Master Data Science And Data Analysis with Pandas

Arun

Master
Data Science
and
Data Analysis
With
Pandas
By
Arun

First printing edition 2020.

ACKNOWLEDGEMENT

First and foremost, praises and thanks to the God, the Almighty, for His showers of blessings throughout my research work to complete the book successfully.

I would like to express my deep and sincere gratitude to my friend and colleague Rita Vishwakarma for her motivation. She has been a great support throughout the journey and encouraged me whenever needed.

I would like to extend my thanks to my colleague Nidhi Srivastava for her faith in me. She always wanted me to share my knowledge in the form of books and contribute to the society.

I am extremely grateful to my parents for their love, prayers, caring and sacrifices for educating and preparing me for my future. I am very much thankful to them for their understanding and continuing support to complete this book. Also, I express my thanks to my sisters and brother for their support and valuable prayers. My Special thanks goes to my teachers who not only educated me but also prepared me for the future. They are the lamps that burns themselves to give light to the society.

Finally, my thanks go to all the people who have supported me to complete the research work directly or indirectly.

Arun

Table Of content

1 Introduction

Today, data is the biggest wealth (after health) as it acts as fuel to many algorithms. Artificial intelligence and data sciences are the biggest examples of this.

To use this data in the algorithm, need comes to handle this data and manage accordingly. Data analysis solves a huge part of the problem. But how to handle this? The answer lies in one of the most used Python libraries for data analysis named ‘Pandas’.

Pandas is mainly used to deal with sequential and tabular data to manage, analyze and manipulate data in convenient and efficient manner. It is built on top of the NumPy library and has two primary data structures Series (1-dimensional) and DataFrame (2-dimensional).

Pandas generally converts your data (from csv, html, excel, etc.) into a two-dimensional data structure (DataFrame). It is size-mutable, potentially heterogeneous tabular data structure with labeled axes (rows and columns). The data in DataFrame is then manipulated as per the need, analyzed and then stored back into some form (like csv, excel etc.).

Let’s look at the advantages of using Pandas as a data analysis tool in the next chapter.

2 Advantages

Pandas is a widely used library for data analysis. The main reasons are:

2.1 Speed: use of Pandas decreases the execution time when compared to the traditional programming.

2.2 Short code: use of Pandas facilitates smaller code compared to the traditional way of writing the code as what would have taken multiple lines of code without Pandas can be done in fewer lines.

2.3 Saves time: as the amount of code needs to be written is less, the amount of time spent on programming is also less and thus saves times for other works.

2.4 Easy: the DataFrame generated with Pandas is easy to analyze and manipulate.

2.5 Versatile: Pandas is a powerful tool which can be used for various tasks like analyzing, visualizing, filtering data as per certain condition, segmentation etc.

2.6 Efficiency: the library has been built in such a way that it can handle large datasets very easily and efficiently. Loading and manipulating data is also very fast.

2.7 Customizable: Pandas contain feature set to apply on the data so that you can customize, edit and pivot it according to your requirement.

2.8 Supports multiple formats for I/O: data handling and analysis on Pandas can be done on various formats available. i.e. the read/write operation for Pandas is supported for multiple formats like CSV, TSV, Excel, etc.

2.9 Python support: Python being most used for data analysis and artificial intelligence enhances the importance and value of Pandas. Pandas being a Python library make Python more useful for data analysis and vice versa.

In the coming chapter, we'll be learning the installation of Pandas.

3 Installation

3.1 Install Pandas

Pandas being a Python library, it's platform independent and can be installed on any machine where Python exists.

Officially Pandas is supported by Python version 2.7 and above.

3.1.1 Installing with Anaconda

If you have anaconda, then Pandas can easily be installed by:

conda install Pandas

OR for a specific version

conda install Pandas=0.20.3

in case you need to install Pandas on a virtual environment then:
create virtual environment:
conda create -n <venvName>
conda create -n venv

activate virtual environment:
source activate <venvName>
source activate venv

install Pandas
conda install Pandas

3.1.2 Installing with PyPI

pip install Pandas

Note: you can create virtual environment here as well and then install Pandas

Create virtual environment
python3 -m venv <venvName>
python3 -m venv venv

activate virtual environment
source activate <venvName>
source activate venv

install Pandas
pip3 install Pandas
or
pip3 install Pandas=0.20.3 (for specific version)

3.2 Install Jupyter Notebook

Any program which used Pandas can be ran as traditional Python program but for better understanding and clarity we prefer Jupyter notebook in data science problems.

The Jupyter Notebook is an open-source web application that allows you to create and share documents that contain live code, equations, visualizations and narrative text. Uses include data cleaning and transformation, numerical simulation, statistical modeling, data visualization, machine learning, and much more. In very simple words, jupyter notebooks makes easy to visualize the data.

Installation:
pip install jupyterlab

Run Jupyter:
Once the notebook has been installed by above commands, we can just write "*jupyter notebook*" on the terminal and a web notebook will be opened. A server will start and will be running while you are working on the notebook. If will kill or close the server, the notebook will also be closed.

Let's learn Dataframes and various ways to create it in the next chapter.

4 Creating DataFrames

DataFrame is the main thing on which we'll be mostly working on. Most manipulation or operation on the data will be applied by means of DataFrame. So now let's learn to create DataFrame by various means.

4.1 Creating DataFrame using dictionary data

This is a simple process in which we just need to pass the json data to the *DataFrame* method.

df = pd.DataFrame(cars)

Here, 'cars' is a json data

```
In [1]: #import libraries

        import pandas as pd
```

```
In [2]: cars = {'Brand': ['Honda','Toyota','Ford','Audi'],
                'Price': [22000,25000,27000,35000]
                }
```

```
In [3]: df = pd.DataFrame(cars)
        df
```

Out[3]:

	Brand	Price
0	Honda	22000
1	Toyota	25000
2	Ford	27000
3	Audi	35000

We have created a Dataframe from the dictionary data we have.

4.2 Creating DataFrame using list data

This is also a simple process of just passing the list to the *DataFrame* method.

4.2.1 Creating single column

df = pd.DataFrame(lst)

Here, 'cars' is a list data

```
In [1]: #import libraries

        import pandas as pd

In [2]: lst = ['Arun', 'Varun', 'Ram', 'Mohan']

In [3]: df = pd.DataFrame(lst)
        df
```

Out[3]:

	0
0	Arun
1	Varun
2	Ram
3	Mohan

This created a Dataframe from the list we have.

4.2.2 Creating multiple columns

We can also create multiple columns from lists

```
In [1]: #import libraries

        import pandas as pd

In [2]: names = ['Arun', 'Varun', 'Ram', 'Mohan']
        age = [29, 21, 32, 51]

In [6]: df = pd.DataFrame(list(zip(names, age)))
        df
```

Out[6]:

	0	1
0	Arun	29
1	Varun	21
2	Ram	32
3	Mohan	51

Hence, we have created a multi column Dataframe from different list we have. The main point to note here is the length of each list should be same. If the length does not match, we'll get an error.

4.3 Adding custom column name

We can add the column name as per our requirement also. This is done by a

parameter '*column*'. This is can be passed for any DataFrame and add a custom column name.

```
In [5]: df = pd.DataFrame(list(zip(names, age)), columns =['Name', 'val'])
        df
```

Out[5]:

	Name	val
0	Arun	29
1	Varun	21
2	Ram	32
3	Mohan	51

In the same way, we can give column name to any DataFrame. The general syntax is:

df = pd.DataFrame(<data>, columns=['<col1>, '<col2>])

4.4 creating DataFrame from list of dictionaries

we can create DataFrames from a list of dictionaries also. Each element (dict) of the list will become a row and the dictionary key value pair will become column and its value.

```
In [32]: lst = [
             {"name":"Arun", "age":29, "gender": "M"},
             {"name":"Varun", "age":21, "gender": "M"},
             {"name":"Ram", "age":32, "gender": "M"},
               ]
```

```
In [33]: df = pd.DataFrame(lst)
         df
```

Out[33]:

	name	age	gender
0	Arun	29	M
1	Varun	21	M
2	Ram	32	M

4.5 creating DataFrame from other files

DataFrame can be created from other file types too like CSV, MS Excel.

Even we can create DataFrame with simple txt files too (with some delimiter).

Here we'll be using example of CSV to create the DataFrame

df = pd.read_csv(<csvName>)

```
In [1]: #import libraries

        import pandas as pd
```

```
In [2]: df = pd.read_csv('name_age.csv')
        df
```

Out[2]:

	name	age
0	Arun	29
1	Varun	21
2	Ram	32
3	Mohan	51

This will give us the data in the csv in the form of a Dataframe.

4.6 Creating blank DataFrame

We sometime need to create a blank DataFrame and then later use it inside some loop. If we try to create the DataFrame inside the loop, it will keep initializing for every iteration.

df = pd.DataFrame

```
In [1]: #import libraries

        import pandas as pd
```

```
In [2]: df = pd.DataFrame
        df
```

```
Out[2]: pandas.core.frame.DataFrame
```

This gives us a blank DataFrame. We can add the column names if we know it in advance by:

df = pd.DataFrame(columns=['name', 'age'])

```
In [3]: df = pd.DataFrame(columns=['name', 'age'])
        df
```

Out[3]:

name	age

This could be important if we want to append data row by row to a Dataframe. In that case it's better to have predefined columns.

Now that we have learned about Dataframes and various ways to create it, in the next chapter we'll look at some basic operation on Dataframe to work with our data.

5 Basics of DataFrames

DataFrames are the tabular representation of the data in the form or rows and columns.

Let's do some basic operations using some csv data.

5.1 Read data

We must be aware of reading the data and creating the DataFrame as this is how we'll be proceeding towards further analysis.

df = pd.read_csv(<path of the csv>)

```
In [6]: df = pd.read_csv('name_age.csv')
        df
```

Out[6]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	f
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m
5	Radha	23	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	f
8	Nidhi	29	10/01/91	f

5.2 Shape of the DataFrame

DataFrame is a two-dimensional matrix and will give the shape as rows and columns by
df.shape

```
In [7]: df.shape
Out[7]: (9, 4)
```

This is a tuple and thus if we need to store the rows and columns into some variables

rows, columns = df.shape

```
In [11]: rows, columns = df.shape

In [12]: rows
Out[12]: 9

In [13]: columns
Out[13]: 4
```

5.3 Top '*n*' rows

df.head() gives us the top 5 entries by default.

```
In [14]: df.head()
Out[14]:
```

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	f
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m

We can even get the desired number of top entries by the same command. If we need top 3 entries the we just need to pass that value to the *head()*.

df.head(3)

```
In [15]: df.head(3)
Out[15]:
```

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	f

5.4 Last 'n' rows

As head method we have tail method as well.
So, we can get the last 5 entries by:

df.tail()

```
In [16]: df.tail()
```

Out[16]:

	name	age	dob	gender
4	Ram	32	07/11/88	m
5	Radha	23	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	f
8	Nidhi	29	10/01/91	f

And last n entries by:

df.tail(n)

```
In [18]: df.tail(2)
```

Out[18]:

	name	age	dob	gender
7	Devi	20	05/01/00	f
8	Nidhi	29	10/01/91	f

5.5 Range of entries

We can even extract a range of entries from somewhere in the DataFrame by

df[5, 8]

here, important point is that this will include row #5 an exclude row #8.

```
In [22]: df[5:8]
```

Out[22]:

	name	age	dob	gender
5	Radha	23	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	f

To access all the rows *df[:]* or just *df* will work

5.6 Accessing the columns

Sometimes, when we extract the data from some alien source, we need to understand the data. To proceed further with the data manipulation, we may need to know the columns present in the data. This can be done by:

df.columns

```
In [23]: df.columns
Out[23]: Index(['name', 'age', 'dob', 'gender'], dtype='object')
```

To get the individual column data we can do

df.<columnName>
df.name

OR

df[“<columnName>”]
df[“name”]

```
In [24]: df.name #df["name"]

Out[24]: 0     Rita
         1     Arun
         2     Sita
         3    Varun
         4      Ram
         5    Radha
         6    Mohan
         7     Devi
         8    Nidhi
         Name: name, dtype: object
```

5.7 Accessing *'n'* columns

Accessing the n columns is similar to that to accessing n rows.

df[["col1", "col2"]]
df[["name", "dob"]]

```
In [33]: df[["name", "dob"]].head()
```

Out[33]:

	name	dob
0	Rita	20/02/97
1	Arun	01/07/91
2	Sita	07/07/06
3	Varun	01/05/99
4	Ram	07/11/88

5.8 Type of column

We can even check the type of the DataFrame by
type(df[<columnName>])
type(df["name"])

```
In [26]: type(df)

Out[26]: pandas.core.frame.DataFrame

In [27]: type(df["name"])

Out[27]: pandas.core.series.Series
```

5.9 Basic operations on column

Let's try some basic operations on DataFrames

5.9.1 maximum

We can get the maximum value of a column by:
df["age"].max()

```
In [35]: df["age"].max()
Out[35]: 51
```

5.9.2 minimum

df["age"].min()

```
In [36]: df["age"].min()
Out[36]: 14
```

5.9.3 mean

df["age"].mean()

```
In [37]: df["age"].mean()
Out[37]: 26.88888888888889
```

5.9.4 standard deviation

df["age"].std()

```
In [38]: df["age"].std()
Out[38]: 10.576441325470071
```

5.10 Describe the DataFrame

We can get the detail of all the data in the DataFrame like it's max, min, mean etc. by just one command
df.describe()

```
In [40]: df.describe()
```

Out[40]:

	age
count	9.000000
mean	26.888889
std	10.576441
min	14.000000
25%	21.000000
50%	23.000000
75%	29.000000
max	51.000000

5.11 Conditional operation on columns

We can have conditional operations on columns too.
E.g.: If we want the rows where age is greater than 30 in the DataFrame then,

df[df["age"] > 30]

```
In [41]: df[df["age"] > 30]
```

Out[41]:

	name	age	dob	gender
4	Ram	32	07/11/88	m
6	Mohan	51	03/03/69	m

If we want the row with minimum age, then

df[df["age"]==df["age"].min()]

```
In [45]: df[df["age"]==df["age"].min()]
```

Out[45]:

	name	age	dob	gender
2	Sita	14	07/07/06	f

If we want only the name of the people whose age is lesser than 30 then,

df["name"][df["age"] < 30]

```
In [46]: df["name"][df["age"] < 30]
Out[46]: 0     Rita
         1     Arun
         2     Sita
         3    Varun
         5    Radha
         7     Devi
         8    Nidhi
         Name: name, dtype: object
```

Or if we need two columns like “name” and “dob”

df[["name",”dob”]][df["age"] < 30]

5.12 accessing row with loc and iloc

The row data can be accessed by two more ways:

df.loc[<index>] and *df.iloc[<index position>]*

df.loc[0] and *df.iloc[0]*

```
In [47]: df.loc[0]
Out[47]: name          Rita
         age             23
         dob       20/02/97
         gender             f
         Name: 0, dtype: object

In [48]: df.iloc[0]
Out[48]: name          Rita
         age             23
         dob       20/02/97
         gender             f
         Name: 0, dtype: object
```

Here, we can see the output of the loc and iloc are same, this is because we have not set any index here. Index here is by default incremental integers.
Loc takes the index value and gives the result for that index, whereas iloc takes the position of an index and gives the row.
We can see the difference if we change the default index. In real problems we

mostly set the index to the actual data. Let's see this in next session.

5.13 Set index

Index by default comes as an incremental integer (0,1…n).

```
In [49]: df.index
Out[49]: RangeIndex(start=0, stop=9, step=1)
```

we can change this to the actual data as:

df.set_index("name")

```
In [50]: df.set_index("name")
Out[50]:
```

name	age	dob	gender
Rita	23	20/02/97	f
Arun	29	01/07/91	m
Sita	14	07/07/06	f
Varun	21	01/05/99	m
Ram	32	07/11/88	m
Radha	23	06/09/97	f
Mohan	51	03/03/69	m
Devi	20	05/01/00	f
Nidhi	29	10/01/91	f

This is correct but has a problem. This command gives a new DataFrame and does not change the existing one.

```
In [53]: df.head()
```

Out[53]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	f
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m

To make the change effective to the existing one, there are two ways.
Store the new DataFrame to the existing one

df = df.set_index(“name”)

OR

We have a parameter called ‘*inplace*’ which can be set to true to make the change effective to the existing DataFrame.

df.set_index("name", inplace=True)

```
In [54]: df.set_index("name", inplace=True)
In [55]: df.head()
```

Out[55]:

	age	dob	gender
name			
Rita	23	20/02/97	f
Arun	29	01/07/91	m
Sita	14	07/07/06	f
Varun	21	01/05/99	m
Ram	32	07/11/88	m

Now we can see the difference of loc and iloc also

```
In [59]:  1 df.loc["Arun"]

Out[59]: age             29
         dob       01/07/91
         gender           m
         Name: Arun, dtype: object
```

```
In [60]: df.iloc[0]

Out[60]: age             23
         dob       20/02/97
         gender           f
         Name: Rita, dtype: object
```

Now we can see that the loc is taking the index value i.e. name (Arun) where as iloc gives result for the position of zero (Rita).

After these basic operations it’s very important to learn reading and writing data into Dataframes from various sources, which we’ll learn in the next chapter.

6 Reading and writing files

Reading and writing is a very necessary part of data science as the data generally will be in some sort of file like CSV or MS Excel.
Here we'll be working of various file formats to read and write data.

6.1 Reading CSV

Reading CSV is a straightforward thing, but we'll discuss some practical issues faced when dealing with the live data.

