Department of Computer Science and Engineering (Data Science)

Subject: Time Series Analysis

Experiment 3

(Data Wrangling and Preparation for Time Series Data)

60009210105
Amitesh Sawarkar
D 12

Aim: Implementation of Data Wrangling and Preparation for Time Series Data.

Theory:

Data Wrangling:
Data wrangling is the art of transformation and mapping raw data into valuable information with the help of preprocessing strategies. It aims to provide rich data that can be utilized to gain maximum insights. It comprises operations such as loading, imputing, applying transformations, treating outliers, cleaning, integrating, dealing with inconsistency, reducing dimensionality, and engineering features. Data wrangling is an integral and main part of machine learning modeling.

Real-world data is undoubtedly messy, and it is not reasonable to use data directly for modeling without performing some wrangling.

Loading Data into Pandas:
Pandas is the most notable framework for data wrangling. It includes data manipulation and analysis tools intended to make data analysis rapid and convenient. In the real world, data is generated via various tools and devices; hence, it comes in different formats such as CSV, Excel, and JSON. In addition, sometimes data needs to be read from a URL. All the data comprises several records and variables.

Loading Data Using CSV:
This dataset depicts the number of female births over time. Pandas has a built-in function to read CSV files. If files have different separators, use sep = ‘ ’ and fill in the separator, as in (;, |,\t,‘ ’). The following code imports the dataset:

```
import pandas as pd
df = pd.read_csv(r'daily-total-female-births-CA.csv')
df.head(5)
# sep can be like ; |
```

Output:

	date	births
0	1959-01-01	35
1	1959-01-02	32
2	1959-01-03	30
3	1959-01-04	31
4	1959-01-05	44

Loading Data Using Excel:

The Istanbul Stock Exchange dataset comprises the returns of the Istanbul Stock Exchange along with seven other international indices (SP, DAX, FTSE, NIKKEI, BOVESPA, MSCE_EU, MSCI_EM) from June 5, 2009, to February 22, 2011. The data is organized by workday. This data is available in Excel format, and Pandas has a built-in function to read the Excel file. The following code imports the dataset:

```
import pandas as pd
dfExcel = pd.read_excel(r'istambul_stock_exchange.xlsx', sheet_name = 'Data')
dfExcel.head(5)
```

Output:

	date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
0	2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
1	2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2	2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
3	2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
4	2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

Loading Data Using JSON:

This example dataset is in JSON format. The following code imports the dataset:

```
dfJson = pd.read_json(r'test.json')
dfJson.head(5)
```

Output:

	Names	Age
0	John	33
1	Sal	45
2	Tim	22
3	Rod	54

Loading Data from a URL:

The Abalone dataset comprises 4,177 observations and 9 variables. It is not in any file structure; instead, we can display it as text at the specific URL. The following code imports the dataset:

```
dfURL = pd.read_csv(r'https://archive.ics.uci.edu/ml/machine-learning-
databases/abalone/abalone.data', names =['Sex',
                           'Length','Diameter', 'Height','Whole weight',
'Shucked weight','Viscera weight', 'Shell weight', 'Rings'])
dfURL.head(5)
```

Output:

Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

Exploring Pandasql and Pandas:

Pandasql is a Python framework for running SQL queries on Pandas DataFrames. It has plenty of essential attributes and a similar purpose as the sqldf framework in R. It helps us to query Pandas DataFrames using SQL syntax. It provides a SQL interface to perform data wrangling on a Pandas DataFrame. Pandasql helps analysts and data engineers who are masters of SQL to transition into Python (Pandas) smoothly.

Use pip install pandasql to install the library.

Selecting the Top Five Records:

With the help of the pandas.head() function, we can fetch the first N records from the dataset. We can do the same operation with the help of Pandassql. The example illustrates how to do it with Pandas and Pandasql.

Pandas:

```
import pandas as pd
dfp = pd.read_excel(r'Absenteeism_at_work.xls')
dfp.head(5)
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
0	11	26	7	3	1	289	36	13	33
1	36	0	7	3	1	118	13	18	50
2	3	23	7	4	1	179	51	18	38
3	7	7	7	5	1	279	5	14	39
4	11	23	7	5	1	289	36	13	33

5 rows × 21 columns

Pandasql:

```
from pandasql import sqldf
dfpsql = pd.read_excel(r'Absenteeism_at_work.xls')
Query_string = """ select * from dfpsql limit 5 """
sqldf(Query_string, globals())
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
0	11	26	7	3	1	289	36	13	33
1	36	0	7	3	1	118	13	18	50
2	3	23	7	4	1	179	51	18	38
3	7	7	7	5	1	279	5	14	39
4	11	23	7	5	1	289	36	13	33

5 rows × 21 columns

Applying a Filter:

Data filtering is a significant part of data preprocessing. With filtering, we can choose a smaller partition of the dataset and use that subset for viewing and munging; we need specific criteria or a rule to filter the data. This is also known as **subsetting data or drill-down data**. The following example illustrates how to apply a filter with Pandas and Pandasql.

