes:-

\s → Space character	\d → Any digit
\S → Any character except space character.	\D → Any character except digit
\w → [a-zA-Z0-9]	. → Any character.
\W → Any character except word character	

Quantifiers:-

a → Exactly one 'a'	a{n} → Exactly n number of a's
a+ → Atleast one 'a'	a{m,n} → minimum m of a's & max n a's
a* → Any number of a's including zero	a{m,} → minimum m number of a's & max any of a's
a? → Atmost One 'a'	a{,n} → minimum any nuber of a's & max n number of a's.

DATE: □□/□□/□□□□

replace(), contains() & split() methods with regular expressions :-

Syntax: Series.str.replace(pat, repl, n=-1, case=None, flags=0, regex=None)

eg: To remove $ & , symbols in Esal column values & then convert values into float type:

eg:- df['Esal'] = df['Esal'].str.replace(r'[$,]', '', regex=True).astype(float)

Contains():

syntax:- series.str.contains(pat, case=True, flags=0, na=None, regex=True)

eg:- C = df['Ename'].str.contains(r'[kmj]', case=False, regex=True)
print(df[C])

Split():-

Syntax: Series.str.split(pat=None, n=-1, expand=False)

split key column into 3 parts: k1, k2, k3

df[['k1', 'k2', 'k3']] = df['key'].str.split(pat=r'[_:]', expand=True)
del df['key']
print(df)

DATE: □□□□□□□□

Working with MultiIndex :- Multi level Index (or) Hierarchical Index.

Sometimes One level index is not enough to access data from the DataFrame. We should go for multiple levels of Indexing which is nothing but MultiIndex.

```
eg: df=pd.read_csv('multi_indexing.csv')
    print(df)
    Print(df.index)   #RangeIndex(start=0, stop=8, step=1)
    print(df.Columns)
```

RangeIndex is the child class of Index. → If we are not providing explicit index Pandas will generate default numeric index which is of type RangeIndex & it is immutable.

→ How to set Our own Column as Index Column

① By using set_index() method → df.set_index(keys='Date', inplace=True)

② By using index_col parameter. → df=pd.read_csv('emp.csv', index_col='Date')

How to define Multi Indexing :- We have to pass list of Column names.

① df.set_index(keys=['Date', 'Domain'], inplace=True)

② df.read_csv('emp.csv', index_col=['Date', 'Domain'])

eg: df=pd.read_csv('emp.csv', index_col=['Date', 'Domain'])

Note:- RangeIndex → Default Numeric Index

Index → Column based Index.

DATE: □□/□□/□□□□

→ How to access data from Multi Indexed DataFrame

There are multiple ways available to extract data :-

① By using loc indexer with index labels :

loc → Label based selection.

iloc → Position based selection.

ex:1:- To get all batches info on '25-01-2023'? → ie access data by using first level index :-

```
df = pd.read_csv('multi_courses.csv', index_col = ['Date', 'Subject'])
print( df.loc[('25-01-2023')])
```

ex-2: To get Java batch information on '25-01-23'?

```
print(df.loc[('25-01-2023', 'Python')])
```

ex:3: To get Fee info of 'Java' batch on 25-01-23 & Timing.

```
print(df.loc[('25-01-2023', 'Python'), 'Fee'])  (or)
print(df.loc[('25-01-2023', 'Python'), ['Fee', 'Timing']])
```

→ How to access with Range of indexes :-

eg:- To get python batch information from '25-01-23 to '28-01-23'?

```
print(df.loc[([('25-01-2023', '26-01-2023')])])
```

ex : 2 :-

```
print(df.loc[(['25-01-23', '26-01-2023'], ['SQL']), :])
```

→ The total tuple is considered as index labels only

: represents all column names.

DATE: ☐☐☐☐☐☐☐☐

eg:3: print(df.loc[(['01-01-22', '06-01-2022'], 'Java'), ['Faculty', 'Timing']])

Syntax: df.loc[(),[]]
() → index labels
[] → Column names.

Note: It is very common & very easy to use loc indexer when compared with iloc indexer.

eg:4: To get Python & Java Batches Fee & Faculty information are of '24-01-22' and '25-01-22' ?

print(df.loc[(['24-01-22', '25-01-22'], ['Java', 'Python']), ('Fee', 'Faculty')])

~~→ How to perform slice operation:-~~

→ How to get cross section from the DataFrame by using xs() method:
xs → Cross section.