6.1.1 Reading

Let's look at the way to read a csv data:

df = pd.read_csv(<csvName>)

```
In [35]: df = pd.read_csv("name_age.csv")
         df
```

Out[35]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	na
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m
5	Radha	not available	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	NaN
8	Nidhi	29	10/01/91	f

6.1.2 Removing header

We can remove the headers present in the CSV if we don't need them.

df = pd.read_csv("name_age.csv", header=None)

6.1.3 Adding custom header

Sometimes we need to give our own names to the header or if the data does not have header present then we can add our own header by:

df = pd.read_csv("name_age.csv", names=["NAME", "AGE", "dateOfBirth",

"GENDER"])

```
In [39]: df = pd.read_csv("name_age.csv", names=["NAME", "AGE", "dateOfBirth", "GENDER"])
         df.head()
```

Out[39]:

	NAME	AGE	dateOfBirth	GENDER
0	name	age	dob	gender
1	Rita	23	20/02/97	f
2	Arun	29	01/07/91	m
3	Sita	14	07/07/06	na
4	Varun	21	01/05/99	m

You can see the difference that we have change the case and “dob” has been changed to “dateOfBirth”.

6.1.4 Reading specific rows

We can use head/tail etc. to access specific data in DataFrame but we can also read specific rows from CSV as:

df = pd.read_csv("name_age.csv", nrows=4)

```
In [42]: df = pd.read_csv("name_age.csv", nrows=4)
         df.head()
```

Out[42]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	na
3	Varun	21	01/05/99	m

The value in ‘nrows’ will be the number of rows we want to see, and this will except the header.

6.1.5 Reading the data from specific row

Sometimes there could be some text in the csv before the actual data starts like:

	A	B	C	D
1	name age data			
2				
3	name	age	dob	gender
4	Rita	23	20/02/97	f
5	Arun	29	01/07/91	m
6	Sita	14	07/07/06	na
7	Varun	21	01/05/99	m
8	Ram	32	07/11/88	m
9	Radha	not available	06/09/97	f
10	Mohan	51	03/03/69	m
11	Devi	20	05/01/00	
12	Nidhi	29	10/01/91	f

Here the actual data starts from row #3. If we'll read this data directly, we'll get garbage data into our DataFrame.

```
In [44]: df = pd.read_csv("name_age.csv")
         df.head()
```

Out[44]:

	name age data	Unnamed: 1	Unnamed: 2	Unnamed: 3
0	NaN	NaN	NaN	NaN
1	name	age	dob	gender
2	Rita	23	20/02/97	f
3	Arun	29	01/07/91	m
4	Sita	14	07/07/06	na

By default, the 1st columns become the header and the blank line becomes the 1st row, which we don't want.
What we can do is:

df = pd.read_csv("name_age.csv", skiprows=2)

This will skip the first two rows in the csv and start the data from 3rd row, which will then be considered as header.

```
In [45]: df = pd.read_csv("name_age.csv", skiprows=2)
         df.head()
```

Out[45]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	na
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m

The alternate way is to specify the row of the header as:

df = pd.read_csv("name_age.csv", header=2)

header=2 means the third row in csv as the indexing starts from 0.

```
In [46]: df = pd.read_csv("name_age.csv", header=2)
         df.head()
```

Out[46]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	na
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m

6.1.6 Cleaning NA data

Many a times in real data some cells might be empty. DataFrame consider that as NAN (not a number) which can be dealt with but if data has its own convention of telling NAN (like NA, not available etc.) then we have to convert them to NAN format so that we can clean them later.
This can be done by:

df = pd.read_csv("name_age.csv", na_values=["na", "not available"])

```
In [47]: df = pd.read_csv("name_age.csv", na_values=["na", "not available"])
         df
```

Out[47]:

	name	age	dob	gender
0	Rita	23.0	20/02/97	f
1	Arun	29.0	01/07/91	m
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	m
4	Ram	32.0	07/11/88	m
5	Radha	NaN	06/09/97	f
6	Mohan	51.0	03/03/69	m
7	Devi	20.0	05/01/00	NaN
8	Nidhi	29.0	10/01/91	f

Comparing the previous and latest data:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	na
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m
5	Radha	not available	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	NaN
8	Nidhi	29	10/01/91	f

	name	age	dob	gender
0	Rita	23.0	20/02/97	f
1	Arun	29.0	01/07/91	m
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	m
4	Ram	32.0	07/11/88	m
5	Radha	NaN	06/09/97	f
6	Mohan	51.0	03/03/69	m
7	Devi	20.0	05/01/00	NaN
8	Nidhi	29.0	10/01/91	f

Row 7 (Devi) already has gender column as NAN because the data was blank there.

Now for example if someone has it's name as "na" (let's assume) and we do the above cleaning then the name will become as NAN whereas the "na" name was correct name.
So, to avoid this and to apply cleaning only to particular columns we can use dictionaries as:

```
In [50]: df = pd.read_csv("name_age.csv", na_values = {
             "gender":["na", "not available"]
         })
         df
```

Out[50]:

	name	age	dob	gender
0	Rita	23	20/02/97	f
1	Arun	29	01/07/91	m
2	Sita	14	07/07/06	NaN
3	Varun	21	01/05/99	m
4	Ram	32	07/11/88	m
5	Radha	not available	06/09/97	f
6	Mohan	51	03/03/69	m
7	Devi	20	05/01/00	NaN
8	Nidhi	29	10/01/91	f

The cleaning has been applied to a specific column. Similarly, if the age is negative value (which is not possible) we can add a cleaning for age column and specify the negative value to be NAN.

6.1.7 Reference

https://Pandas.pydata.org/Pandas-docs/stable/reference/api/Pandas.read_csv.html

6.2 Writing to CSV

Now once we have cleaned the data or at any point of time you can write back to csv as:

df.to_csv(<fileName>)

```
In [53]: df.to_csv("newCSV.csv")
```

```
In [54]: 1 !ls "newCSV.csv"
```

```
newCSV.csv
```

Now if we read the file again, we can see that the file has stored the index

also, which sometimes we may not need.

6.2.1 Avoiding index

We have index (by default integers starting from 0) in every DataFrame. We may not need these indexes to be a part of our CSV, then we can avoid them. Generally, if we don't avoid the indexes, our CSV looks like:

```
In [55]: !cat "newCSV.csv"

,name,age,dob,gender
0,Rita,23.0,20/02/97,f
1,Arun,29.0,01/07/91,m
2,Sita,14.0,07/07/06,
3,Varun,21.0,01/05/99,m
4,Ram,32.0,07/11/88,m
5,Radha,,06/09/97,f
6,Mohan,51.0,03/03/69,m
7,Devi,20.0,05/01/00,
8,Nidhi,29.0,10/01/91,f
```

To avoid index to be stored in the csv we can do:

df.to_csv("newCSV.csv", index=False)

```
In [56]: df.to_csv("newCSV.csv", index=False)

In [57]:  1 !ls "newCSV.csv"

newCSV.csv

In [58]: !cat "newCSV.csv"

name,age,dob,gender
Rita,23.0,20/02/97,f
Arun,29.0,01/07/91,m
Sita,14.0,07/07/06,
Varun,21.0,01/05/99,m
Ram,32.0,07/11/88,m
Radha,,06/09/97,f
Mohan,51.0,03/03/69,m
Devi,20.0,05/01/00,
Nidhi,29.0,10/01/91,f
```

Now we don't have indexes in our csv.

6.2.2 Writing only specific columns

We can customize the column that we need to write back to the new csv by:

df.to_csv("newCSV.csv", columns=["name", "age"], index=False)

```
In [59]: df.to_csv("newCSV.csv", columns=["name", "age"], index=False)

In [60]:  1 !ls "newCSV.csv"
newCSV.csv

In [63]: df_new = pd.read_csv("newCSV.csv")
         df_new.head()
```

Out[63]:

	name	age
0	Rita	23.0
1	Arun	29.0
2	Sita	14.0
3	Varun	21.0
4	Ram	32.0

6.2.3 Avoid writing headers

We can avoid writing headers to the new csv as well by

df.to_csv("newCSV.csv", header=False)

6.2.4 reference

https://Pandas.pydata.org/Pandas-docs/stable/reference/api/Pandas.DataFrame.to_csv.html

6.3 Read Excel

We can get data in the for of excel too. Working with excel is similar to that of csv.

df_new = pd.read_excel("personsData.xlsx")

```
In [73]: df_new = pd.read_excel("personsData.xlsx")
         df_new.head()
```

Out[73]:

	name	age	dob	gender
0	Rita	23.0	20/02/97	f
1	Arun	29.0	01/07/91	m
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	m
4	Ram	32.0	07/11/88	m

6.3.1 Reading sheets

excel is known for sheets, by default the '*read_excel()*' takes the 1st sheet of the excel. But if we can specify specific sheet as well.

```
In [76]: df_new = pd.read_excel("personsData.xlsx", "new_data")
         df_new.head()
```

Out[76]:

	name	age	dob	gender
0	Michael	27	1993-09-09	m
1	anna	23	1997-11-03	f

6.3.2 Passing function to columns

We can customize the cell depending on our own need by writing the function and passing it as a parameter.

For example, we wrote a function to convert gender (from "m" to "Male" and "f" to "Female")

```
In [35]: def completeGender(cell):
             if cell=="m":
                 return "Male"
             elif cell=="f":
                 return "Female"
             else:
                 return "NA"
```

We can pass this function to the read_excel() as:

df_new = pd.read_excel("personsData.xlsx", converters = {
"gender" : completeGender
})

```
In [39]: df_new = pd.read_excel("personsData.xlsx", converters = {
             "gender" : completeGender
         })
         df_new.head()
```

Out[39]:

	name	age	dob	gender
0	Rita	23.0	20/02/97	Female
1	Arun	29.0	01/07/91	Male
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	Male
4	Ram	32.0	07/11/88	Male

We can supply the sheet name also:

df_new = pd.read_excel("personsData.xlsx", sheet_name = "data", converters = {
"gender" : completeGender
})

6.3.3 Some basic common functions in read_excel and read_csv are:

The use of some very basic parameters in excel and csv are same as: *'header'*, *'index'*, *'skiprows'*, *'nrows'*, *'na_values'* etc. which are covered above.

6.3.4 Reference

https://Pandas.pydata.org/Pandas-docs/stable/reference/api/Pandas.read_excel.html

6.4 Writing excels

Writing to excel is simple and is done by:

df_new.to_excel("newExcel.xlsx")

This by default takes the sheet name as "sheet1"

We can also provide our name to the sheet as:

6.4.1 Writing to a custom sheet name

df_new.to_excel("newExcel.xlsx", sheet_name="new_sheet")

6.4.2 Avoid index

df_new.to_excel("newExcel.xlsx", sheet_name="new_sheet", index=False)

6.4.3 Avoid headers

df_new.to_excel("newExcel.xlsx", sheet_name="new_sheet", header=False)

6.4.4 write at a particular row and column

df_new.to_excel("newExcel.xlsx", sheet_name="new_sheet", index=False, startrow=1, startcol=2)

the startrow and stratcol will write the DataFrame from that location in the excel (indexing starts from 0)

	A	B	C	D	E	F
1						
2			name	age	dob	gender
3			Rita	23	20/02/97	Female
4			Arun	29	01/07/91	Male
5			Sita	14	07/07/06	
6			Varun	21	01/05/99	Male
7			Ram	32	07/11/88	Male
8			Radha		06/09/97	Female
9			Mohan	51	03/03/69	Male
10			Devi	20	05/01/00	
11			Nidhi	29	10/01/91	Female

6.4.5 Writing multiple sheets to the same excel file

We may have to write multiple sheets in an Excel which can also be done in Pandas.
Let’s create two data frames and write that in different sheets of the Excel.

```
In [65]: df1 = pd.read_excel("personsData.xlsx", sheet_name = "data")
         df1.head()
```

Out[65]:

	name	age	dob	gender
0	Rita	23.0	20/02/97	f
1	Arun	29.0	01/07/91	m
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	m
4	Ram	32.0	07/11/88	m

```
In [66]: df2 = pd.read_excel("personsData.xlsx", sheet_name = "new_data")
         df2.head()
```

Out[66]:

	name	age	dob	gender
0	Michael	27	1993-09-09	m
1	anna	23	1997-11-03	f

And then using ExcelWriter() we can write to multiple sheets:

with pd.ExcelWriter("multipleSheet.xlsx") as writer:
df1.to_excel(writer, sheet_name="data1")
df2.to_excel(writer, sheet_name="data2")

```
In [71]: with pd.ExcelWriter("multipleSheet.xlsx") as writer:
             df1.to_excel(writer, sheet_name="data1", index=False)
             df2.to_excel(writer, sheet_name="data2", index=False)
```

```
In [72]: df = pd.read_excel("multipleSheet.xlsx", sheet_name = "data1")
         df
```

Out[72]:

	name	age	dob	gender
0	Rita	23.0	20/02/97	f
1	Arun	29.0	01/07/91	m
2	Sita	14.0	07/07/06	NaN
3	Varun	21.0	01/05/99	m
4	Ram	32.0	07/11/88	m
5	Radha	NaN	06/09/97	f
6	Mohan	51.0	03/03/69	m
7	Devi	20.0	05/01/00	NaN
8	Nidhi	29.0	10/01/91	f

```
In [73]: df = pd.read_excel("multipleSheet.xlsx", sheet_name = "data2")
         df
```

Out[73]:

	name	age	dob	gender
0	Michael	27	1993-09-09	m
1	anna	23	1997-11-03	f

we now have two different sheets in the same Excel file

6.4.6 reference

https://Pandas.pydata.org/Pandas-docs/stable/reference/api/Pandas.DataFrame.to_excel.html

6.5 Reading and writing txt file

The data can be provided in txt format also, but the important point is there has to be a delimiter in the data (whitespaces or tabs are also fine)

6.5.1 Reading txt

df = pd.read_csv("text.txt",delimiter="\t")

Any data with a delimiter becomes a csv and thus we can use read csv to read that data and provide a parameter *'delimiter'* to access that data. Here, the delimiter is "tab"

Similarly we can write to txt with to_csv() and a delimiter

6.5.2 Writing to txt

df.to_csv(“test.txt”, sep='\t', index=False)

6.6 Reference

We have covered the major I/O methods that used in daily life but there are few more which can be used. For further reference you can check the below link:

6.6.1 https://Pandas.pydata.org/Pandas-docs/stable/user_guide/io.html

The source data needs to be cleaned and after reading data we should be able to handle that data, most importantly the missing values, as it may cause issues in analysis. Let’s learn about handling the missing data once read from source, in the next chapter.

7 working with missing data

In real data, there will be lot of missing values, which needs to be dealt with before proceeding further. Let's see some of the practical cases that can arise in real life data.

Let's load our test data:

```
In [2]: df = pd.read_csv("weather.csv")
        df
```

Out[2]:

	date	temperature	windSpeed	status
0	20200506	35.6582	10.788378	sunny
1	20200509	NaN	NaN	NaN
2	20200510	30.9343	NaN	rainy
3	20200511	NaN	6.889682	cloudy
4	20200512	13.9082	19.012990	rainy
5	20200513	23.9382	NaN	sunny

7.1 Managing timestamp

In our data we can see the date column is in unreadable format. We have to first mat it as a timestamp.

df = pd.read_csv("weather.csv", parse_dates=["date"])

```
In [3]: df = pd.read_csv("weather.csv", parse_dates=["date"])
        df
```

Out[3]:

	date	temperature	windSpeed	status
0	2020-05-06	35.6582	10.788378	sunny
1	2020-05-09	NaN	NaN	NaN
2	2020-05-10	30.9343	NaN	rainy
3	2020-05-11	NaN	6.889682	cloudy
4	2020-05-12	13.9082	19.012990	rainy
5	2020-05-13	23.9382	NaN	sunny

7.2 Set index

The index in our data is integer which is not a part of our data. We need to set index which is our own data.

df.set_index('date', inplace=True)

```
In [4]: df.set_index('date', inplace=True)
        df
```

Out[4]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

7.3 Check if data is “na” or “notna”

To check weather our data is “na” or “notna” Pandas provide us with *isna()* and *notna().*

pd.isna(df['temperature'])

OR

df['temperature'].isna()

```
In [5]: pd.isna(df['temperature']) #df['temperature'].isna()
Out[5]: date
        2020-05-06    False
        2020-05-09     True
        2020-05-10    False
        2020-05-11     True
        2020-05-12    False
        2020-05-13    False
        Name: temperature, dtype: bool
```

df['windSpeed'].notna()

```
In [6]: df['windSpeed'].notna()
Out[6]: date
        2020-05-06     True
        2020-05-09    False
        2020-05-10    False
        2020-05-11     True
        2020-05-12     True
        2020-05-13    False
        Name: windSpeed, dtype: bool
```

We can do a similar thing on complete DataFrame and can create a new DataFrame to use it accordingly.

df.isna()

```
In [7]: df.isna()
Out[7]:
```

date	temperature	windSpeed	status
2020-05-06	False	False	False
2020-05-09	True	True	True
2020-05-10	False	True	False
2020-05-11	True	False	False
2020-05-12	False	False	False
2020-05-13	False	True	False

df.notna()

```
In [8]: df.notna()
```

Out[8]:

date	temperature	windSpeed	status
2020-05-06	True	True	True
2020-05-09	False	False	False
2020-05-10	True	False	True
2020-05-11	False	True	True
2020-05-12	True	True	True
2020-05-13	True	False	True

The is a point that should be noted that “One has to be mindful that in Pytho, the nan's don’t compare equal, but None's do. Note that Pandas uses the fact that np.nan != np.nan, and treats None like np.nan.”

```
In [9]: None == None
Out[9]: True

In [10]: np.nan == np.nan
Out[10]: False
```

So as compared to above, a scalar equality comparison versus a None/np.nan doesn’t provide useful information.

```
In [11]: df['temperature'] == np.nan
Out[11]: date
         2020-05-06    False
         2020-05-09    False
         2020-05-10    False
         2020-05-11    False
         2020-05-12    False
         2020-05-13    False
         Name: temperature, dtype: bool
```

7.4 Check if a data has missing datetime

For datetime format, the missing value is represented as “NaT”. we can check this by:
df.loc[[0,4,5],["date"]] = np.nan

with the above command we have made the 1st , 5th and 6th row of column date a nan.

```
In [13]: df.loc[[0,4,5],["date"]] = np.nan
         df
```

Out[13]:

	date	temperature	windSpeed	status
0	NaT	35.6582	10.788378	sunny
1	2020-05-09	NaN	NaN	NaN
2	2020-05-10	30.9343	NaN	rainy
3	2020-05-11	NaN	6.889682	cloudy
4	NaT	13.9082	19.012990	rainy
5	NaT	23.9382	NaN	sunny

We can see that the rows of column date, that we made nan is now showing as NaT (not a time).