Pandas:

```
dfp[(dfp['Age'] >=30) & (dfp['Age'] <=45)]
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	A
0	11	26	7	3	1	289	36	13	
2	3	23	7	4	1	179	51	18	
3	7	7	7	5	1	279	5	14	
4	11	23	7	5	1	289	36	13	
5	3	23	7	6	1	179	51	18	
7	20	23	7	6	1	260	50	11	
8	14	19	7	2	1	155	12	14	
9	1	22	7	2	1	235	11	14	
10	20	1	7	2	1	260	50	11	
11	20	1	7	3	1	260	50	11	

Pandasql:

```
Query_string = """ select * from dfpsql where age>=30 and age<=45 """
sqldf(Query_string, globals())
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36	13
1	3	23	7	4	1	179	51	18
2	7	7	7	5	1	279	5	14
3	11	23	7	5	1	289	36	13
4	3	23	7	6	1	179	51	18
5	20	23	7	6	1	260	50	11
6	14	19	7	2	1	155	12	14
7	1	22	7	2	1	235	11	14
8	20	1	7	2	1	260	50	11
9	20	1	7	3	1	260	50	11

Distinct (Unique):
Several duplicate records exist in the dataset. If we want to select the number of unique values for the specific variable, then we can use the unique() function of Pandas. The following example illustrates how to do this with Pandas and Pandasql.

Pandas
```
dfp['ID'].unique()
```

Output:
```
array([11, 36,  3,  7, 10, 20, 14,  1, 24,  6, 33, 18, 30,  2, 19, 27, 34,
        5, 15, 29, 28, 13, 22, 17, 31, 23, 32,  9, 26, 21,  8, 25, 12, 16,
        4, 35], dtype=int64)
```

Pandasql:
```
Query_string = """ select distinct ID from dfpsql;"""
sqldf(Query_string, globals())
```

Output:

	ID
0	11
1	36
2	3
3	7
4	10
5	20
6	14
7	1
8	24
9	6
10	33

IN:
Data filtering is a process of extracting essential data from a dataset by using some condition. There are several methods to do filtering on a dataset. Sometimes we want to investigate whether the data has been associated with a particular DataFrame or Series. In such an event, we can use Pandas' isin() function, which checks whether values are present in the sequence. The procedure can also be carried out in Pandasql. The following example illustrates how to do it with Pandas and Pandasql.

Pandas:
```
dfp[dfp.Age.isin([20,30,40])]
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
47	15	23	9	5	1	291	31	12
49	15	14	9	2	4	291	31	12
65	22	23	10	5	4	179	26	9
71	15	23	10	5	4	291	31	12
75	15	14	10	3	4	291	31	12
83	17	21	11	5	4	179	22	17
84	15	23	11	5	4	291	31	12
87	15	14	11	2	4	291	31	12
91	17	21	11	4	4	179	22	17
97	15	23	11	5	4	291	31	12

Pandasql:

```
Query_string = """ select * from dfpsql where Age in(20,30,40);"""
sqldf(Query_string, globals())
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	15	23	9	5	1	291	31	12
1	15	14	9	2	4	291	31	12
2	22	23	10	5	4	179	26	9
3	15	23	10	5	4	291	31	12
4	15	14	10	3	4	291	31	12
5	17	21	11	5	4	179	22	17
6	15	23	11	5	4	291	31	12
7	15	14	11	2	4	291	31	12
8	17	21	11	4	4	179	22	17
9	15	23	11	5	4	291	31	12

NOT IN:

The NOT IN operation is used for a similar purpose as explained earlier. If we want to check whether a value is not part of a sequence, we can use the tilde (~) symbol to perform a NOT IN operation.