Syntax: df.xs(key, axis=0, level=None, drop_level=True)
Returns cross-section from one Series/DataFrame.

eg: df=pd.read_csv('multi_courses.csv', index_col=['Date', 'Subject'])
print(df.xs('25-01-2023'))

eg: To get Java Batches info for all dates:
print(df.xs('Java', level=1, drop_level=False))

DATE: ☐☐/☐☐/☐☐☐☐

Eg. To get Java info on '28-01-2023'?
print(df.xs('28-01-2023', 'Java'), level=(0,1), drop_level=False))

→ The attributes of MultiIndex Object:
We can use multiple attributes on MultiIndex object. These are very helpful if ~~the~~ are working on very complex hierarchical DataFrames.

Eg: df = pd.read_csv('new_batches.csv', index_col=['Date', 'Subject'])
print(df) print(type(df))
print(df.index) print(type(df.index))

① names: print(df.index.names) List of names for each of the index levels.
print(df.index.names[0])

② nlavels: Integer number of levels in the multiinde
print(df.index.nlevels)

③ Levels: The uniare labels for each level:-
print(df.index.levels)

④ levshape: A tuple with the length of each level.
print(df.index.levshape) → (5, 4)
∴ First level index contains 5 values, 2nd level index Contains 4 levels.

⑤ Size: (df.index.size)

⑥ values: (df.index.values)

DATE: □□□□□□□□

→ How to add another level of the Multi-Index :-

1st way: Read again data & Create dataFrame

df = pd.read_csv('emp.csv', index_col = ['Date', 'Subject', 'Faculty'])

2nd way: If DataFrame already available.

If DataFrame already available, then we should add new level of index by using set_index() method.

Syntax: df.set_index(keys, drop=True, append=False, inplace=False)

eg: df = pd.read_csv('emp.csv', index_col = ['Date', 'Subject'])
df.set_index(keys = 'Faculty', append=True, inplace=True)
print(df) print(df.index.names)

→ How to swap index levels: In Two ways:

① By using index position: To swap any two levels

Syntax: df.swaplevel(i=-2, j=-1, axis=0)

eg: df1 = df.swaplevel() → innermost 2 level i.e last 2 levels will be swapped
df2 = df.swaplevel(0,2) → We can use either
df3 = df.swaplevel('Date', 'Subject') index position (or) Index label

② By using reorder_levels(): To swap more than 2 levels.

Syntax: df.reorder_levels(order, axis=0)

eg: df1 = df.reorder_levels(order = [2, 0, 1])
df2 = df.reorder_levels(order = ['Subject', 'Faculty', 'Date'])

DATE: □□/□□/□□□□

→ How to remove existing index level from MultiIndexed DataFrame:
By two ways : ① By using droplevel() method. (not recomended)
② By using reset_index() method.

① By using droplevel() :-

```
df=pd.read_csv('multi_courses.csv',index_col=['Date','Subject'])
df1 = df.droplevel(level='Subject')
print(df1)
```

Not recommended to use droplevel() because there may be a chance of loss of information.

② By using reset_index() method :-

Syntax :- df.reset_index(level=None ,drop=False, inplace=False, col_level=0)

eg: df = pd.read_csv('emp.csv', index_col=['Date','Subject','Faculty'])

```
df.reset_index(level=[0,2],inplace=True)
print(df)    print(df.index_names)
```

To remove all indexes

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

DATE: □□ □□ □□□□

→ How to Sort index by using Sort_index() method:-

It is highly recommended to perform sorting of indices.

Syntax:- df.Sort_index(axis=0, level=None, key=None, ascending=True, inplace=False

eg:- df=pd.read_csv('emp.csv', index_col=['Date', 'subject', 'Faculty'])
df.sort_index(inplace=True)
print(df)

→ How to Sort a particular levels :-

df.sort_index(level=2, inplace=True, ascending=False)
df.sort_index(level=(0,1), ascending=[False, True], inplace=True)

→ Important Methods of Multi-Index :-

Multi-Index is a Class in pandas library.

(1) is_lexsorted() :- Returns True if index is lexicographically sorted.

(2) Sortlevel() :- df.index.sortlevel(level=0, ascending=False))