With this let’s try inserting some missing data to the DataFrame.

7.5 Inserting missing date

Let’s insert missing date to a string series.

```
In [14]: s = pd.Series(["aa", "bb", "cc"])
         s.loc[0] = None
         s.loc[1] = np.nan
         s
```

```
Out[14]: 0    None
         1     NaN
         2      cc
         dtype: object
```

Here, we have added None to index 0 and nan to index 1 and the output is as expected.

Let’s try the same thing to a series of integer.

```
In [15]: s = pd.Series([11, 22, 33])
         s.loc[0] = None
         s.loc[1] = np.nan
         s

Out[15]: 0     NaN
         1     NaN
         2    33.0
         dtype: float64
```

We can see that though we inserted None to index 0, it shows it as NaN. So for missing values integer will always show NaN and same is the case with datetime, which shows NaT.

7.6 Filling the missing data

Let's try various issues and trick to fill the missing data into a DataFrame.

To fill the value for missing data, Pandas provide us with a method called *fillna()*

Our DataFrame is:

```
In [16]: df
```

Out[16]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

7.6.1 Filling a common value to all missing data

Let's try filling 0 to all the missing data

df.fillna(0)

```
In [17]: df.fillna(0)
```

Out[17]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	0.0000	0.000000	0
2020-05-10	30.9343	0.000000	rainy
2020-05-11	0.0000	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	0.000000	sunny

This works nice but 0 in status does not mean anything. i.e. different columns needs to be treated differently.

7.6.2 Adding missing data to individual columns

The same method can be used to add missing data for various columns differently. We just need to pass a dictionary as below.

```
In [18]: df.fillna({
             "temperature":0,
             "windSpeed":0,
             "status":"sunny"
         })
```

Out[18]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	0.0000	0.000000	sunny
2020-05-10	30.9343	0.000000	rainy
2020-05-11	0.0000	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	0.000000	sunny

We can access individual columns by an alternate way also.

```
In [19]: df["status"].fillna("new status")

Out[19]: date
         2020-05-06         sunny
         2020-05-09    new status
         2020-05-10         rainy
         2020-05-11        cloudy
         2020-05-12         rainy
         2020-05-13         sunny
         Name: status, dtype: object
```

7.6.3 Forward fill (row)

Forward fill is a method to forward the data from the row above the missing value. Thus all the missing value will get filled with the value above. If there are multiple missing values consecutively , they will also get filled with the same value of the above available data.

df.fillna(method="ffill")

note: pad / ffill = fill value forward

```
In [20]: df.fillna(method="ffill")

Out[20]:
```

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	35.6582	10.788378	sunny
2020-05-10	30.9343	10.788378	rainy
2020-05-11	30.9343	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	19.012990	sunny

We can see that all the missing values have been filled with the existing value above them.

7.6.4 Backward fill (row)

Similar to that for forward fill, backward fill also fills the data bus as the name suggest, this fills the data from back, i.e. from bottom.
So the missing data will be filled from the existing data below.

df.fillna(method="bfill")

note: bfill / backfill = fill value backward

```
In [21]: df.fillna(method="bfill")
```

Out[21]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	30.9343	6.889682	rainy
2020-05-10	30.9343	6.889682	rainy
2020-05-11	13.9082	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

7.6.5 Forward fill (column)

We can even forward fill the data in column wise also with the help of *'axis'* parameter.

df.fillna(method="ffill", axis="columns")

```
In [22]: df.fillna(method="ffill", axis="columns")
```

Out[22]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.7884	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	30.9343	rainy
2020-05-11	NaN	6.88968	cloudy
2020-05-12	13.9082	19.013	rainy
2020-05-13	23.9382	23.9382	sunny

Backward fill (column)
Backward fill can also be done in columns.

df.fillna(method="bfill", axis="columns")

```
In [23]: df.fillna(method="bfill", axis="columns")
```

Out[23]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.7884	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	rainy	rainy
2020-05-11	6.88968	6.88968	cloudy
2020-05-12	13.9082	19.0[illegible]	[illegible]iny
2020-05-13	23.9382	sunny	sunny

7.6.6 Limiting the forward/backward fill

We can limit the number of rows or columns getting filled.
df.fillna(method="ffill", limit=1)

```
In [24]: df.fillna(method="ffill", limit=1)
```

Out[24]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	35.6582	10.788378	sunny
2020-05-10	30.9343	NaN	rainy
2020-05-11	30.9343	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	19.012990	sunny

This limit’s the filling rows/columns.

```
In [20]: df.fillna(method="ffill")
```

Out[20]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	35.6582	10.788378	sunny
2020-05-10	30.9343	10.788378	rainy
2020-05-11	30.9343	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	19.012990	sunny

```
In [24]: df.fillna(method="ffill", limit=1)
```

Out[24]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	35.6582	10.788378	sunny
2020-05-10	30.9343	NaN	rainy
2020-05-11	30.9343	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	19.012990	sunny

The comparison makes this very clear. For windspeed column the data gets copies from 6^{th} to 9^{th} and 10^{th} in general but with limit the data gets copied only once i.e. from 6^{th} to 9^{th} .

7.6.7 Filling with Pandas objects

There are many Pandas objects like df.sum(), df.max(), etc. we can fill the missing values with these too.

df.fillna(df.mean())

```
In [25]: df.fillna(df.mean()) #only works for integer
```

Out[25]:

date	temperature	windSpeed	status
2020-05-06	35.658200	10.788378	sunny
2020-05-09	26.109725	12.230350	NaN
2020-05-10	30.934300	12.230350	rainy
2020-05-11	26.109725	6.889682	cloudy
2020-05-12	13.908200	19.012990	rainy
2020-05-13	23.938200	12.230350	sunny

Mean is most suitable here.

7.6.8 Filling for specific range of columns

We can do filling for a specific range of column too as:

df.fillna(df.mean()['temperature':'windSpeed'])

```
In [26]: df.fillna(df.mean()['temperature':'windSpeed'])
         # df.fillna(df.mean()['temperature':'temperature'])
```

Out[26]:

date	temperature	windSpeed	status
2020-05-06	35.658200	10.788378	sunny
2020-05-09	26.109725	12.230350	NaN
2020-05-10	30.934300	12.230350	rainy
2020-05-11	26.109725	6.889682	cloudy
2020-05-12	13.908200	19.012990	rainy
2020-05-13	23.938200	12.230350	sunny

Alternatively, with where clause.

df.where(pd.notna(df), df.mean(), axis='columns')

```
In [27]: df.where(pd.notna(df), df.mean(), axis='columns')
```

Out[27]:

date	temperature	windSpeed	status
2020-05-06	35.658200	10.788378	sunny
2020-05-09	26.109725	12.230350	NaN
2020-05-10	30.934300	12.230350	rainy
2020-05-11	26.109725	6.889682	cloudy
2020-05-12	13.908200	19.012990	rainy
2020-05-13	23.938200	12.230350	sunny

7.7 Interpolate missing value

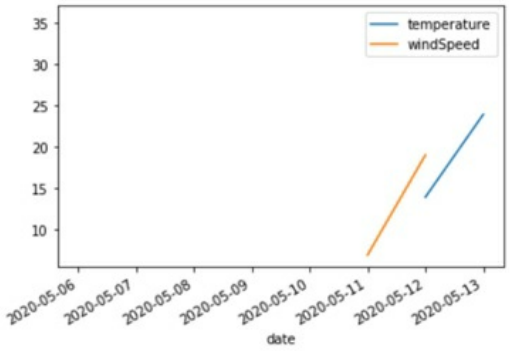
We can interpolate missing values based on different methods. This is done by an object in DataFrame as *interpolate()*. By default, interpolate() does linear
interpolation.
Let’s analyse our raw data first.

```
In [28]: df
```

Out[28]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

```
In [29]: df.plot()
Out[29]: <matplotlib.axes._subplots.AxesSubplot at 0x1152d71f0>
```



The data seems to be so much discontinuous due to the missing values. We need to fill something which makes the graph smooth which would look like some real value.

7.7.1 Linear interpolate

Let's try linear interpolation with our data

Linear interpolation involves estimating a new value by connecting two adjacent known values with a straight line.

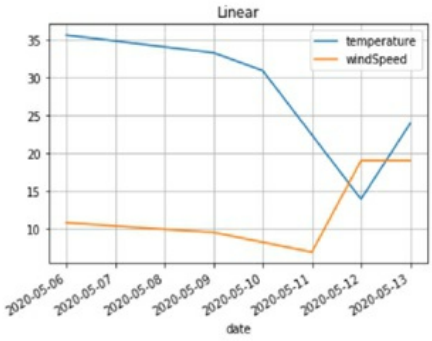
df.interpolate()

```
In [30]: df.interpolate()
```

Out[30]:

date	temperature	windSpeed	status
2020-05-06	35.65820	10.788378	sunny
2020-05-09	33.29625	9.488813	NaN
2020-05-10	30.93430	8.189248	rainy
2020-05-11	22.42125	6.889682	cloudy
2020-05-12	13.90820	19.012990	rainy
2020-05-13	23.93820	19.012990	sunny

```
In [31]: df.interpolate().plot(title="Linear", grid=1)
Out[31]: <matplotlib.axes._subplots.AxesSubplot at 0x111b8fe50>
```



Considering the value highlighted in red (temperature column and row for date 6th, 9th and 10th). The value at 9th was missing. Linear interpolation found the idle value for 9th so as to make a smooth and linear graph (can check in the plot for those dates.

7.7.2 Time interpolate

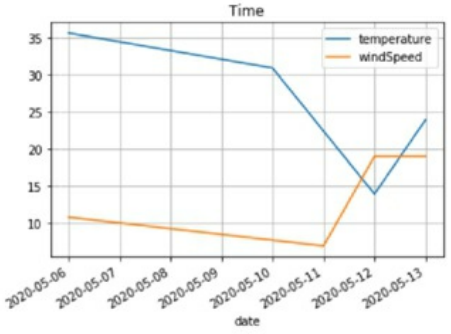
In the above data the graph and the value is fine but if we'll look at the date, there are few dates missing in between (7th and 8th). In this case getting a value for 9th which makes the plot linear is not that correct.
Here we can do time interpolation, which will resolve our issue.

df.interpolate(method="time")

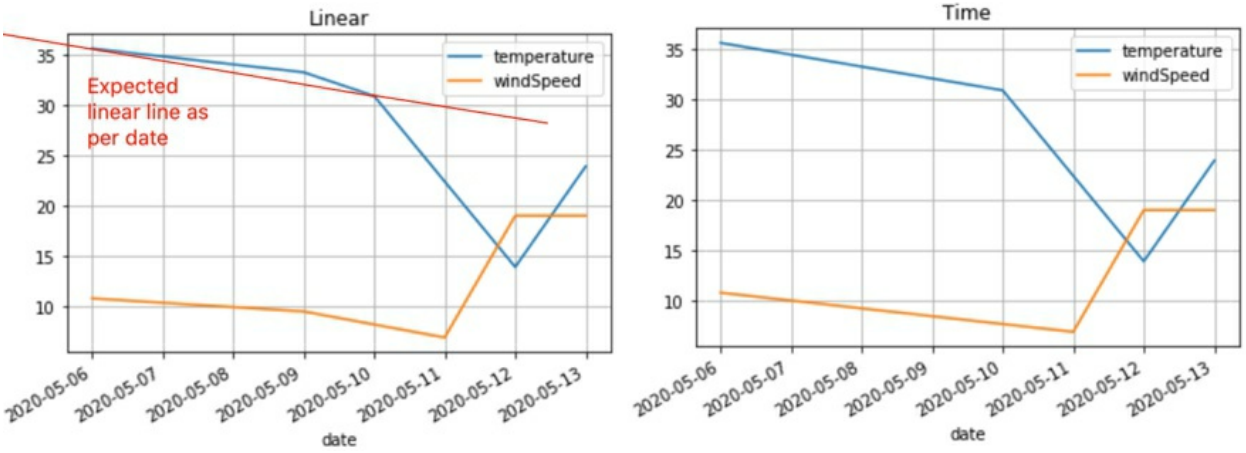
```
In [32]: df.interpolate(method="time")
Out[32]:
```

date	temperature	windSpeed	status
2020-05-06	35.658200	10.788378	sunny
2020-05-09	32.115275	8.449161	NaN
2020-05-10	30.934300	7.669422	rainy
2020-05-11	22.421250	6.889682	cloudy
2020-05-12	13.908200	19.012990	rainy
2020-05-13	23.938200	19.012990	sunny

```
In [33]: df.interpolate(method="time").plot(title="Time", grid=1)
Out[33]: <matplotlib.axes._subplots.AxesSubplot at 0x111cb2940>
```



7.7.2.1 Comparing linear and time plots



From above comparison we can see that the when considering the missing dates the time plot is much smoother as the value at 9th is better optimised.

7.7.3 Other methods of interpolation

Let's consider an integer DataFrame to discuss other methods of interpolate with date:

```
df1 = pd.DataFrame({'A': [2, 3.3, np.nan, 3.7, 7.6, 8.8],
                    'B': [1.25, np.nan, np.nan, 4.3, 14.23, 16.4]})
```

```
In [34]: df1 = pd.DataFrame({'A': [2, 3.3, np.nan, 3.7, 7.6, 8.8],
                             'B': [1.25, np.nan, np.nan, 4.3, 14.23, 16.4]})
         df1
```

Out[34]:

	A	B
0	2.0	1.25
1	3.3	NaN
2	NaN	NaN
3	3.7	4.30
4	7.6	14.23
5	8.8	16.40

7.7.3.1 Linear interpolation

df1.interpolate()

```
In [36]: df1.interpolate()
```

Out[36]:

	A	B
0	2.0	1.250000
1	3.3	2.266667
2	3.5	3.283333
3	3.7	4.300000
4	7.6	14.230000
5	8.8	16.400000

7.7.3.2 Barycentric interpolation

df1.interpolate(method='barycentric')

```
In [38]: df1.interpolate(method='barycentric')
```

Out[38]:

	A	B
0	2.00	1.25
1	3.30	-9.52
2	2.23	-6.06
3	3.70	4.30
4	7.60	14.23
5	8.80	16.40

7.7.3.3 Pchip interpolation

df1.interpolate(method='pchip')

```
In [40]: df1.interpolate(method='pchip')
```

Out[40]:

	A	B
0	2.000000	1.250000
1	3.300000	1.566495
2	3.488632	2.560768
3	3.700000	4.300000
4	7.600000	14.230000
5	8.800000	16.400000

7.7.3.4 Akima interpolation

df1.interpolate(method='akima')

```
In [42]: df1.interpolate(method='akima')
```

Out[42]:

	A	B
0	2.000000	1.250000
1	3.300000	-0.772951
2	3.444216	0.175210
3	3.700000	4.300000
4	7.600000	14.230000
5	8.800000	16.400000

7.7.3.5 Spline interpolation

df1.interpolate(method='spline', order=2)

```
In [44]: df1.interpolate(method='spline', order=2)
```

Out[44]:

	A	B
0	2.000000	1.250000
1	3.300000	-0.855010
2	3.440909	0.547068
3	3.700000	4.300000
4	7.600000	14.230000
5	8.800000	16.400000

7.7.3.6 Polynomial interpolation

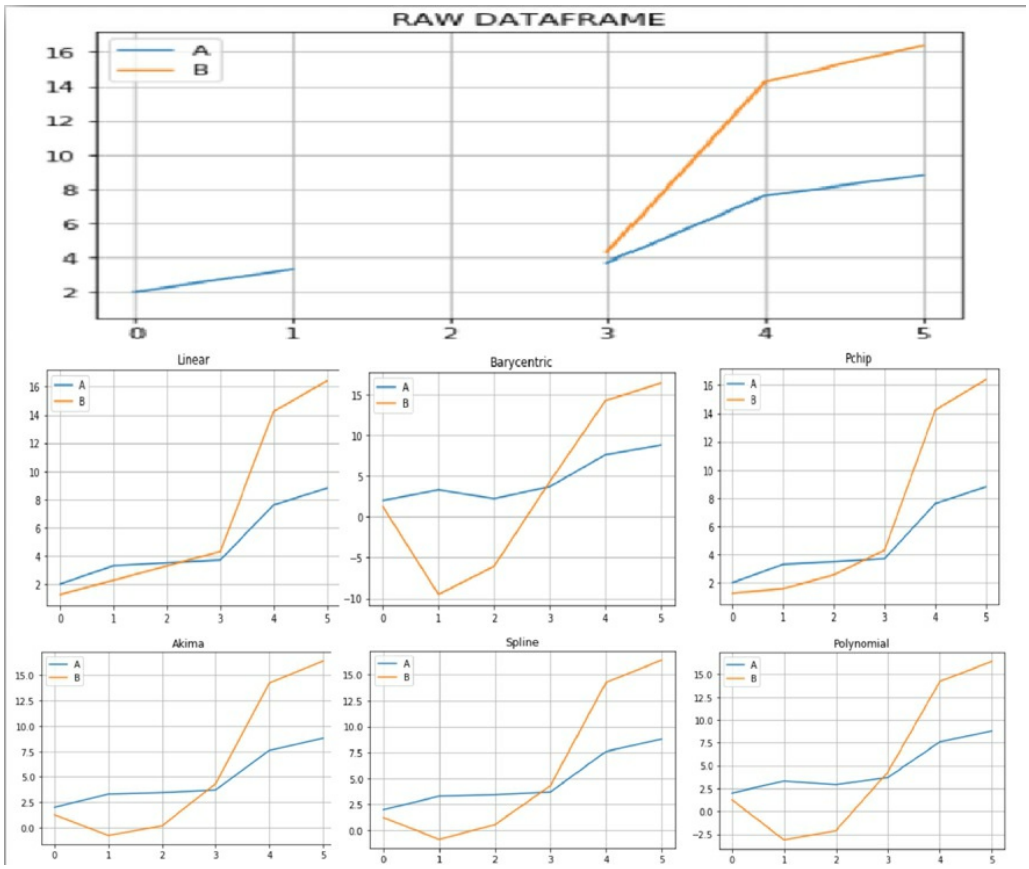
df1.interpolate(method='polynomial', order=2)

```
In [46]: df1.interpolate(method='polynomial', order=2)
```

Out[46]:

	A	B
0	2.000000	1.250000
1	3.300000	-3.129744
2	2.905405	-2.113077
3	3.700000	4.300000
4	7.600000	14.230000
5	8.800000	16.400000

7.7.3.7 Comparing all graphs for various methods



Thus depending on the data we can choose the method that we need for our need.