Pandas:

```
dfp[~dfp.Age.isin([20,30,40])]
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36	13
1	36	0	7	3	1	118	13	18
2	3	23	7	4	1	179	51	18
3	7	7	7	5	1	279	5	14
4	11	23	7	5	1	289	36	13
5	3	23	7	6	1	179	51	18
6	10	22	7	6	1	361	52	3
7	20	23	7	6	1	260	50	11
8	14	19	7	2	1	155	12	14
9	1	22	7	2	1	235	11	14

Pandasql:

```
Query_string = """ select * from dfpsql where Age not in(20,30,40);"""
sqldf(Query_string, globals())
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	11	26	7	3	1	289	36	13
1	36	0	7	3	1	118	13	18
2	3	23	7	4	1	179	51	18
3	7	7	7	5	1	279	5	14
4	11	23	7	5	1	289	36	13
5	3	23	7	6	1	179	51	18
6	10	22	7	6	1	361	52	3
7	20	23	7	6	1	260	50	11
8	14	19	7	2	1	155	12	14
9	1	22	7	2	1	235	11	14

Ascending Data Order:

ORDER BY sorts the result-set in ascending or descending order based on the value selected. Pandas has a sort_value() function that can use different sorting algorithms, such as quicksort, mergesort, and heapsort. The default value is ascending order, whose Boolean value is True. Except for that axis-wise, we do perform sorting such as 0 for index and 1 for columns. SQL has an ORDER BY clause to perform a similar sorting operation.

Pandas:

```
dfp.sort_values(by = ['Age','Service_time'], ascending= True)
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
40	27	23	9	3	1	184	42	7
118	27	23	1	5	2	184	42	7
132	27	23	1	5	2	184	42	7
137	27	23	2	6	2	184	42	7
149	27	23	2	3	2	184	42	7
209	27	7	5	4	3	184	42	7
269	27	6	8	4	1	184	42	7
6	10	22	7	6	1	361	52	3
22	10	13	8	2	1	361	52	3
25	10	25	8	2	1	361	52	3

Pandasql:

```
Query_string = """ select * from dfpsql order by Age,Service_time;"""
sqldf(Query_string, globals())
```

Output:

	ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
0	27	23	9	3	1	184	42	7
1	27	23	1	5	2	184	42	7
2	27	23	1	5	2	184	42	7
3	27	23	2	6	2	184	42	7
4	27	23	2	3	2	184	42	7
5	27	7	5	4	3	184	42	7
6	27	6	8	4	1	184	42	7
7	10	22	7	6	1	361	52	3
8	10	13	8	2	1	361	52	3
9	10	25	8	2	1	361	52	3

Descending Data Order:

As mentioned, the sort_value() function can sort results in ascending or descending order. It needs the parameter ascending = False to sort values in descending order. You can also update some other parameters, as explained for ORDER BY ascending.

Pandas:

```
dfp.sort_values(by = ['Age','Service_time'], ascending= False)
```

Output:

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time
9	18	8	3	1	228	14	16
9	18	5	4	3	228	14	16
9	1	10	4	4	228	14	16
9	25	3	3	2	228	14	16
9	12	3	3	2	228	14	16
9	25	3	4	2	228	14	16
9	6	7	2	1	228	14	16
9	6	7	3	1	228	14	16
35	0	0	6	3	179	45	14
36	0	7	3	1	118	13	18

Pandasql:

```
Query_string = """ select * from dfpsql order by Age Desc,Service_time
Desc;"""
sqldf(Query_string, globals())
```

Output:

ID	Reason_for_absence	Month_of_absence	Day_of_the_week	Seasons	Transportation_expense	Distance_from_Residence_to_Work	Service_time	Age
9	18	8	3	1	228	14	16	58
9	18	5	4	3	228	14	16	58
9	1	10	4	4	228	14	16	58
9	25	3	3	2	228	14	16	58
9	12	3	3	2	228	14	16	58
9	25	3	4	2	228	14	16	58
9	6	7	2	1	228	14	16	58
9	6	7	3	1	228	14	16	58
35	0	0	6	3	179	45	14	53
36	0	7	3	1	118	13	18	50

Aggregation:

Aggregation is the process of mining data so the data can be investigated, collected, and presented in a summarized manner. It allows you to perform a calculation on single or multiple vectors, usually accompanied by the group by() functions in Pandas. It performs numerous operations such as splitting, applying, and combining. This example illustrates the aggregation operation in Pandas and Pandasql. For the following example, we are leveraging .agg in Pandas to achieve the required result.

Pandas:

```
dfp.agg({'Transportation_expense': ['count','min', 'max', 'mean']})
```

Output:

	Transportation_expense
count	740.00000
min	118.00000
max	388.00000
mean	221.32973

Pandasql:

```
Query_string = """ select count(Transportation_expense) as count,
min(Transportation_expense) as min, max(Transportation_expense) as max,
avg(Transportation_expense) as mean from dfp;"""
sqldf(Query_string, globals())
```

Output:

	count	min	max	mean
0	740	118	388	221.32973

GROUP BY:

We can use GROUP BY to arrange identical data into groups based on the aggregation function used. In other words, it groups rows that have similar values into summary rows. We perform this operation with the groupby() function in Pandas. SQL has its GROUP BY clause to perform a similar operation. The example illustrates the GROUP BY operation in Pandas and Pandasql.

Pandas:

```
dfp.groupby('ID')['Service_time'].sum()
```

Output:

```
ID
1        322
2         72
3       2034
4         13
5        247
6        104
7         84
8         28
9        128
10        72
11       520
12         7
13       180
14       406
15       444
16        48
17       340
```

Pandasql:

```
Query_string = """ select ID , sum(Service_time) as Sum_Service_time from dfp
group by ID;"""
sqldf(Query_string, globals())
```

Output:

ID	Sum_Service_time
1	322
2	72
3	2034
4	13
5	247
6	104
7	84
8	28
9	128
10	72
11	520

GROUP BY with Aggregation:

The groupby() function provides a group of similar data based on a selected feature with that data; we can perform different aggregation operations with the .agg() function on other features in Pandas. In SQL, we use GROUP BY to apply aggregate functions on groups of data returned from a query. FILTER is a modifier used with an aggregate function to bound the values used in an aggregation.

Pandas:

```
dfp.groupby('Reason_for_absence').agg({'Age': ['mean','min','max']})
```

Output:

	Age		
	mean	min	max
Reason_for_absence			
0	39.604651	28	53
1	37.687500	28	58
2	28.000000	28	28
3	40.000000	40	40
4	45.000000	41	49
5	41.666667	37	50
6	38.500000	27	58
7	32.866667	27	46
8	36.500000	28	40

Pandasql:

```
Query_string = """ select Reason_for_absence , avg(Age) as mean, min(Age) as
min, max(Age) as max from dfp
group by Reason_for_absence;"""
sqldf(Query_string, globals())
```

Output:

Reason_for_absence	mean	min	max
0	39.604651	28	53
1	37.687500	28	58
2	28.000000	28	28
3	40.000000	40	40
4	45.000000	41	49
5	41.666667	37	50
6	38.500000	27	58
7	32.866667	27	46
8	36.500000	28	40

Join (Merge):

The terms join and merge mean the same thing in Pandas, Python, and other languages like SQL and R. In reality, their fundamental operation is equivalent, but their way of executing an operation is different. A join in Pandas utilizes an index to consolidate two data sources, while a merge looks for overlapping columns to merge. In Pandas, the merge() and join() functions are used. In SQL, the MERGE and JOIN operators perform the same operation as in Pandas. This example illustrates the aggregation operation in Pandas and Pandasql. We've created Sample Employee and Department tables to illustrate the join examples.

Pandas:

```
import pandas as pd
data1 = {
        'Empid': [1011, 1012, 1013, 1014, 1015],
        'Name': ['John', 'Rahul', 'Rick', 'Morty', 'Tim'],
        'Designation': ['Manager', 'Research Engineer', ' Research Engineer', 
'VP', 'Delivery Manager'],
        'Date_of_joining': ['01-Jan-2000', '23-sep-2006', '11-Jan-2012', '21-
Jan-1991', '12-Jan-1990']}
Emp_df = pd.DataFrame(data1, columns = ['Empid', 'Name', 
'Designation','Date_of_joining'])
Emp_df
```

Output:

Empid	Name	Designation	Date_of_joining
1011	John	Manager	01-Jan-2000
1012	Rahul	Research Engineer	23-sep-2006
1013	Rick	Research Engineer	11-Jan-2012
1014	Morty	VP	21-Jan-1991
1015	Tim	Delivery Manager	12-Jan-1990

Pandasql:

```
data2 = {
        'Empid': [1011, 1017, 1013, 1019, 1015],
        'Deptartment': ['Management', 'Research', 'Research', 'Management', 
'Delivery'],
        'Total_Experience': [18, 10, 10, 28, 22]}
Dept_df = pd.DataFrame(data2, columns = ['Empid', 'Deptartment', 
'Total_Experience'])
Dept_df
```

Output:

Empid	Deptartment	Total_Experience
1011	Management	18
1017	Research	10
1013	Research	10
1019	Management	28
1015	Delivery	22

INNER JOIN:

An inner merge/inner join keeps rows where the merge value contains an "on" value in both the DataFrames. It selects the records that have matching values (joining columns) in one or more tables.

Pandas:

```
pd.merge(Emp_df, Dept_df, left_on='Empid',right_on='Empid', how='inner')
```

Output:

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22

Pandasql:

```
Query_string = """ select * from Emp_df a INNER JOIN Dept_df b ON a.Empid = b.Empid;"""
sqldf(Query_string, globals())
```

Output:

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011	Management	18
1013	Rick	Research Engineer	11-Jan-2012	1013	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	1015	Delivery	22

LEFT JOIN:

A left merge or left join keeps a row on the left DataFrame when the missing value finds the "on" variable in the right DataFrame and adds the empty or NaN value in that result. All the records from the left table and selected matching records based on the joining condition from one or more tables.

Pandas:

```
pd.merge(Emp_df, Dept_df,left_on='Empid',right_on='Empid', how='left')
```

Output:

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	NaN
1013	Rick	Research Engineer	11-Jan-2012	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	NaN
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22.0

Pandasql:

```
Query_string = """ select * from Emp_df a LEFT JOIN Dept_df b ON  a.Empid = b.Empid;"""
sqldf(Query_string, globals())
```

Output:

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011.0	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	None	NaN
1013	Rick	Research Engineer	11-Jan-2012	1013.0	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	None	NaN
1015	Tim	Delivery Manager	12-Jan-1990	1015.0	Delivery	22.0

RIGHT JOIN:

A right merge or right join keeps every row on the right DataFrame. Where the missing values find the “on” variable in the left columns, you add the empty/ NaN values to the result. All the records from the left table and selected matching records based on the joining condition from one or more tables.

Pandas:

```
pd.merge(Emp_df, Dept_df,left_on='Empid',right_on='Empid', how='right')
```

Output:

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22
1017	NaN	NaN	NaN	Research	10
1019	NaN	NaN	NaN	Management	28

Pandasql:

```
Query_string = """ select
a.Empid,Name,Designation,Date_of_joining,Deptartment,Total_Experience from
Dept_df a LEFT JOIN Emp_df b ON  a.Empid = b.Empid;"""
sqldf(Query_string, globals())
```

Output:

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18
1017	None	None	None	Research	10
1013	Rick	Research Engineer	11-Jan-2012	Research	10
1019	None	None	None	Management	28
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22

OUTER JOIN:

An outer merge/full outer join returns all the rows from left and right DataFrames only where it matches or returns NaNs or empty values. It returns all the records from both the tables based on the joining condition regardless of whether they match or not. Not matching ones will be nulls.

Pandas:

```
pd.merge(Emp_df, Dept_df,left_on='Empid',right_on='Empid', how='outer')
```

Output:

Empid	Name	Designation	Date_of_joining	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	NaN
1013	Rick	Research Engineer	11-Jan-2012	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	NaN
1015	Tim	Delivery Manager	12-Jan-1990	Delivery	22.0
1017	NaN	NaN	NaN	Research	10.0
1019	NaN	NaN	NaN	Management	28.0

Pandasql:

```
Query_string = """ select * from Emp_df a left OUTER JOIN Dept_df b
ON  a.Empid = b.Empid;"""
sqldf(Query_string, globals())
```

Output:

Empid	Name	Designation	Date_of_joining	Empid	Deptartment	Total_Experience
1011	John	Manager	01-Jan-2000	1011.0	Management	18.0
1012	Rahul	Research Engineer	23-sep-2006	NaN	None	NaN
1013	Rick	Research Engineer	11-Jan-2012	1013.0	Research	10.0
1014	Morty	VP	21-Jan-1991	NaN	None	NaN
1015	Tim	Delivery Manager	12-Jan-1990	1015.0	Delivery	22.0

Summary of the DataFrame:

Descriptive statistics is a method used to depict data in a meaningful way to represent either the entire population or a sample. It has two sections: the measure of **central tendency**, which is used to measure the summary of a sample and population (e.g., mean, median, and mode), and the **measure of variability**, which is used to measure the spread or dispersion in the dataset (e.g., range, interquartile, variance, and standard deviation).

In Pandas, it provides a built-in function called **describe()**. The following example illustrates the detailed result in Pandas. In Pandas, the describe() function returns a number of descriptive statistics values (e.g., mean, standard deviation, min, median, max, first quartile, third quartile).

```
import pandas as pd
dfsumm = pd.read_csv(r'https://archive.ics.uci.edu/ml/machine-learning-databases/abalone/abalone.data', names =['Sex',
                            'Length','Diameter', 'Height','Whole weight',
'Shucked weight','Viscera weight', 'Shell weight', 'Rings'])
dfsumm.head(5)
```

Output:

	Sex	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
	M	0.455	0.365	0.095	0.5140	0.2245	0.1010	0.150	15
	M	0.350	0.265	0.090	0.2255	0.0995	0.0485	0.070	7
	F	0.530	0.420	0.135	0.6770	0.2565	0.1415	0.210	9
	M	0.440	0.365	0.125	0.5160	0.2155	0.1140	0.155	10
	I	0.330	0.255	0.080	0.2050	0.0895	0.0395	0.055	7

```
dfsumm.describe()
```

Output:

	Length	Diameter	Height	Whole weight	Shucked weight	Viscera weight	Shell weight	Rings
count	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000	4177.000000
mean	0.523992	0.407881	0.139516	0.828742	0.359367	0.180594	0.238831	9.933684
std	0.120093	0.099240	0.041827	0.490389	0.221963	0.109614	0.139203	3.224169
min	0.075000	0.055000	0.000000	0.002000	0.001000	0.000500	0.001500	1.000000
25%	0.450000	0.350000	0.115000	0.441500	0.186000	0.093500	0.130000	8.000000
50%	0.545000	0.425000	0.140000	0.799500	0.336000	0.171000	0.234000	9.000000
75%	0.615000	0.480000	0.165000	1.153000	0.502000	0.253000	0.329000	11.000000
max	0.815000	0.650000	1.130000	2.825500	1.488000	0.760000	1.005000	29.000000

Resampling:

Resampling is a method to convert the time-based observed data into a different interval. In other words, it is used to change the time frequency into another time-frequency format. For instance, say we want to change monthly data into a year-wise format or upsample week data into hours. We would perform either upsample or downsample operations. For that, every data object must have a DateTime-like index(Datetimeindex, PeriodIdnex, TimedeltaIndex).

```
import pandas as pd
df = pd.read_csv(r'daily-total-female-births-CA.csv',index_col =0,
                 parse_dates=['date'])
df.head(5)
```

Output:

	births
date	
1959-01-01	35
1959-01-02	32
1959-01-03	30
1959-01-04	31
1959-01-05	44

Resampling by Month:

```
df.births.resample('M').mean()
```

Output:

```
date
1959-01-31    39.129032
1959-02-28    41.000000
1959-03-31    39.290323
1959-04-30    39.833333
1959-05-31    38.967742
1959-06-30    40.400000
1959-07-31    41.935484
1959-08-31    43.580645
1959-09-30    48.200000
1959-10-31    44.129032
1959-11-30    45.000000
1959-12-31    42.387097
Freq: M, Name: births, dtype: float64
```

Resampling by Quarter:

```
df.births.resample('Q').mean()
```

Output:

```
date
1959-03-31    39.766667
1959-06-30    39.725275
1959-09-30    44.532609
1959-12-31    43.826087
Freq: Q-DEC, Name: births, dtype: float64
```

Resampling by Year:

```
df.births.resample('Y').mean()
```

Output:

```
date
1959-12-31    41.980822
Freq: A-DEC, Name: births, dtype: float64
```

Resampling by Week:

```
df.births.resample('W').mean()
```

Output:

```
date
1959-01-04    32.000000
1959-01-11    37.714286
1959-01-18    44.285714
1959-01-25    41.142857
1959-02-01    35.142857
1959-02-08    40.428571
1959-02-15    42.857143
1959-02-22    42.428571
1959-03-01    40.000000
1959-03-08    39.428571
1959-03-15    36.571429
1959-03-22    40.857143
1959-03-29    39.142857
1959-04-05    41.142857
1959-04-12    37.857143
1959-04-19    37.285714
1959-04-26    40.142857
```

Resampling on a Semimonthly basis:

```
df.births.resample('SM').mean()
```

Output:

```
date
1958-12-31    37.642857
1959-01-15    41.375000
1959-01-31    38.533333
1959-02-15    43.384615
1959-02-28    38.000000
1959-03-15    39.812500
1959-03-31    38.333333
1959-04-15    40.666667
1959-04-30    39.133333
1959-05-15    39.625000
1959-05-31    40.800000
1959-06-15    38.600000
1959-06-30    43.733333
1959-07-15    41.375000
1959-07-31    43.733333
1959-08-15    43.250000
```

Windowing Function:

A windowing function is used to calculate the number of operations, such as rolling count, rolling sum, rolling mean, rolling median, rolling variance, rolling standard deviation, rolling min, rolling max, rolling correlation, rolling covariance, rolling skewness, rolling kurtosis, rolling quantile, etc. In addition, we can perform some other operations such as expanding window and exponential weighted moving window.