7.7.4 Limiting the interpolation

We can even provide a limit on how many rows, interpolation should be applied by:

df1.interpolate(limit=1)

```
In [48]: df1.interpolate(limit=1)
```

Out[48]:

	A	B
0	2.0	1.250000
1	3.3	2.266667
2	3.5	NaN
3	3.7	4.300000
4	7.6	14.230000
5	8.8	16.400000

7.7.5 Interpolation direction

Similar to *ffill* and *bfill* interpolation can also be directed.

df1.interpolate(limit=1, limit_direction='backward')

The *limit_direction* could also be both for both the direction and forward for forward direction.

```
In [49]: # limit_direction

         df1.interpolate(limit=1, limit_direction='backward')
```

Out[49]:

	A	B
0	2.0	1.250000
1	3.3	NaN
2	3.5	3.283333
3	3.7	4.300000
4	7.6	14.230000
5	8.8	16.400000

Here the index 2 has been updated because of index 4 (backward)

df1.interpolate(limit=1, limit_direction='backward')

```
In [50]: df1.interpolate(limit=1, limit_direction='both')
```

Out[50]:

	A	B
0	2.0	1.250000
1	3.3	2.266667
2	3.5	3.283333
3	3.7	4.300000
4	7.6	14.230000
5	8.8	16.400000

Here index 1 is updated because of index 0 and index 2 because of index 3 due to interpolation in both the direction and limit 1.

7.7.6 Limit area of interpolation

We can also restrict our missing value to be filled with inside or outside values

To analyse this let's take an example of series

dff = pd.Series([np.nan, np.nan, 35, np.nan, np.nan, np.nan, 55, np.nan, np.nan])

```
In [51]: dff = pd.Series([np.nan, np.nan, 35, np.nan, np.nan,
                          np.nan, 55, np.nan, np.nan])
         dff
```

```
Out[51]: 0     NaN
         1     NaN
         2    35.0
         3     NaN
         4     NaN
         5     NaN
         6    55.0
         7     NaN
         8     NaN
         dtype: float64
```

7.7.6.1 Inside

dff.interpolate(limit_direction='both', limit_area='inside', limit=1)

```
In [52]: dff.interpolate(limit_direction='both', limit_area='inside', limit=1)

Out[52]: 0     NaN
         1     NaN
         2    35.0
         3    40.0
         4     NaN
         5    50.0
         6    55.0
         7     NaN
         8     NaN
         dtype: float64
```

In above example we can see that the interpolation is done from both side between index 2 (35) and index 6 (55) i.e. ***inside***

7.7.6.2 Outside

dff.interpolate(limit_direction='both', limit_area='outside', limit=1)

```
In [53]: dff.interpolate(limit_direction='both', limit_area='outside', limit=1)

Out[53]: 0     NaN
         1    35.0
         2    35.0
         3     NaN
         4     NaN
         5     NaN
         6    55.0
         7    55.0
         8     NaN
         dtype: float64
```

In above example we can see that the interpolation is done from both side away from index 2 (35) and index 6 (55) i.e. ***outside***

7.8 Drop the missing value

We can remove or drop the missing value from the existing data. This process in real data science is done only if it's difficult or expensive to generate the missing data. The main reason to avoid this is removing the missing data (row) will lead to reduced training size and thus should only be practiced if we have sufficient data to train the model.

Let's use the weather DataFrame (df)

```
In [54]: df
```

Out[54]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-09	NaN	NaN	NaN
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

7.8.1 Drop row with at least 1 missing value

The dropping of the data with missing value is done by dropna() object of DataFrame.
By default dropna() will drop any row with at least one missing value.

df.dropna()

```
In [55]: df.dropna() # drops the row for atleast 1 nan
```

Out[55]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-12	13.9082	19.012990	rainy

We can see that in our data we have only 2 rows without any missing values. And thus rest all are droped.

7.8.2 Drop row with all missing values

We can remove only rows with all missing values also by:

df.dropna(how="all")

```
In [56]: # drop if finds all NaN in a row
         df.dropna(how="all") # date 9 is dropped
```

Out[56]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

Here we can see that all the values in row with date 9 was missing and thus dropped by the command

7.8.3 Set threshold to drop

we can set a threshold value to dropna(). If a threshold amount of data exist then that row will not be dropped
for example if we set the threshold as 2 then any row with 2 or more data will not be dropped but any row with 0 (no data) or 1 data will be dropped

df.dropna(thresh=1)

```
In [57]: df.dropna(thresh=1)
```

Out[57]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-10	30.9343	NaN	rainy
2020-05-11	NaN	6.889682	cloudy
2020-05-12	13.9082	19.012990	rainy
2020-05-13	23.9382	NaN	sunny

Threshold = 1
Thus, row with date 9 was dropped and it has no data.

```
In [58]: df.dropna(thresh=3)
```

Out[58]:

date	temperature	windSpeed	status
2020-05-06	35.6582	10.788378	sunny
2020-05-12	13.9082	19.012990	rainy

Threshold = 3
Thus, row with date 9 (no data), 10 (2 data), 11 (2 data) and 13 (2 data) are dropped. Only rows with 3 or more data will not be dropped i.e. row with date 6 and 12.

7.9 Replace the data

Sometimes we get the data but the data has some garbage or irrelevant form. Some data could be dealt better in other form for example a report cared data having performance scores as excellent, very good, good, fair and poor. Rather than this if we use rating as 1, 2, 3, 4 and 5 for excellent, very good, good, fair and poor respectively then it will be easy to analyse data. This is an example or personal choice.
But this is easy and appropriate way called mapping.

7.9.1 Replace a column with new column

Let's look at this by an example:

```
ranks = ['a', 'b', 'c', 'd']
names = ['Raju', 'Ramu', 'Priya', 'Sneha']

dfnew = pd.DataFrame(list(zip(names, ranks)), columns=["names", "ranks"])
```

```
In [59]: ranks = ['a', 'b', 'c', 'd']
         names = ['Raju', 'Ramu', 'Priya', 'Sneha']
```

```
In [60]: dfnew = pd.DataFrame(list(zip(names, ranks)),
                              columns=["names", "ranks"])
         dfnew
```

Out[60]:

	names	ranks
0	Raju	a
1	Ramu	b
2	Priya	c
3	Sneha	d

Here ranks as a, b, c, d doesn't looks good and doesn't make much sense.

We can replace the alphabets from numbers by replace().

dfnew = dfnew.replace(['a', 'b', 'c', 'd'],[1,2,3,4])

```
In [61]: #to change the ranks from alphabet to integer
         dfnew = dfnew.replace(['a', 'b', 'c', 'd'],[1,2,3,4])
         dfnew
```

Out[61]:

	names	ranks
0	Raju	1
1	Ramu	2
2	Priya	3
3	Sneha	4

The above data makes better sense now.

7.9.2 Replace with mapping dictionary

We can even use dictionary to specify the position which needs to be replaced

dfnew.replace({3: 10, 1: 100})

```
In [62]: dfnew.replace({3: 10, 1: 100})
```

Out[62]:

	names	ranks
0	Raju	100
1	Ramu	2
2	Priya	10
3	Sneha	4

Here we can see that the position 3 has been replaced with 10 and position 1 with 100.

Let’s look at some practical problems that may occur in real life data.

Let’s take a new data as:

```
In [63]: df_replace = pd.read_csv("weather_replace.csv")
         df_replace
```

Out[63]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	-1	xxxx	0
2	20200510	30.9343 c	xxxx	rainy
3	20200511	-1	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	0	-1	sunny

Here we can see that the data is corrupt as:
Temperature can’t be -1
windspeed can’t be -1 or “xxxx”
status should not be 0

we need to clean these data

7.9.3 replacing value with NaN

this is the main thing that we need to do while cleaning a data. If we convert the garbage values with NaN then we can further clean the data as discussed in above topics.

df_replace.replace("-1", np.nan)

```
In [64]: # temp cant be -1 so that needs to be nan
         df_replace.replace("-1", np.nan)
```

Out[64]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	NaN	xxxx	0
2	20200510	30.9343 c	xxxx	rainy
3	20200511	NaN	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	0	NaN	sunny

We successfully replaced all -1 with NaN's. but still we need to clean further.

7.9.4 Replace multiple values with NaN

The windspeed column still have "xxxx" present which we have to clean. This can be done by passing an array of data that needs to be cleaned.

df_replace.replace(["-1", "xxxx"], np.nan)

```
In [65]: # xxxx is also not a valid data.
         # to update multiple data in df we can

         df_replace.replace(["-1", "xxxx"], np.nan)
```

Out[65]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	NaN	NaN	0
2	20200510	30.9343 c	NaN	rainy
3	20200511	NaN	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	0	NaN	sunny

Alternatively, this can also be achieved by passing a dictionary.

df_replace.replace({"-1": np.nan, "xxxx": np.nan})

In [66]:
```
df_replace.replace({
    "-1": np.nan,
    "xxxx": np.nan,
})
```

Out[66]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	NaN	NaN	0
2	20200510	30.9343 c	NaN	rainy
3	20200511	NaN	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	0	NaN	sunny

Let's try to clean status also by the same way.

In [67]:
```
# but still what about 0 in status
# if we do
df_replace.replace(["-1", "xxxx", "0"], np.nan)
```

Out[67]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	NaN	NaN	NaN
2	20200510	30.9343 c	NaN	rainy
3	20200511	NaN	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	NaN	NaN	sunny

Thus we successfully manages status column also. But there is a problem here. This process removes the 0 in the temperature column also which is a valid data.

7.9.5 Replacing data as per columns

To resolve the above issue we can use a dictionary which will contain column names and the value that needs to be replaced.

df_new = df_replace.replace({"temperature": "-1", "windSpeed": ["xxxx",

"-1"], "status": '0', }, np.nan)

```
In [68]: df_new = df_replace.replace({
             "temperature": "-1",
             "windSpeed": ["xxxx", "-1"],
             "status": '0',
         }, np.nan)

         df_new
```

Out[68]:

	date	temperature	windSpeed	status
0	20200506	35.6582 c	10.788378 kmph	sunny
1	20200509	NaN	NaN	NaN
2	20200510	30.9343 c	NaN	rainy
3	20200511	NaN	6.8896825 kmph	cloudy
4	20200512	13.9082 c	19.01299 kmph	rainy
5	20200513	0	NaN	sunny

7.9.6 Regex and replace

We can see that some data has their unit's associated with them. We may have to get rid of them to proceed further. This can be achieved by the use of regex in replace().

df_new.replace('[A-Za-z]', '', regex=True)

```
In [69]: df_new.replace('[A-Za-z]', '', regex=True)
```

Out[69]:

	date	temperature	windSpeed	status
0	20200506	35.6582	10.788378	
1	20200509	NaN	NaN	NaN
2	20200510	30.9343	NaN	
3	20200511	NaN	6.8896825	
4	20200512	13.9082	19.01299	
5	20200513	0	NaN	

This replace has solved our problem or unit's associated with the values but removed everything from the status column.

7.9.7 Regex on specific columns

We can apply regex to solve our issues on specific columns too. This can be done by the use of a dictionary. We can pass the dictionary with column as key and corresponding regex as value.

df_new.replace({"temperature": '[A-Za-z]', "windSpeed":'[A-Za-z]'}, '', regex=True)

```
In [70]: df_new.replace({
             "temperature": '[A-Za-z]',
             "windSpeed":'[A-Za-z]',
         }, '', regex=True)
```

Out[70]:

	date	temperature	windSpeed	status
0	20200506	35.6582	10.788378	sunny
1	20200509	NaN	NaN	NaN
2	20200510	30.9343	NaN	rainy
3	20200511	NaN	6.8896825	cloudy
4	20200512	13.9082	19.01299	rainy
5	20200513	0	NaN	sunny

This solves our problem completely and we have completely removed all the garbage.

With this we are done with working with missing data and will work on other Pandas methods where we can group the similar data together in coming chapter.

8 Groupby

Many a times the data has some values, repeated for some different values like temperature on different months of an year. In that case the year will be same but the temperature for each month would change. In such cares we sometimes have to group the data together. In the above example we can group the years together, so that we can easily analyse the annual temperature together for each year.
Pandas provide us with such feature by *groupby()*. This grouping is very similar to that of groupby function in SQL database. Pandas allows us to group the things together and apply different functions on them which we'll learn in this chapter in detail.

Let us consider some students data for various subjects and their respective marks for two semesters.

```
In [1]: import pandas as pd
        import numpy as np
```

```
In [2]: students_df = pd.read_csv("students.csv")
        students_df
```

Out[2]:

	name	subject	sem1	sem2
0	Nisha	Physics	88	91
1	Arun	Physics	92	95
2	Neha	Physics	78	81
3	Varun	Physics	60	63
4	Nisha	Chemistry	61	64
5	Arun	Chemistry	72	75
6	Neha	Chemistry	82	85
7	Varun	Chemistry	59	62
8	Nisha	Maths	70	73
9	Arun	Maths	48	51
10	Neha	Maths	83	86
11	Varun	Maths	63	66
12	Nisha	Biology	71	74
13	Arun	Biology	84	87
14	Neha	Biology	57	60
15	Varun	Biology	71	74

In the above data few students (Nisha, Arun, Neha and Varun) are present with their marks in semester 1 and 2 for various subjects.

8.1 Creating group object

As we store the result of any operation into some variable so that we can use that variable further in the program, we store the group result in a variable. This variable acts as an object as groupby returns an object which can be used to do further operations on the group.

students = students_df.groupby(['name'])

```
In [3]: students = students_df.groupby(['name'])
        students

Out[3]: <pandas.core.groupby.generic.DataFrameGroupBy object at
        0x11a2994f0>
```

In the above example the groupby function has been applied on the column ‘name’ and created an object named ‘students’ which we’ll use further in the chapter for various functions.

8.2 Simple operations with group

Lest see some simple operations that can be applied on groups.