```
import pandas as pd
dfExcelwin = pd.read_excel(r'istambul_stock_exchange.xlsx', sheet_name =
'Data'
                              ,index_col =0,
                parse_dates=['date'])
dfExcelwin.head(5)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

Rolling Window:

The rolling window feature supports the following methods: count, sum, mean, median, var, std, min, max, corr, cov, skew, kurt, quantile, sum, and aggregate.

```
dfExcelwin.rolling(window=4).mean().head(10)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-08	-0.007473	-0.010220	-0.005993	-0.004728	-0.003110	-0.004651	0.010624	0.000351	-0.000536
2009-01-09	-0.013946	-0.017400	-0.010206	-0.010244	-0.007261	-0.005770	0.000385	-0.005570	-0.009617
2009-01-12	-0.027600	-0.035943	-0.017858	-0.015739	-0.011734	-0.019070	-0.017807	-0.011518	-0.017468
2009-01-13	-0.016523	-0.029423	-0.009802	-0.015700	-0.006086	-0.023393	-0.007940	-0.010305	-0.013671
2009-01-14	-0.011263	-0.017132	-0.019158	-0.024614	-0.018705	-0.012650	-0.025086	-0.020220	-0.010984
2009-01-15	-0.013563	-0.023863	-0.013443	-0.024533	-0.019112	-0.024143	-0.015066	-0.020491	-0.014891

Expanding Window:

Expanding window supports the following methods: count, sum, mean, median, var, std, min, max, corr, cov, skew, kurt, quantile, sum, aggregate, and quantile.

```
dfExcelwin.expanding(min_periods=4).mean().head(10)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-08	-0.007473	-0.010220	-0.005993	-0.004728	-0.003110	-0.004651	0.010624	0.000351	-0.000536
2009-01-09	-0.004006	-0.006244	-0.009101	-0.007757	-0.005030	-0.004616	0.006546	-0.001917	-0.001989
2009-01-12	-0.008204	-0.012264	-0.011388	-0.008718	-0.005029	-0.012020	-0.003520	-0.003672	-0.005429
2009-01-13	-0.004825	-0.010551	-0.009510	-0.009998	-0.005188	-0.010303	-0.002507	-0.004894	-0.005343
2009-01-14	-0.009368	-0.013676	-0.012575	-0.014671	-0.010908	-0.008651	-0.007231	-0.009934	-0.005760
2009-01-15	-0.008254	-0.014075	-0.011030	-0.015213	-0.011289	-0.013295	-0.003059	-0.010172	-0.007723

Exponentially Weighted Moving Window:

Expanding window supports the following methods: mean, std, var, corr, and cov.

```
dfExcelwin.ewm(com=0.5).mean().head(10)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.028008	0.033454	0.004670	0.006890	0.010623	0.003122	0.021987	0.011680	0.013711
2009-01-07	-0.011363	-0.007951	-0.019657	-0.010226	-0.016625	0.012933	-0.018088	-0.008226	-0.009638
2009-01-08	-0.045684	-0.059767	-0.004099	-0.011239	-0.005718	-0.022838	0.013213	-0.006427	-0.016243
2009-01-09	-0.008502	-0.013292	-0.015770	-0.017019	-0.010398	-0.010545	-0.002168	-0.009481	-0.010593
2009-01-12	-0.022313	-0.032698	-0.020478	-0.014687	-0.006812	-0.036242	-0.036670	-0.011464	-0.018628
2009-01-13	0.002871	-0.011071	-0.005648	-0.016679	-0.006365	-0.012070	-0.009830	-0.011968	-0.009423
2009-01-14	-0.026493	-0.027394	-0.024574	-0.037152	-0.036090	-0.002080	-0.030147	-0.034140	-0.008925

Shifting:

Shifting is also known as “lag” and moves a value back or forward in time. Pandas has a df.shift() function to shift an index by the desired number of periods with an optional time frequency.