8.2.1 First

First gives us the 1st values of each group

students.first()

```
In [4]: students.first()
```

Out[4]:

name	subject	sem1	sem2
Arun	Physics	92	95
Neha	Physics	78	81
Nisha	Physics	88	91
Varun	Physics	60	63

8.2.2 Last

Similar to that of first, last gives us the last values of each group.

students.last()

```
In [5]: students.last()
```

Out[5]:

name	subject	sem1	sem2
Arun	Biology	84	87
Neha	Biology	57	60
Nisha	Biology	71	74
Varun	Biology	71	74

8.2.3 Max

Get the maximum value of the group

students['sem1'].max()

```
In [6]: students['sem1'].max()
```

```
Out[6]: name
        Arun     92
        Neha     83
        Nisha    88
        Varun    71
        Name: sem1, dtype: int64
```

8.2.4 Min

Get the minimum value of the group

students['sem1'].min()

```
In [7]: students['sem1'].min()
```

```
Out[7]: name
        Arun     48
        Neha     57
        Nisha    61
        Varun    59
        Name: sem1, dtype: int64
```

8.2.5 Mean

Get the mean/average of the group

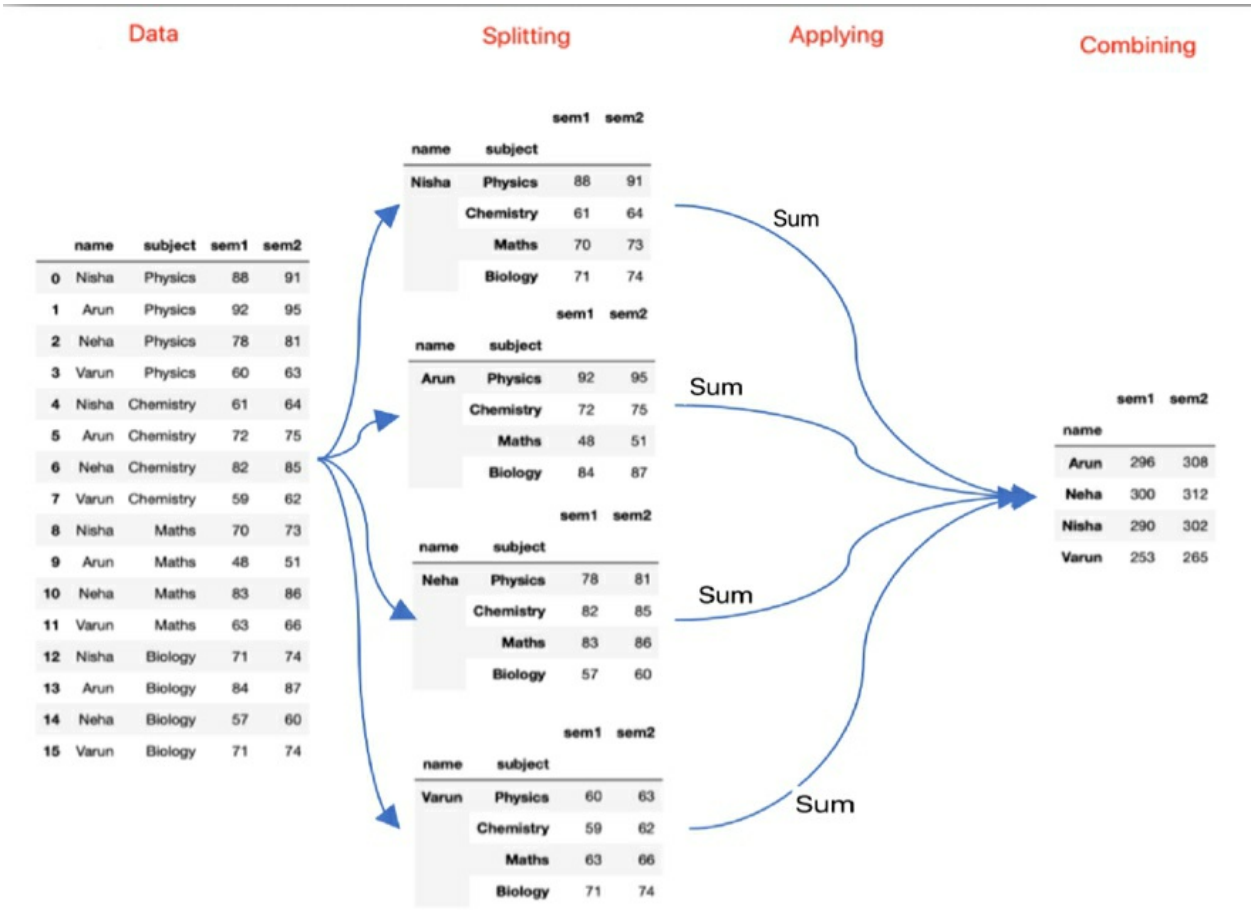
students['sem1'].mean()

```
In [8]: students['sem1'].mean()

Out[8]: name
        Arun     74.00
        Neha     75.00
        Nisha    72.50
        Varun    63.25
        Name: sem1, dtype: float64
```

8.3 Working of groupby

We have seen some very basic functions of groupby and now let's understand how it works.
A groupby operation
Takes the data
Split's the data
Apply a function (min, max, mean)
Combines the data

Let us understand this with a detailed diagram.



	name	subject	sem1	sem2
0	Nisha	Physics	88	91
1	Arun	Physics	92	95
2	Neha	Physics	78	81
3	Varun	Physics	60	63
4	Nisha	Chemistry	61	64
5	Arun	Chemistry	72	75
6	Neha	Chemistry	82	85
7	Varun	Chemistry	59	62
8	Nisha	Maths	70	73
9	Arun	Maths	48	51
10	Neha	Maths	83	86
11	Varun	Maths	63	66
12	Nisha	Biology	71	74
13	Arun	Biology	84	87
14	Neha	Biology	57	60
15	Varun	Biology	71	74

		sem1	sem2
name	subject		
Nisha	Physics	88	91
	Chemistry	61	64
	Maths	70	73
	Biology	71	74

		sem1	sem2
name	subject		
Arun	Physics	92	95
	Chemistry	72	75
	Maths	48	51
	Biology	84	87

		sem1	sem2
name	subject		
Neha	Physics	78	81
	Chemistry	82	85
	Maths	83	86
	Biology	57	60

		sem1	sem2
name	subject		
Varun	Physics	60	63
	Chemistry	59	62
	Maths	63	66
	Biology	71	74

	sem1	sem2
name		
Arun	296	308
Neha	300	312
Nisha	290	302
Varun	253	265

In the above diagram we can see that the data is taken and split into various groups. Now each group is acts as a new DataFrame and a function is applied on them. The resultant is calculated and combined together to for a single DataFrame again.

8.4 Iterate through groups

Once we create the groupby object we can use this iterate through various groups.

8.4.1 Group details

students.groups

```
In [9]: students.groups
```

```
Out[9]: {'Arun': Int64Index([1, 5, 9, 13], dtype='int64'),
         'Neha': Int64Index([2, 6, 10, 14], dtype='int64'),
         'Nisha': Int64Index([0, 4, 8, 12], dtype='int64'),
         'Varun': Int64Index([3, 7, 11, 15], dtype='int64')}
```

8.4.2 Iterate for groups

We can use the same object to iterate through the groups

for student, student_df in students:
    print(student)
    print(student_df)

```
In [10]: for student, student_df in students:
             print(student)
             print(student_df)
```

```
Arun
    name    subject  sem1  sem2
1   Arun    Physics    92    95
5   Arun  Chemistry    72    75
9   Arun      Maths    48    51
13  Arun    Biology    84    87
Neha
    name    subject  sem1  sem2
2   Neha    Physics    78    81
6   Neha  Chemistry    82    85
10  Neha      Maths    83    86
14  Neha    Biology    57    60
Nisha
     name    subject  sem1  sem2
0   Nisha    Physics    88    91
4   Nisha  Chemistry    61    64
8   Nisha      Maths    70    73
12  Nisha    Biology    71    74
Varun
     name    subject  sem1  sem2
3   Varun    Physics    60    63
7   Varun  Chemistry    59    62
11  Varun      Maths    63    66
15  Varun    Biology    71    74
```

8.5 Get a specific group

We can get the specific group and work on that

students_df.groupby(['name']).get_group('Neha')

```
In [11]: students_df.groupby(['name']).get_group('Neha')
```

Out[11]:

	name	subject	sem1	sem2
2	Neha	Physics	78	81
6	Neha	Chemistry	82	85
10	Neha	Maths	83	86
14	Neha	Biology	57	60

The above results gives us the complete data of the group ‘Neha’ which we can use to perform further operations.

8.6 Detailed view of the groups data

We have seen above that we can apply various operations like max, min, mean etc on the object. The same can be done with another function called ‘describe’. The describe() function provides us so much data all together.

students.describe()

```
In [12]: students.describe()
```

Out[12]:

	sem1								sem2							
	count	mean	std	min	25%	50%	75%	max	count	mean	std	min	25%	50%	75%	max
name																
Arun	4.0	74.00	19.183326	48.0	66.00	78.0	86.00	92.0	4.0	77.00	19.183326	51.0	69.00	81.0	89.00	95.0
Neha	4.0	75.00	12.192894	57.0	72.75	80.0	82.25	83.0	4.0	78.00	12.192894	60.0	75.75	83.0	85.25	86.0
Nisha	4.0	72.50	11.269428	61.0	67.75	70.5	75.25	88.0	4.0	75.50	11.269428	64.0	70.75	73.5	78.25	91.0
Varun	4.0	63.25	5.439056	59.0	59.75	61.5	65.00	71.0	4.0	66.25	5.439056	62.0	62.75	64.5	68.00	74.0

describe() could be very useful when we need to have an overview of the data.

8.7 Group by sorting

By default the result from groupby() is sorted but we can provide sort=False so that the operation does not sort the result. This will provide a potential

speedups.

8.7.1 Sorted data (default)

students_df.groupby(['name']).sum()

```
In [13]: students_df.groupby(['name']).sum()
```

Out[13]:

name	sem1	sem2
Arun	296	308
Neha	300	312
Nisha	290	302
Varun	253	265

8.7.2 Unsorted data

students_df.groupby(['name'], sort=False).sum()

```
In [14]: students_df.groupby(['name'], sort=False).sum()
```

Out[14]:

name	sem1	sem2
Nisha	290	302
Arun	296	308
Neha	300	312
Varun	253	265

From the above two outputs, we can see observe that the default feature of groupby provides a sorted result (with keys i.e. names)

8.8 Various functions associated with groupby object

We should be aware of various methods available with groupby object.
In Python we can check the methods associated with any object by:
dir(<object>)

Here, in this case we can get the methods by:

dir(students)

```
In [15]: dir(students)

Out[15]: '__annotations__, __class__, __delattr__, __dict__, __dir__, __doc__, __eq__, __format__, __ge__, __getattr__, __ge
         tattribute__, __getitem__, __gt__, __hash__, __init__, __init_subclass__, __iter__, __le__, __len__, __lt__, __modu
         le__, __ne__, __new__, __reduce__, __reduce_ex__, __repr__, __setattr__, __sizeof__, __str__, __subclasshook__, __w
         eakref__, _accessors, _add_numeric_operations, _agg_examples_doc, _agg_see_also_doc, _aggregate, _aggregate_frame,
         _aggregate_item_by_item, _aggregate_multiple_funcs, _apply_filter, _apply_to_column_groupbys, _apply_whitelist, _as
         sure_grouper, _bool_agg, _builtin_table, _choose_path, _concat_objects, _constructor, _cumcount_array, _cython_agg_
         blocks, _cython_agg_general, _cython_table, _cython_transform, _define_paths, _deprecations, _dir_additions, _dir_d
         eletions, _ensure_type, _fill, _get_cython_func, _get_cythonized_result, _get_data_to_aggregate, _get_index, _get_i
         ndices, _gotitem, _group_selection, _insert_inaxis_grouper_inplace, _internal_names, _internal_names_set, _is_built
         in_func, _iterate_column_groupbys, _iterate_slices, _make_wrapper, _obj_with_exclusions, _python_agg_general, _pyth
         on_apply_general, _reindex_output, _reset_cache, _reset_group_selection, _selected_obj, _selection, _selection_list
         , _selection_name, _set_group_selection, _set_result_index_ordered, _transform_fast, _transform_general, _transform
         _item_by_item, _transform_should_cast, _try_aggregate_string_function, _try_cast, _wrap_agged_blocks, _wrap_aggrega
         ted_output, _wrap_applied_output, _wrap_frame_output, _wrap_transformed_output, agg, aggregate, all, any, apply, ba
         ckfill, bfill, boxplot, corr, corrwith, count, cov, cumcount, cummax, cummin, cumprod, cumsum, describe, diff, dtyp
         es, expanding, ffill, fillna, filter, first, get_group, groups, head, hist, idxmax, idxmin, indices, last, mad, max
         , mean, median, min, name, ndim, ngroup, ngroups, nth, nunique, ohlc, pad, pct_change, pipe, plot, prod, quantile,
         rank, resample, rolling, sem, sem1, sem2, shift, size, skew, std, subject, sum, tail, take, transform, tshift, var'
```

This would be useful to know what method is available for your use.

8.9 Length

We can know the length of the group, i.e number of groups or the length of each group by:

8.9.1 Len of an object

We can find the number of groups formed in an object

len(students)

```
In [16]: len(students)

Out[16]: 4
```

8.9.2 Length of each group

We can get the length of each group element too

len(students.get_group('Arun'))

```
In [17]: len(students.get_group('Arun'))

Out[17]: 4
```

8.10 Groupby with multi-index

Date can be multi-indexed many times. We can access these data too using

Pandas

Let’s create multi index for our data.

students_df.sort_values(by=['name'], inplace=True)
students_df.set_index(['name', 'subject'], inplace=True)

now ‘name’ and ‘subject’ both are indexes to our data

```
In [18]: students_df.sort_values(by=['name'], inplace=True)
         students_df.set_index(['name', 'subject'], inplace=True)
         students_df
```

Out[18]:

		sem1	sem2
name	**subject**		
Arun	**Physics**	92	95
	Chemistry	72	75
	Maths	48	51
	Biology	84	87
Neha	**Physics**	78	81
	Chemistry	82	85
	Maths	83	86
	Biology	57	60
Nisha	**Physics**	88	91
	Chemistry	61	64
	Maths	70	73
	Biology	71	74
Varun	**Physics**	60	63
	Chemistry	59	62
	Maths	63	66
	Biology	71	74

We have created a DataFrame above with multiple indexes.
Now we can group this data, perform different operations on various levels of indexes.

8.10.1 Grouping on level numbers

Grouping in multi index can be done by level number (name = level 0 and subject = level 1)

grouped = students_df.groupby(level=0)
grouped = students_df.groupby(level=1)

```
In [19]: grouped = students_df.groupby(level=0)
         grouped.sum()
```

Out[19]:

name	sem1	sem2
Arun	296	308
Neha	300	312
Nisha	290	302
Varun	253	265

This will give sum of all the various subjects from each group (name)

```
In [20]: grouped = students_df.groupby(level=1)
         grouped.sum()
```

Out[20]:

subject	sem1	sem2
Biology	283	295
Chemistry	274	286
Maths	264	276
Physics	318	330

This will give the sum of marks for all the names for each subject

8.10.2 grouping on level names

grouping on multi indexed DataFrame can also be done as per the index name.

grouped = students_df.groupby(level='name')
grouped = students_df.groupby(level='subject')

```
In [21]: grouped = students_df.groupby(level="name")
         grouped.sum()
```

Out[21]:

name	sem1	sem2
Arun	296	308
Neha	300	312
Nisha	290	302
Varun	253	265

This will give sum of all the various subjects from each group (name)

```
In [22]: grouped = students_df.groupby(level="subject")
         grouped.sum()
```

Out[22]:

subject	sem1	sem2
Biology	283	295
Chemistry	274	286
Maths	264	276
Physics	318	330

This will give the sum of marks for all the names for each subject

8.11 Grouping DataFrame with index level and columns

With Pandas we can also group a combination of columns and index levels by:

students_df.groupby([pd.Grouper(level='name'), 'sem1']).sum()

```
In [23]: students_df.groupby([pd.Grouper(level='name'), 'sem1']).sum()
```

Out[23]:

		sem2
name	**sem1**	
Arun	**48**	51
	72	75
	84	87
	92	95
Neha	**57**	60
	78	81
	82	85
	83	86
Nisha	**61**	64
	70	73
	71	74
	88	91
Varun	**59**	62
	60	63
	63	66
	71	74

The grouper object is used to perform this function.
We can also do the same thing by:

students_df.groupby(['name', 'sem1']).sum()

```
In [24]: # alternatively
         students_df.groupby(['name', 'sem1']).sum()
```

Out[24]:

		sem2
name	**sem1**	
Arun	**48**	51
	72	75
	84	87
	92	95
Neha	**57**	60
	78	81
	82	85
	83	86
Nisha	**61**	64
	70	73
	71	74
	88	91
Varun	**59**	62
	60	63
	63	66
	71	74

Passing the index and the column as two elements of a list.

8.12 Aggregation

Aggregated function returns a single aggregated value for each group. Once the object is created we can use aggregation or agg to perform the aggregated operation i.e operations on each group.

For example, if we check the size with normal object we'll get the size of each group, but aggregated function gives the size of each group, column wise.

```
In [25]: students.size()
Out[25]: name
         Arun     4
         Neha     4
         Nisha    4
         Varun    4
         dtype: int64
```

```
In [26]: students.aggregate(np.size)
```

Out[26]:

name	subject	sem1	sem2
Arun	4	4	4
Neha	4	4	4
Nisha	4	4	4
Varun	4	4	4

```
In [27]: students.aggregate(np.mean)
```

Out[27]:

name	sem1	sem2
Arun	74.00	77.00
Neha	75.00	78.00
Nisha	72.50	75.50
Varun	63.25	66.25

This is how we perform a single aggregation function to the object.

8.12.1 Applying multiple aggregate functions at once

This is a great feature where we can apply multiple functions to the groupby object and can analyse them in a single command.

students.agg([np.sum, np.mean, np.std])

```
In [28]: students.agg([np.sum, np.mean, np.std])
```

Out[28]:

	sem1			sem2		
	sum	mean	std	sum	mean	std
name						
Arun	296	74.00	19.183326	308	77.00	19.183326
Neha	300	75.00	12.192894	312	78.00	12.192894
Nisha	290	72.50	11.269428	302	75.50	11.269428
Varun	253	63.25	5.439056	265	66.25	5.439056

8.12.2 Multiple aggregate function to selected columns

We can apply the aggregate functions to selected columns too.

students['sem1'].agg([np.sum, np.mean, np.std])

```
In [29]: students['sem1'].agg([np.sum, np.mean, np.std])
```

Out[29]:

	sum	mean	std
name			
Arun	274	68.50	10.661457
Neha	264	66.00	14.583095
Nisha	318	79.50	14.271183
Varun	283	70.75	11.026483

8.12.3 Renaming the column names for aggregate functions

Based on above example we can observe that the name for the columns are getting defaulted. We can give custom names to our aggregate functions columns too by passing a dictionary with key as default column name and value as custom column name to the rename function.

students['sem1'].agg([np.sum, np.mean, np.std]).rename(columns={
"sum": "total",
"mean": "average",
"std": "standardDeviation"

})

```
In [30]: students['sem1'].agg([np.sum, np.mean, np.std]).rename(columns={
             "sum": "total",
             "mean": "average",
             "std": "standardDeviation"
         })
```

Out[30]:

name	total	average	standardDeviation
Arun	274	68.50	10.661457
Neha	264	66.00	14.583095
Nisha	318	79.50	14.271183
Varun	283	70.75	11.026483

Now we have our own defined column names.

8.12.4 Named aggregation

If we want to apply aggregated functions to different columns and provide custom name too, we can use named aggregation.

```
In [31]: students.agg(
             sem1_min_marks=pd.NamedAgg(column='sem1', aggfunc='max'),
             sem2_max_marks=pd.NamedAgg(column='sem2', aggfunc='max'),
             sem1_avg_marks=pd.NamedAgg(column='sem1', aggfunc='mean'),
             sem2_avg_marks=pd.NamedAgg(column='sem2', aggfunc='mean'),
         )
```

Out[31]:

name	sem1_min_marks	sem2_max_marks	sem1_avg_marks	sem2_avg_marks
Arun	82	85	68.50	71.50
Neha	83	86	66.00	69.00
Nisha	92	95	79.50	82.50
Varun	84	87	70.75	73.75

Here, we have performed the aggregation function on various columns and with our own custom column names.

The object “NamedAgg” takes 1st argument as the column name and 2nd as the function that needs to be performed.
Thus the query can be re-written as:

```
students.agg(
    min_marks=pd.NamedAgg('sem1', 'min'),
    max_marks=pd.NamedAgg('sem1', 'max'),
    avg_marks=pd.NamedAgg('sem1', 'mean'),
)
```

```
In [32]: # alternatively

         students.agg(
             min_marks=pd.NamedAgg('sem1', 'min'),
             max_marks=pd.NamedAgg('sem1', 'max'),
             avg_marks=pd.NamedAgg('sem1', 'mean'),
         )
```

Out[32]:

name	min_marks	max_marks	avg_marks
Arun	59	82	68.50
Neha	48	83	66.00
Nisha	60	92	79.50
Varun	57	84	70.75

8.12.5 Custom agg function on various columns

We can perform a custom function to the columns too by:

```
students.agg({
    "sem1": "sum",
    "sem2": lambda x: np.std(x, ddof=1),
})
```

```
In [33]: students.agg({
             "sem1": "sum",
             "sem2": lambda x: np.std(x, ddof=1),
         })
```

Out[33]:

name	sem1	sem2
Arun	274	10.661457
Neha	264	14.583095
Nisha	318	14.271183
Varun	283	11.026483

We have used “lambda function” to sem2 column.

8.13 Transformation

Transformation is a method applied to the groupby object that returns a result that has the same size to that of the original data that’s being grouped.
Generally whenever we apply a function to a groupby object, it returns a result (a row) for each group. Thus the data gets reduced.

Let’s look at an example.

```
In [34]: students2_df = students_df.reset_index()
         students2 = students2_df.groupby('name')
```

```
In [35]: students2.mean()
```

Out[35]:

name	sem1	sem2
Arun	74.00	77.00
Neha	75.00	78.00
Nisha	72.50	75.50
Varun	63.25	66.25

This give a mean result of a groupby object which is grouped by name. The result here is 4 rows (one for each group)

```
In [36]: students2.transform(np.mean)
```

Out[36]:

	sem1	sem2
0	74.00	77.00
1	74.00	77.00
2	74.00	77.00
3	74.00	77.00
4	75.00	78.00
5	75.00	78.00
6	75.00	78.00
7	75.00	78.00
8	72.50	75.50
9	72.50	75.50
10	72.50	75.50
11	72.50	75.50
12	63.25	66.25
13	63.25	66.25
14	63.25	66.25
15	63.25	66.25

Here, we have applied a transformation on the groupby object for mean. The result contain 16 rows same as the original data.

```
In [37]: len(students2_df)
Out[37]: 16

In [38]: len(students2.mean())
Out[38]: 4

In [39]: len(students2.transform(np.mean))
Out[39]: 16
```

The original data contains 16 rows.

Normal mean on the groupby object gives 4 rows
Transformation method provides 16 rows (same as original data)

This could be used in data science for feature extraction mainly in image processing.
Can also have a practical advantage on filling the missing data as the size is same a original DataFrame.

The most important point to note here that when using fillna(), the inplace must be False to avoid the change in the original DataFrame. This may cause unexpected results.

8.13.1 Custom functions in transformation

We can provide our own custom functions to Transformation method as well.

score2 = lambda x: (x.max()-x.min())
students2.transform(score2)

```
In [40]: score = lambda x: (x - x.mean()) / x.std()*10
         score2 = lambda x: (x.max()-x.min())

In [41]: students2.transform(score2)
```

Out[41]:

	sem1	sem2
0	44	44
1	44	44
2	44	44
3	44	44
4	26	26
5	26	26
6	26	26
7	26	26
8	27	27
9	27	27
10	27	27
11	27	27
12	12	12
13	12	12
14	12	12
15	12	12

8.13.2 Filling missing data

If we have missing data we can use transformation to fill that data too.

score3 = lambda x: x.fillna(x.mean())
students2.transform(score3)

```
In [42]: score3 = lambda x: x.fillna(x.mean())
         students2.transform(score3)
```

Out[42]:

	sem1	sem2
0	92	95
1	72	75
2	48	51
3	84	87
4	78	81
5	82	85
6	83	86
7	57	60
8	88	91
9	61	64
10	70	73
11	71	74
12	60	63
13	59	62
14	63	66
15	71	74

In this example we don’t have any missing data but this works as expected and fill the mean() value of the column to the missing places.

8.14 Window operations

8.14.1 Rolling

This is a method which provides feature of rolling window calculation. This is generally used in signal processing and time series data.

```
In [43]: # rolling

         students2_df.groupby('name').rolling(4).sem1.sum()
```

```
Out[43]: name
         Arun   0       NaN
                1       NaN
                2       NaN
                3     296.0
         Neha   4       NaN
                5       NaN
                6       NaN
                7     300.0
         Nisha  8       NaN
                9       NaN
                10      NaN
                11    290.0
         Varun  12      NaN
                13      NaN
                14      NaN
                15    253.0
         Name: sem1, dtype: float64
```

This takes the '*rolling()*' as 4 and provided the sum of sem1 for each group. Let's take group "Arun". The sum of 1st, 2nd, 3th and 4th data is calculated and shown the result of 4th as that was passed as value to '*rollong()*'. You can try for other rolling values.

8.14.2 Expanding

Rolling operations are for fixed window sizes. However, this operation is an expanding window size.
Example:

Suppose you want to predict the weather, you have 1000 days of data:

Rolling: let's say window size is 10. For first prediction, it will use (the previous) 10 days of data and predict day 11. For next prediction, it will use the 2nd day (data point) to 11th day of data.

Expanding: For first prediction it will use 10 days of data. However, for second prediction it will use 10 + 1 days of data. The window has therefore named "expanded."

```
In [44]: # expanding

         students2_df.groupby('name').expanding().sum()
```

Out[44]:

		sem1	sem2
name			
Arun	**0**	92.0	95.0
	1	164.0	170.0
	2	212.0	221.0
	3	296.0	308.0
Neha	**4**	78.0	81.0
	5	160.0	166.0
	6	243.0	252.0
	7	300.0	312.0
Nisha	**8**	88.0	91.0
	9	149.0	155.0
	10	219.0	228.0
	11	290.0	302.0
Varun	**12**	60.0	63.0
	13	119.0	125.0
	14	182.0	191.0
	15	253.0	265.0

8.15 Filtration

The filter() function is used to return the subset rows or columns of DataFrame according to labels in the specified index.

students_df.filter(like='run', axis=0)

```
In [45]: students_df.filter(like='run', axis=0)
```

Out[45]:

		sem1	sem2
name	**subject**		
Arun	**Physics**	92	95
	Chemistry	72	75
	Maths	48	51
	Biology	84	87
Varun	**Physics**	60	63
	Chemistry	59	62
	Maths	63	66
	Biology	71	74

We can see that a filter has been applied on the DataFrame and provided us the filtered result fur filter “run”. Arun and Varun contains “run”.

Another example,
f*iltering = pd.Series([10, 11, 12, 13, 14, 15])*
filtering.groupby(filtering).filter(lambda x: x.sum() > 12)

```
In [46]: filtering = pd.Series([10, 11, 12, 13, 14, 15])

In [47]: filtering.groupby(filtering).filter(lambda x: x.sum() > 12)
Out[47]: 3    13
         4    14
         5    15
         dtype: int64
```

A filter was applied on the series to provide the value greater than 12 and found the result accordingly.

8.16 Instance methods

Let’s discuss some of the instance methods used in Pandas to make the process fast and easy

8.16.1 Sum, mean, max, min etc

students.sum()

```
In [48]: students.sum()
```

Out[48]:

name	sem1	sem2
Arun	296	308
Neha	300	312
Nisha	290	302
Varun	253	265

Provides the sum of the groupby object

8.16.2 Fillna

fillna() could be directly used to the groupby object

students.fillna(method='pad')

```
In [49]: students.fillna(method='pad')
```

Out[49]:

		sem1	sem2
name	**subject**		
Arun	**Physics**	92	95
	Chemistry	72	75
	Maths	48	51
	Biology	84	87
Neha	**Physics**	78	81
	Chemistry	82	85
	Maths	83	86
	Biology	57	60
Nisha	**Physics**	88	91
	Chemistry	61	64
	Maths	70	73
	Biology	71	74
Varun	**Physics**	60	63
	Chemistry	59	62
	Maths	63	66
	Biology	71	74

8.16.3 Fetching nth row

We can fetch the nth row of each groups by:

students.nth(1)
the above command will fetch the 1st row of all the groups.

```
In [50]: students.nth(1)
```

Out[50]:

name	sem1	sem2
Arun	82	85
Neha	83	86
Nisha	78	81
Varun	57	60

8.17 Apply

Some operations might not fit into aggregation or transformation. In such cases we can use apply().
It takes function and applies on every single value of the group.

```
def f(group):
    return pd.DataFrame({
        'original': group,
        'reduced': group - group.mean()
    })
students['sem1'].apply(f)
```

here we can see that we can apply any custom function to the columns.

```
In [51]: def f(group):
             return pd.DataFrame({
                 'original': group,
                 'reduced': group - group.mean()
             })
```

```
In [52]: students['sem1'].apply(f)
```

Out[52]:

		original	reduced
name	**subject**		
Arun	**Physics**	92	12.50
	Chemistry	72	3.50
	Maths	48	-18.00
	Biology	84	13.25
Neha	**Physics**	78	-1.50
	Chemistry	82	13.50
	Maths	83	17.00
	Biology	57	-13.75
Nisha	**Physics**	88	8.50
	Chemistry	61	-7.50
	Maths	70	4.00
	Biology	71	0.25
Varun	**Physics**	60	-19.50
	Chemistry	59	-9.50
	Maths	63	-3.00
	Biology	71	0.25

8.18 Plotting

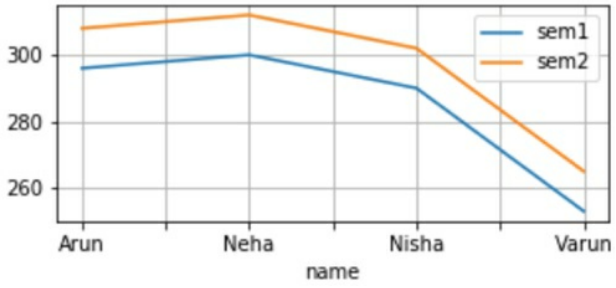
We can plot the columns in the DataFrame as per group as we have plot and boxplot function available.

8.18.1 Lineplot

The plot works as in matplotlib.

```
In [53]: import matplotlib.pyplot as plt
         %matplotlib inline
         fig, ax = plt.subplots(figsize=(5,2))
         students_df.groupby(['name']).sum().plot(grid=1, ax=ax)
```

```
Out[53]: <matplotlib.axes._subplots.AxesSubplot at 0x114e56c70>
```



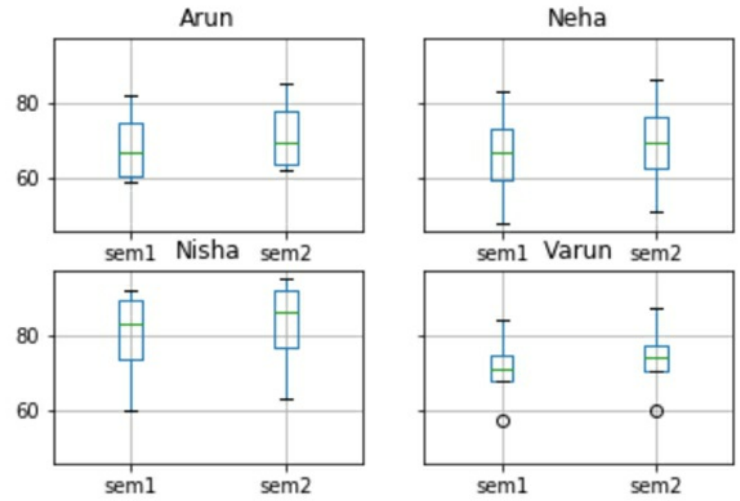
8.18.2 Boxplot

Boxplot is simple and gives a great information. The boxplot with groupby object provides a boxplot for each group.

Boxplot for groupby object

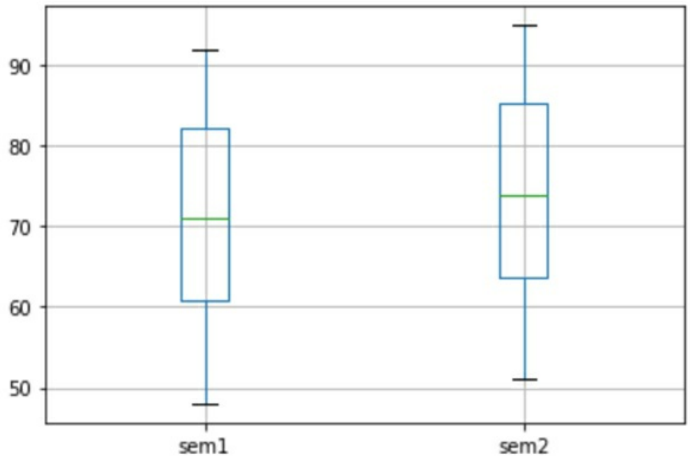
```
In [54]: students.boxplot()
```

```
Out[54]: Arun          AxesSubplot(0.1,0.559091;0.363636x0.340909)
         Neha      AxesSubplot(0.536364,0.559091;0.363636x0.340909)
         Nisha           AxesSubplot(0.1,0.15;0.363636x0.340909)
         Varun        AxesSubplot(0.536364,0.15;0.363636x0.340909)
         dtype: object
```



Boxplot or DataFrame

```
In [55]: students_df.boxplot()
Out[55]: <matplotlib.axes._subplots.AxesSubplot at 0x114022400>
```



Important point to note here is that the boxplot for groupby object is different from the boxplot from DataFrame as a plot for each group is plotted for groupby object.

Let’s look at process of concatenation in the next chapter.

9 Concatenation

Many a times we don't get the desired data from a single source which lead to creation of multiple DataFrames for each source. To bring these data together for further processing we need to get them together. Pandas provide us with an operation called "*concat*" for the same.

Concatenation is a very important and frequently used operation in real scenarios.

Let's look at some working examples.

9.1 Concatenate series

Let's start with series concatenation

arun = pd.Series(['a', 'r', 'u', 'n'])
neha = pd.Series(['n', 'e', 'h', 'a'])
pd.concat([arun, neha])

```
In [1]: import pandas as pd
        import numpy as np
```

```
In [2]: arun = pd.Series(['a', 'r', 'u', 'n'])
        neha = pd.Series(['n', 'e', 'h', 'a'])
        pd.concat([arun, neha])
```

```
Out[2]: 0    a
        1    r
        2    u
        3    n
        0    n
        1    e
        2    h
        3    a
        dtype: object
```

9.2 Concatenate DataFrames

Let's have to look on the concatenation if the DataFrames.

arun_scores = {
"subjects": ['maths', 'physics', 'chemistry', 'biology'],
"sem1":[60, 70, 80, 90],

"sem2":[63, 71, 85, 89],
}

neha_scores = {
"subjects": ['maths', 'physics', 'chemistry', 'computers'],
"sem1":[63, 77, 89, 92],
"sem2":[73, 81, 88, 83],
}

The above two DataFrames are two test DataFrames we are using as examples

In [3]:
```
arun_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'biology'],
    "sem1":[60, 70, 80, 90],
    "sem2":[63, 71, 85, 89],
}
arun_df = pd.DataFrame(arun_scores)
arun_df
```

Out[3]:

	subjects	sem1	sem2
0	maths	60	63
1	physics	70	71
2	chemistry	80	85
3	biology	90	89

In [4]:
```
neha_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'computers'],
    "sem1":[63, 77, 89, 92],
    "sem2":[73, 81, 88, 83],
}
neha_df = pd.DataFrame(neha_scores)
neha_df
```

Out[4]:

	subjects	sem1	sem2
0	maths	63	73
1	physics	77	81
2	chemistry	89	88
3	computers	92	83

Let’s concatenate this

df = pd.concat([arun_df, neha_df])

```
In [5]: df = pd.concat([arun_df, neha_df])
        df
```

Out[5]:

	subjects	sem1	sem2
0	maths	60	63
1	physics	70	71
2	chemistry	80	85
3	biology	90	89
0	maths	63	73
1	physics	77	81
2	chemistry	89	88
3	computers	92	83

9.3 Managing duplicate index

We can see that the concatenation of the DataFrame has been successfully done but the indexes are also been copied. We may not need the duplicate index. This can be resolved as:

df = pd.concat([arun_df, neha_df], ignore_index=True)

```
In [6]: df = pd.concat([arun_df, neha_df], ignore_index=True)
        df
```

Out[6]:

	subjects	sem1	sem2
0	maths	60	63
1	physics	70	71
2	chemistry	80	85
3	biology	90	89
4	maths	63	73
5	physics	77	81
6	chemistry	89	88
7	computers	92	83

The duplicate index are now gone as we ignored the indexes while concatenating by, ignore_index=True (False by default).