```
import pandas as pd
dfshift = pd.read_excel(r'istambul_stock_exchange.xlsx', sheet_name = 'Data'
                                   ,index_col =0,
                 parse_dates=['date'])
dfshift.head(5)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-06	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-07	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-08	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-09	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802

```
dfshift.shift(periods=3).head(7)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-06	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-07	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN	NaN
2009-01-08	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-09	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-12	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-13	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424

```
dfshift.shift(periods=-1).head(7)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-06	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-07	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424
2009-01-08	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474	-0.009764	-0.010989	-0.007802
2009-01-09	-0.029191	-0.042361	-0.022823	-0.013526	-0.005026	-0.049039	-0.053849	-0.012451	-0.022630
2009-01-12	0.015445	-0.000272	0.001757	-0.017674	-0.006141	0.000000	0.003572	-0.012220	-0.004827
2009-01-13	-0.041168	-0.035552	-0.034032	-0.047383	-0.050945	0.002912	-0.040302	-0.045220	-0.008677

```
dfshift.shift(periods=3, axis =1).head(7)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	NaN	NaN	NaN	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000
2009-01-06	NaN	NaN	NaN	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162
2009-01-07	NaN	NaN	NaN	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293
2009-01-08	NaN	NaN	NaN	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061
2009-01-09	NaN	NaN	NaN	0.009860	0.009658	-0.021533	-0.019873	-0.012710	-0.004474
2009-01-12	NaN	NaN	NaN	-0.029191	-0.042361	-0.022823	-0.013526	-0.005026	-0.049039
2009-01-13	NaN	NaN	NaN	0.015445	-0.000272	0.001757	-0.017674	-0.006141	0.000000

```
dfshift.shift(periods=3,fill_value=0).head(7)
```

Output:

date	ISE	ISED	SP	DAX	FTSE	NIKKEI	BOVESPA	EU	EM
2009-01-05	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-06	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-07	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000	0.000000
2009-01-08	0.035754	0.038376	-0.004679	0.002193	0.003894	0.000000	0.031190	0.012698	0.028524
2009-01-09	0.025426	0.031813	0.007787	0.008455	0.012866	0.004162	0.018920	0.011341	0.008773
2009-01-12	-0.028862	-0.026353	-0.030469	-0.017833	-0.028735	0.017293	-0.035899	-0.017073	-0.020015
2009-01-13	-0.062208	-0.084716	0.003391	-0.011726	-0.000466	-0.040061	0.028283	-0.005561	-0.019424

Handling Missing Data:
A missing value or missing data occurs when no data value is found at a respective number of places in either a DataFrame or dataset. A missing value presents many problems in the dataset. It reduces the statistical power of data, and the lost data can increase the bias in the dataset. So, it is essential to handle this missing value to maintain the characteristics of the data. Pandas has a number of methods to handle the missing values such as bfill(), ffill(), interpolate(), and fillna().

```
import pandas as pd
dfmiss = pd.read_csv(r'daily-total-female-births-CA-with_nulls.csv',index_col
=0,
                 parse_dates=['date'])
dfmiss
```

Output:

	births
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	NaN
1959-01-09	38.0
1959-01-10	27.0

```
# Snippet to check for nulls
dfmiss.isnull().sum()
```

Output:

```
births    16
dtype: int64
```

BFILL:
The backward method fills the missing values in the dataset and uses the next valid observation to fill the gap.

```
dfmiss.bfill()
```

Output:

	births
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	38.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

FFILL:
The forward fill method propagates the last valid observation forward to the next valid one.

```
dfmiss.ffill()
```

Output:

	births
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	45.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

FILLNA:
This replaces NA/NaN values using a constant value.

```
dfmiss.fillna(10)
```

Output:

	births
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	10.0
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

INTERPOLATE:

This method interpolates values linearly in a forward direction.

```
#interpolate
dfmiss.interpolate(method='linear',limit_direction='forward')
```

Output:

	births
date	
1959-01-01	35.0
1959-01-02	32.0
1959-01-03	30.0
1959-01-04	31.0
1959-01-05	44.0
1959-01-06	29.0
1959-01-07	45.0
1959-01-08	41.5
1959-01-09	38.0
1959-01-10	27.0
1959-01-11	38.0

Lab Assignments to complete:

Perform the following tasks using the datasets mentioned. Download the datasets from the link given:

Link:

https://drive.google.com/drive/folders/107ecl29wJUZdnKl--PqvRfkUe8MaQHse?usp=sharing

Dataset 1: Smart City Index Headers

1. Implement data wrangling with the help of various pre-processing strategies.

COLAB FILE UPLOADED HERE -

https://colab.research.google.com/drive/1MULrCO9Z69FuBuCZgJp38wMaYsMZUqAz?usp=sharing