9.4 Adding keys to DataFrames

As we have combined the DataFrames ignoring the indexes, it's difficult to recognise the DataFrame from which they have been taken.
This issue can be taken care as:

df = pd.concat([arun_df, neha_df], keys=["arun", "neha"])

```
In [7]: df = pd.concat([arun_df, neha_df], keys=["arun", "neha"])
        df
```

Out[7]:

		subjects	sem1	sem2
arun	0	maths	60	63
	1	physics	70	71
	2	chemistry	80	85
	3	biology	90	89
neha	0	maths	63	73
	1	physics	77	81
	2	chemistry	89	88
	3	computers	92	83

As we can see that the keys provides an additional index to the final DataFrame and this additional index helps us to recognise the original DataFrame or the source too.

9.5 Use of keys

With the new index we generated with the help of keys, we can access or work on the specific data of a source in the final DataFrame.

df.loc['arun']

```
In [8]: df.loc['arun']
```

Out[8]:

	subjects	sem1	sem2
0	maths	60	63
1	physics	70	71
2	chemistry	80	85
3	biology	90	89

9.6 Adding DataFrame as a new column

Until now, we have seen that the concatenation of the DataFrames add by rows by default.

arun_sem3_scores = {
"subjects": ['maths', 'physics', 'chemistry', 'biology'],
"sem3":[50, 64, 88, 81],
}

```
In [9]: arun_sem3_scores = {
            "subjects": ['maths', 'physics', 'chemistry', 'biology'],
            "sem3":[50, 64, 88, 81],
        }
        df_additional = pd.DataFrame(arun_sem3_scores)
        df_additional
```

Out[9]:

	subjects	sem3
0	maths	50
1	physics	64
2	chemistry	88
3	biology	81

df = pd.concat([arun_df, df_additional])

```
In [10]: df = pd.concat([arun_df, df_additional])
         df
```

Out[10]:

	subjects	sem1	sem2	sem3	
0	maths	60.0	63.0	NaN	first DataFrame
1	physics	70.0	71.0	NaN	
2	chemistry	80.0	85.0	NaN	
3	biology	90.0	89.0	NaN	
0	maths	NaN	NaN	50.0	Second DataFrame
1	physics	NaN	NaN	64.0	
2	chemistry	NaN	NaN	88.0	
3	biology	NaN	NaN	81.0	

If we need the same data to be a new column we can provide a parameter *'axis=1'* (0 by default)

df = pd.concat([arun_df, df_additional], axis=1)

```
In [11]: df = pd.concat([arun_df, df_additional], axis=1)
         df
```

Out[11]:

	subjects	sem1	sem2	subjects	sem3
0	maths	60	63	maths	50
1	physics	70	71	physics	64
2	chemistry	80	85	chemistry	88
3	biology	90	89	biology	81

9.6.1 Removing unwanted columns in column concatenation

We can see that the subject column has been repeated above which is not needed and we only need the data for sem3 column. We can do the same as:

df = pd.concat([arun_df, df_additional['sem3']], axis=1)

```
In [12]: df = pd.concat([arun_df, df_additional['sem3']], axis=1)
         df
```

Out[12]:

	subjects	sem1	sem2	sem3
0	maths	60	63	50
1	physics	70	71	64
2	chemistry	80	85	88
3	biology	90	89	81

9.6.2 Series in columns

We can concatenate series as well in columns by:

pd.concat([arun, neha], axis=1)

```
In [13]: arun
Out[13]: 0    a
         1    r
         2    u
         3    n
         dtype: object

In [14]: neha
Out[14]: 0    n
         1    e
         2    h
         3    a
         dtype: object

In [15]: pd.concat([arun, neha], axis=1)
Out[15]:
```

	0	1
0	a	n
1	r	e
2	u	h
3	n	a

9.7 Rearranging the order of column

When two data frames have same column like subject in our case. If two DataFrames have same subject columns but the subjects are not in same order, then normal concatenation give a weird result which may not be useful in real world.

Let’s take an example

```
In [16]: arun_sem3_scores = {
             "subjects": ['physics', 'chemistry', 'maths', 'biology'],
             "sem3":[50, 64, 88, 81],
         }
         df_additional = pd.DataFrame(arun_sem3_scores)
         df_additional
```

Out[16]:

	subjects	sem3
0	physics	50
1	chemistry	64
2	maths	88
3	biology	81

The order of the subjects for sem1 and sem2 marks in arun_df are 'maths', 'physics', 'chemistry', 'biology' where as for sem3 in ad_additional is 'physics', 'chemistry', 'maths', 'biology' (different from arun_df). This data when concatenated gives:

```
In [17]: df = pd.concat([arun_df, df_additional], axis=1)
         df
```

Out[17]:

	subjects	sem1	sem2	subjects	sem3
0	maths	60	63	physics	50
1	physics	70	71	chemistry	64
2	chemistry	80	85	maths	88
3	biology	90	89	biology	81

We can see that how the concatenation have just added the DataFrames as it is without maintaining the subject column common.
We can handle this issue just by providing the correct index to the DataFrames. Indexes will be generally 0, 1, …. So if we know the data in df_additional then we can provide a custom index to it so that that subjects are correctly arranged

df_additional = pd.DataFrame(arun_sem3_scores, index=[1,2,0,3])

0 -> maths
1 -> physics

2 -> chemistry
3 -> biology

```
In [18]: arun_sem3_scores = {
             "subjects": ['physics', 'chemistry', 'maths', 'biology'],
             "sem3":[50, 64, 88, 81],
         }
         df_additional = pd.DataFrame(arun_sem3_scores, index=[1,2,0,3])
         df_additional
```

Out[18]:

	subjects	sem3
1	physics	50
2	chemistry	64
0	maths	88
3	biology	81

```
In [19]: df = pd.concat([arun_df, df_additional], axis=1)
         df
```

Out[19]:

	subjects	sem1	sem2	subjects	sem3
0	maths	60	63	maths	88
1	physics	70	71	physics	50
2	chemistry	80	85	chemistry	64
3	biology	90	89	biology	81

We can see that adding the custom index (that matches arun_df) we can resolve our issue.
This can be better resolved by ‘merge’ which we’ll see in next chapter.

9.8 Join DataFrame and series

Pandas allows us to join DataFrame with series as well.

```
In [20]: s = pd.Series([88, 76, 74, 72], name="sem4")
         s
```

```
Out[20]: 0    88
         1    76
         2    74
         3    72
         Name: sem4, dtype: int64
```

```
In [21]: df = pd.concat([arun_df, s], axis=1)
         df
```

Out[21]:

	subjects	sem1	sem2	sem4
0	maths	60	63	88
1	physics	70	71	76
2	chemistry	80	85	74
3	biology	90	89	72

We can see that the series s with sem4 marks has been joined to DataFrame (arun_df) with marks of sem1 and sem2.

9.9 Concatenating multiple DataFrames /series

We can concatenate multiple DataFrames and series also.

df = pd.concat([arun_df, df_additional['sem3'], s], axis=1)

where arun_df and ad_additional are DataFrame and s is a series.

```
In [22]: df = pd.concat([arun_df, df_additional['sem3'], s], axis=1)
         df
```

Out[22]:

	subjects	sem1	sem2	sem3	sem4
0	maths	60	63	88	88
1	physics	70	71	50	76
2	chemistry	80	85	64	74
3	biology	90	89	81	72

We have concatenated complete arun_df, a part of df_additional (i.e. only

sem3 column) and a series (sem4).

In next chapter, we’ll learn the merging operation on two dataframes.

10 Merge

As discussed in previous chapter, we can get data from various sources and thus various DataFrames. Merge is a very important and efficient way of merging two (or more) DataFrames.

Let’s understand with some examples.

```
arun_sem1_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'biology'],
    "sem1":[60, 70, 80, 90],
}
sem1_df = pd.DataFrame(arun_sem1_scores)

arun_sem2_scores = {
    "subjects": ['physics', 'chemistry', 'maths', 'biology'],
    "sem2":[73, 81, 88, 83],
}
sem2_df = pd.DataFrame(arun_sem2_scores)
```

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
arun_sem1_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'biology'],
    "sem1":[60, 70, 80, 90],
}
sem1_df = pd.DataFrame(arun_sem1_scores)
sem1_df
```

Out[2]:

	subjects	sem1
0	maths	60
1	physics	70
2	chemistry	80
3	biology	90

```
In [3]: arun_sem2_scores = {
            "subjects": ['physics', 'chemistry', 'maths', 'biology'],
            "sem2":[73, 81, 88, 83],
        }
        sem2_df = pd.DataFrame(arun_sem2_scores)
        sem2_df
```

Out[3]:

	subjects	sem2
0	physics	73
1	chemistry	81
2	maths	88
3	biology	83

10.1 Merging DataFrames

We have taken two DataFrames (‘sem1_df’ & ‘sem2_df’). The most important part of merge over concat is that, the merge() combines the DataFrames on the basis of values of common columns whereas concat() just appends the DataFrames.

```
In [4]: df = pd.merge(sem1_df, sem2_df, on='subjects')
        df
```

Out[4]:

	subjects	sem1	sem2
0	maths	60	88
1	physics	70	73
2	chemistry	80	81
3	biology	90	83

‘on’ works as the reference on which the DataFrames are going to be merged. This is similar to that of ‘join’ clause in sql queries.

10.2 Merging different values of “ON” (joining) column

We. Have seen that we can easily join the DataFrames with same subjects (column). Let’s try to merge DataFrames with different value of subject column.

arun_sem1_scores = {

"subjects": ['maths', 'physics', 'chemistry', 'litrature'],
"sem1":[60, 70, 80, 55],
}
sem1_df = pd.DataFrame(arun_sem1_scores)

arun_sem2_scores = {
"subjects": ['physics', 'chemistry', 'maths', 'biology', 'computers'],
"sem2":[73, 81, 88, 83, 88],
}
sem2_df = pd.DataFrame(arun_sem2_scores)

In [5]:
```
arun_sem1_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'litrature'],
    "sem1":[60, 70, 80, 55],
}
sem1_df = pd.DataFrame(arun_sem1_scores)
sem1_df
```

Out[5]:

	subjects	sem1
0	maths	60
1	physics	70
2	chemistry	80
3	litrature	55

In [6]:
```
arun_sem2_scores = {
    "subjects": ['physics', 'chemistry', 'maths', 'biology', 'computers'
    "sem2":[73, 81, 88, 83, 88],
}
sem2_df = pd.DataFrame(arun_sem2_scores)
sem2_df
```

Out[6]:

	subjects	sem2
0	physics	73
1	chemistry	81
2	maths	88
3	biology	83
4	computers	88

Merging the above DataFrames together.

```
In [7]: df = pd.merge(sem1_df, sem2_df, on='subjects')
        df
```

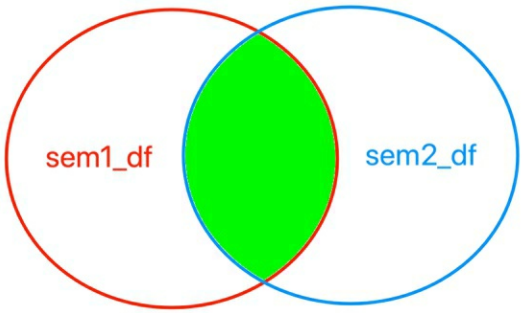
Out[7]:

	subjects	sem1	sem2
0	maths	60	88
1	physics	70	73
2	chemistry	80	81

Here we can see that the data is diminished i.e. some of the columns are lose and we can only see columns which are common (maths, physics and chemistry) in both the DataFrames.

This is because the merge operation by default merges the DataFrames with inner join. If we need to provide any other types of merge then we have to pass a parameter *'how'* as explained below.

10.2.1 Merging with Inner join

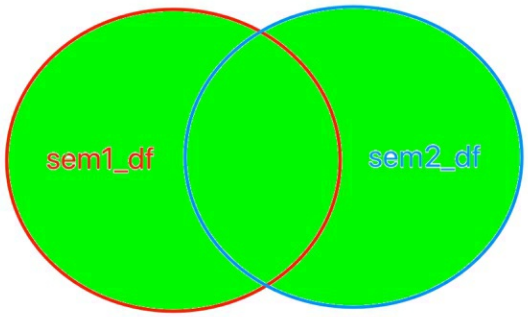


This is how an inner join looks like. This provides the result common to both the DataFrames.

Let's check out some other types of join.

10.2.2 Merging with outer join

The full outer join takes all the data from both the DataFrames.



df = pd.merge(sem1_df, sem2_df, on='subjects', how='outer')

how = “outer” makes the change

```
In [8]: # can see all the columns in both the df's

        df = pd.merge(sem1_df, sem2_df, on='subjects', how='outer')
        df
```

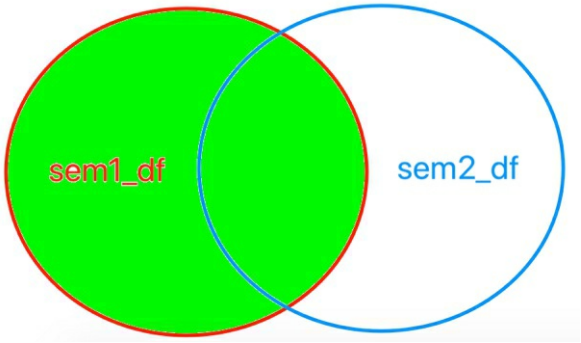
Out[8]:

	subjects	sem1	sem2
0	maths	60.0	88.0
1	physics	70.0	73.0
2	chemistry	80.0	81.0
3	litrature	55.0	NaN
4	biology	NaN	83.0
5	computers	NaN	88.0

Here we can see that we are getting all the rows (subjects) and if any of the other columns (sem1 or sem2) does not have corresponding values, shows NaN.

10.2.3 Merging with left join

A left join takes all the value from the left DataFrame (even the common ones).



df = pd.merge(sem1_df, sem2_df, on='subjects', how='left')

how = “left” makes the change

the left or right is decide as per the DataFrame passed to the merge operation. The 1st one (sem1_df here) becomes left and the 2nd one (sem2_df here) becomes right.

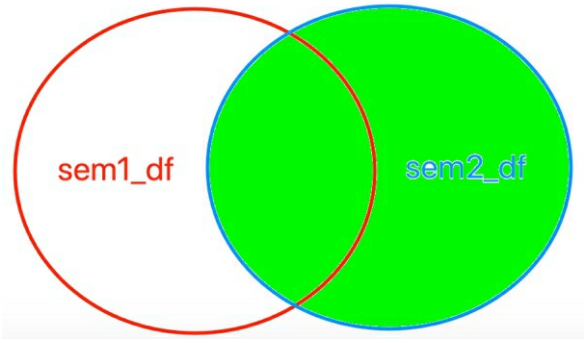
```
In [9]: # all subjects (rows) from  sem1

        df = pd.merge(sem1_df, sem2_df, on='subjects', how='left')
        df
```

Out[9]:

	subjects	sem1	sem2
0	maths	60	88.0
1	physics	70	73.0
2	chemistry	80	81.0
3	litrature	55	NaN

10.2.4 Merging with right join

By now it would be clear that what a right join would be doing. It takes all the rows (subjects) from the right DataFrame (sem2_df).



df = pd.merge(sem1_df, sem2_df, on='subjects', how='right')

how = “right” makes the change

```
In [10]: # all subjects (rows) from  sem2

         df = pd.merge(sem1_df, sem2_df, on='subjects', how='right')
         df
```

Out[10]:

	subjects	sem1	sem2
0	maths	60.0	88
1	physics	70.0	73
2	chemistry	80.0	81
3	biology	NaN	83
4	computers	NaN	88

10.3 Knowing the source DataFrame after merge

It’s sometime important to know the DataFrame or the source from where the merge has been done. By observation also we can tell this as both the values will be present in sem1 and sem2 columns if the subject exists in both the source DataFrames. If the sem1 column has value and sem2 is NaN then the source of the data (row) is sem1_df and if sem2 value is present and sem1 is NaN then the source DataFrame is sem2_df.
This can be ease by *‘indicator’*. This is a parameter which is by default false but if made true we can get the source DataFrames.

df = pd.merge(sem1_df, sem2_df, on='subjects', how='outer', indicator=True)

```
In [11]: # indicator tells the source df

         df = pd.merge(sem1_df, sem2_df, on='subjects', how='outer', indicator=True)
         df
```

Out[11]:

	subjects	sem1	sem2	_merge
0	maths	60.0	88.0	both
1	physics	70.0	73.0	both
2	chemistry	80.0	81.0	both
3	litrature	55.0	NaN	left_only
4	biology	NaN	83.0	right_only
5	computers	NaN	88.0	right_only

10.4 Merging DataFrames with same column names

We can come across some situations where the different DataFrames have same column names. Let's look at what happens in such situations.

Let's take a DataFrame which has same column as sem1_df

```
neha_sem1_scores = {
    "subjects": ['maths', 'physics', 'chemistry', 'computers'],
    "sem1":[65, 75, 83, 80],
}
neha_sem1_df = pd.DataFrame(neha_sem1_scores)
```

```
In [12]: neha_sem1_scores = {
             "subjects": ['maths', 'physics', 'chemistry', 'computers'],
             "sem1":[65, 75, 83, 80],
         }
         neha_sem1_df = pd.DataFrame(neha_sem1_scores)
         neha_sem1_df
```

Out[12]:

	subjects	sem1
0	maths	65
1	physics	75
2	chemistry	83
3	computers	80

Let's try merging this data (neha_sem1_df) with sem1_df.

```
In [13]: # same columns will automatically have _x and _y

         df = pd.merge(sem1_df, neha_sem1_df, on='subjects', how='outer')
         df
```

Out[13]:

	subjects	sem1_x	sem1_y
0	maths	60.0	65.0
1	physics	70.0	75.0
2	chemistry	80.0	83.0
3	litrature	55.0	NaN
4	computers	NaN	80.0

What we can observe is that though the column names were same, merge operation has added suffixes to the column names (_x & _y) by default.

This is great as we can now easily do operations on column. But what if in case we need our own custom suffixes instead of _x and _y.

We can do this by a parameter called *'suffixes'*. With this parameter we can pass a tuple of our own custom suffixes.

df = pd.merge(sem1_df, neha_sem1_df, on='subjects', how='outer', suffixes= ('_arun', '_neha'))

```
In [14]: # to avoid the _x & _y suffix we use suffixes

         df = pd.merge(sem1_df, neha_sem1_df, on='subjects', how='outer', suffixes=('_arun', '_neha'))
         df
```

Out[14]:

	subjects	sem1_arun	sem1_neha
0	maths	60.0	65.0
1	physics	70.0	75.0
2	chemistry	80.0	83.0
3	litrature	55.0	NaN
4	computers	NaN	80.0

After applying the *'suffixes'* parameter, the column names have been customised.

10.5 Other ways of joining

There are few other types of ways to join two (or more) DataFrames.

```
In [15]: sem1_df = sem1_df.set_index('subjects')
         sem1_df
```

Out[15]:

subjects	sem1
maths	60
physics	70
chemistry	80
litrature	55

```
In [16]: sem2_df = sem2_df.set_index('subjects')
         sem2_df
```

Out[16]:

subjects	sem2
physics	73
chemistry	81
maths	88
biology	83
computers	88

10.5.1 Join

It's a convenient way of joining two DataFrames which are differently-indexed.

sem1_df.join(sem2_df)

```
In [17]: sem1_df.join(sem2_df)
```

Out[17]:

subjects	sem1	sem2
maths	60	88.0
physics	70	73.0
chemistry	80	81.0
litrature	55	NaN

This by default used inner join. We can even use ‘how’ on joins to specify how we need to join the DataFrames as in merge as:

sem1_df.join(sem2_df, how='outer')
sem1_df.join(sem2_df, how='left')
sem1_df.join(sem2_df, how='right')

```
In [18]: sem1_df.join(sem2_df, how='outer')
         # sem1_df.join(sem2_df, how='left')
         # sem1_df.join(sem2_df, how='right')
```

Out[18]:

subjects	sem1	sem2
biology	NaN	83.0
chemistry	80.0	81.0
computers	NaN	88.0
litrature	55.0	NaN
maths	60.0	88.0
physics	70.0	73.0

10.5.2 Append

Append is a shortcut to concat(). These have predated concat().

sem1_df.append(sem2_df)

it appends two (or More) DataFrames. These concat only along axix=0 i.e. only rows.

```
In [19]: sem1_df.append(sem2_df)
```

Out[19]:

subjects	sem1	sem2
maths	60.0	NaN
physics	70.0	NaN
chemistry	80.0	NaN
litrature	55.0	NaN
physics	NaN	73.0
chemistry	NaN	81.0
maths	NaN	88.0
biology	NaN	83.0
computers	NaN	88.0

This appends one dataframe on other.

Let’s learn about pivoting operation in the next chapter.

11 Pivot

Pivot is used to reshape a given DataFrame organised by given index/column values. It uses unique values from given index/columns to create the axis of the resulting DataFrame. Pivot does not support data aggregation.

Let’s look into it in detail using an example.

Let’s take the weather report of various status (Mumbai, Delhi and Kolkata) for 3 different dates.

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_csv('weather_pivot.csv')
        df
```

Out[2]:

	date	city	temperature	windspeed
0	01/03/20	mumbai	32	9
1	02/03/20	mumbai	35	8
2	03/03/20	mumbai	33	6
3	01/03/20	delhi	40	7
4	02/03/20	delhi	38	9
5	03/03/20	delhi	37	8
6	01/03/20	kolkata	35	9
7	02/03/20	kolkata	36	6
8	03/03/20	kolkata	35	7

This is a good practice to always convert dates in the DataFrame to timestamp as generally it will be in string format.

```
In [3]: type(df['date'][0])
Out[3]: str
```

Conversion to timestamp

```
In [4]: # chande data to datetime format
        df['date']=pd.to_datetime(df['date'])
        df
```

Out[4]:

	date	city	temperature	windspeed
0	2020-01-03	mumbai	32	9
1	2020-02-03	mumbai	35	8
2	2020-03-03	mumbai	33	6
3	2020-01-03	delhi	40	7
4	2020-02-03	delhi	38	9
5	2020-03-03	delhi	37	8
6	2020-01-03	kolkata	35	9
7	2020-02-03	kolkata	36	6
8	2020-03-03	kolkata	35	7

```
In [5]: type(df['date'][0])
Out[5]: pandas._libs.tslibs.timestamps.Timestamp
```

We can observe that the time before conversion was string format but after conversion becomes timestamp.

11.1 Multilevel columns

Pivot creates multilevel columns. Let’s take date as index and city as column from above DataFrame.

df.pivot(index="date",columns="city")

```
In [6]: # multilevel columns

        df.pivot(index="date",columns="city")
```

Out[6]:

	temperature			windspeed		
city	delhi	kolkata	mumbai	delhi	kolkata	mumbai
date						
2020-01-03	40	35	32	7	9	9
2020-02-03	38	36	35	9	6	8
2020-03-03	37	35	33	8	7	6

We can see that temperature and windspeed has become one level of column whereas city became another.

11.2 Data for selected value (column)

We can obtain the specific value (column) from the above DataFrame too where we can select the specific column i.e. temperature or windspeed.

df.pivot(index="date",columns="city", values="windspeed")

```
In [7]: # to select specific value (column)

        df.pivot(index="date",columns="city", values="windspeed")
```

Out[7]:

city	delhi	kolkata	mumbai
date			
2020-01-03	7	9	9
2020-02-03	9	6	8
2020-03-03	8	7	6

Here we have obtained the subset of the original DataFrame i.e. data for only windspeed. There is an alternate way to do the same.

df.pivot(index="date",columns="city")["windspeed"]

In [8]:
```
# alternatively,

df.pivot(index="date",columns="city")["windspeed"]
```

Out[8]:

city	delhi	kolkata	mumbai
date			
2020-01-03	7	9	9
2020-02-03	9	6	8
2020-03-03	8	7	6

11.3 Error from duplicate values

This is what a pivot function does in general but we have to take care of the duplicate values.

For example:

```
df = pd.DataFrame({
    "first":list("aabbcc"),
    "second":list("xxyyzz"),
    "third":[1,2,3,4,5,6]
})
```

In [9]:
```
df = pd.DataFrame({
    "first":list("aabbcc"),
    "second":list("xxyyzz"),
    "third":[1,2,3,4,5,6]
})
df
```

Out[9]:

	first	second	third
0	a	x	1
1	a	x	2
2	b	y	3
3	b	y	4
4	c	z	5
5	c	z	6

We can see, first two rows are the same for our index and columns arguments.

If we apply pivot on this data we get and error as:
Index contains duplicate entries, cannot reshape

```
In [10]: # A ValueError is raised if there are any duplicates.

         df.pivot(index="first",columns="second")

         # ERROR: Index contains duplicate entries, cannot reshape
```

```
---------------------------------------------------------------------------
ValueError                                Traceback (most recent call last)
<ipython-input-10-79c796bc49f1> in <module>
      1 # A ValueError is raised if there are any duplicates.
      2
----> 3 df.pivot(index="first",columns="second")
      4
      5 # ERROR: Index contains duplicate entries, cannot reshape
```

This is all about pivot . let’s look at pivot table in the next chapter.

12 Pivot table

We generally work on pivot table provided in Excel, which is very handy to get a detailed summary of the data and analyse the data. Pandas provide the same to us with a function called *'pivot_table()'*.

Let's look at it in detail by some examples.

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.read_csv("weather_pivotTable.csv")
df['date']=pd.to_datetime(df['date'])
df
```

Out[2]:

	date	city	temperature	windspeed	time
0	2020-01-03	mumbai	43	9	morning
1	2020-01-03	mumbai	42	11	evening
2	2020-01-03	delhi	40	8	morning
3	2020-01-03	delhi	42	8	evening
4	2020-01-03	kolkata	38	6	morning
5	2020-01-03	kolkata	37	8	evening
6	2019-01-12	mumbai	22	12	morning
7	2019-01-12	mumbai	20	10	evening
8	2019-01-12	delhi	18	9	morning
9	2019-01-12	delhi	19	7	evening
10	2019-01-12	kolkata	21	7	morning
11	2019-01-12	kolkata	23	10	evening

The above example contains morning and evening weather details of some cities (Mumbai, Delhi and Kolkata) for 2 different dates.
Let's apply the pivot table on the above data.

df.pivot_table(index="city", columns="date")

```
In [3]: # pivot table for the data
        # by default takes mean on the values

        df.pivot_table(index="city", columns="date")
```

Out[3]:

	temperature		windspeed	
date	2019-01-12	2020-01-03	2019-01-12	2020-01-03
city				
delhi	18.5	41.0	8.0	8.0
kolkata	22.0	37.5	8.5	7.0
mumbai	21.0	42.5	11.0	10.0

We have applied pivot table for the above data with city as index and dates as columns.
The output provides us the average temperature and windspeed of different cities on various dates.
We can do the same in different order also i.e. date as index and cities as columns.

```
In [4]: df.pivot_table(index="date", columns="city")
```

Out[4]:

	temperature			windspeed		
city	delhi	kolkata	mumbai	delhi	kolkata	mumbai
date						
2019-01-12	18.5	22.0	21.0	8.0	8.5	11.0
2020-01-03	41.0	37.5	42.5	8.0	7.0	10.0

12.1 Aggregate function

By default the above result provides us the mean of the temperature and windspeed on various dates. We can also pass any custom function of any other function to the above data so that we can get the result accordingly. This is done by a parameter *'aggfunc'*.

df.pivot_table(index="city", columns="date", aggfunc="sum")

```
In [5]: df.pivot_table(index="city", columns="date", aggfunc="sum")
```

Out[5]:

	temperature		windspeed	
date	2019-01-12	2020-01-03	2019-01-12	2020-01-03
city				
delhi	37	82	16	16
kolkata	44	75	17	14
mumbai	42	85	22	20

We can see that we have passed the *'aggfunc'* parameter to provide us the sum of the results.

We can also know the count. i.e. the number of dates available for that city.

```
In [6]: df.pivot_table(index="city", columns="date", aggfunc="count")
```

Out[6]:

	temperature		time		windspeed	
date	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03
city						
delhi	2	2	2	2	2	2
kolkata	2	2	2	2	2	2
mumbai	2	2	2	2	2	2

12.1.1 List of function to aggfunc

We can pass a list of function to *'aggfunc'* parameter too as:

df.pivot_table(index="city", columns="date", aggfunc=[min, max, sum])

```
In [7]: df.pivot_table(index="city", columns="date", aggfunc=[min, max, sum])
```

Out[7]:

	min						max						sum			
	temperature		time		windspeed		temperature		time		windspeed		temperature		windspeed	
date	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03
city																
delhi	18	40	evening	evening	7	8	19	42	morning	morning	9	8	37	82	16	16
kolkata	21	37	evening	evening	7	6	23	38	morning	morning	10	8	44	75	17	14
mumbai	20	42	evening	evening	10	9	22	43	morning	morning	12	11	42	85	22	20

We can see that the result provides us the min, max and sum of all the

available columns.

12.1.2 Custom functions to individual columns

We can provide separate functions to individual columns too. This is possible by passing a dictionary with keys as column name and values as the function to be applied.

df.pivot_table(index="city", columns="date", aggfunc={
"temperature":[min, max, "mean"],
"windspeed":"sum"
})

```
In [8]: df.pivot_table(index="city", columns="date", aggfunc={
            "temperature":[min, max, "mean"],
            "windspeed":"sum"
        })
```

Out[8]:

	temperature						windspeed	
	max		mean		min		sum	
date	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03	2019-01-12	2020-01-03
city								
delhi	19.0	42.0	18.5	41.0	18.0	40.0	16	16
kolkata	23.0	38.0	22.0	37.5	21.0	37.0	17	14
mumbai	22.0	43.0	21.0	42.5	20.0	42.0	22	20

12.2 Apply pivot_table() on desired columns

We can apply the pivot_table for our own desired columns too. This is done by passing a parameter *'value'*. The column name provided to the parameter will be the only one to which pivot_table applies and return a DataFrame for inly this column.

df.pivot_table(index="city", columns="date", aggfunc="sum", values="windspeed")

```
In [9]: df.pivot_table(index="city", columns="date", aggfunc="sum", values="windspeed")
```

Out[9]:

date	2019-01-12	2020-01-03
city		
delhi	16	16
kolkata	17	14
mumbai	22	20

We can do this alternatively, in which we pull out the desired column after the pivot table is applied.

df.pivot_table(index="city", columns="date", aggfunc="sum")["windspeed"]

```
In [10]: df.pivot_table(index="city", columns="date", aggfunc="sum")["windspeed"]
```

Out[10]:

date	2019-01-12	2020-01-03
city		
delhi	16	16
kolkata	17	14
mumbai	22	20

We can see in the above example that how we have aggregated the desired column.

12.3 Margins

Sometimes we need to know the mean of the row. In that case we can directly use the parameter *'margins'* provided by pivot_table().

df.pivot_table(index="city", columns="date", values="temperature", margins=True)

```
In [11]: # average of values on two different dates i.e. average of each row (named as "ALL" by default)
         df.pivot_table(index="city", columns="date", values="temperature", margins=True)
```

Out[11]:

date	2019-01-12 00:00:00	2020-01-03 00:00:00	All
city			
delhi	18.5	41.000000	29.750000
kolkata	22.0	37.500000	29.750000
mumbai	21.0	42.500000	31.750000
All	20.5	40.333333	30.416667

We can see that a new column has been added (All) that gives the average value of each row.

12.3.1 Naming the margin column

By default the margin column comes with name ‘All’. We can modify the margin column name with a custom name by using a parameter called *‘margins_name’*.

df.pivot_table(index="city", columns="date", values="temperature", margins=True, margins_name="average")

```
In [12]: df.pivot_table(index="city",
                        columns="date",
                        values="temperature",
                        margins=True,
                        margins_name="average")
```

Out[12]:

date	2019-01-12 00:00:00	2020-01-03 00:00:00	average
city			
delhi	18.5	41.000000	29.750000
kolkata	22.0	37.500000	29.750000
mumbai	21.0	42.500000	31.750000
average	20.5	40.333333	30.416667

12.4 Grouper

Let’s say we need to know the average temperature in a particular month or an year, we can have the pivot table grouped as per the month or the year etc by using grouper.

Let’s look at an example.

df.pivot_table(index=pd.Grouper(freq='Y', key="date"), columns="city")

here we are applying grouper to the index with frequency as ‘year’ and on column ‘date’. This will provide us the average temperature and windspeed for each year. Similarly we can have the frequency as ‘M’ for average temperature and windspeed for each month.

```
In [13]: df.pivot_table(index=pd.Grouper(freq='Y', key="date"), columns="city")
```

Out[13]:

	temperature			windspeed		
city	delhi	kolkata	mumbai	delhi	kolkata	mumbai
date						
2019-12-31	18.5	22.0	21.0	8.0	8.5	11.0
2020-12-31	41.0	37.5	42.5	8.0	7.0	10.0

The most important point to note here is that the date should be formatted to timestamp format otherwise we will get error as the *'freq'* parameter can only understand the timestamp format.

```
In [14]: # the frequency parameter will only work, if we have the
         # date as a timestamp
         # df['date']=pd.to_datetime(df['date'])

         type(df['date'][0])
```

```
Out[14]: pandas._libs.tslibs.timestamps.Timestamp
```

12.5 Filling the missing value in pivot table

We can sometimes encounter NaN after using pivot table in which case we may need to fill the NaN cells with some value. This can be done in pivot table by *'fill_value'* parameter.

```
df = pd.DataFrame({
    "first":list("aaabbbccd"),
    "second":list("xyzxyzxyy"),
    "third":[1,2,3,4,5,6,7,8,9]
})
```

```
In [15]: df = pd.DataFrame({
             "first":list("aaabbbccd"),
             "second":list("xyzxyzxzz"),
             "third":[1,2,3,4,5,6,7,8,9]
         })
         df
```

Out[15]:

	first	second	third
0	a	x	1
1	a	y	2
2	a	z	3
3	b	x	4
4	b	y	5
5	b	z	6
6	c	x	7
7	c	z	8
8	d	z	9

df.pivot_table(index="first", columns="second")

```
In [16]: df.pivot_table(index="first", columns="second")
```

Out[16]:

	third		
second	**x**	**y**	**z**
first			
a	1.0	2.0	3.0
b	4.0	5.0	6.0
c	7.0	NaN	8.0
d	NaN	NaN	9.0

After applying the pivot table on the data we have some NaN (missing values). These missing values may sometime need to be filled with some other value.

df.pivot_table(index="first", columns="second", fill_value="NIL")

```
In [17]: # fill_value could be used as parameter to avoid missing values

         df.pivot_table(index="first", columns="second", fill_value="NIL")
```

Out[17]:

	third		
second	x	y	z
first			
a	1	2	3.0
b	4	5	6.0
c	7	NIL	8.0
d	NIL	NIL	9.0

The *'fill_value'* parameter takes the value 'NIL' and fills the missing values with it.

with this we have completed pivot_table. Let's look at reshaping the DataFrame in next chapter.

13 Reshape DataFrame using melt

In real life problems we may sometime need data which is in column to be in rows. This in Pandas is possible by using “melt”. Melt unpivots the DataFrame from wide format (columns) to long format (rows).

Let’s understand this with an example.

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_csv('subject_melt.csv')
        df
```

Out[2]:

	subjects	arun	varun	neha
0	maths	72	88	87
1	physics	92	74	81
2	chemistry	55	69	78
3	biology	82	77	89
4	computers	68	71	76

We have taken a data which has various subjects and their marks for respective students. The students are in various columns. Using melt we can transform the columns of various students into rows.

df1 = pd.melt(df, id_vars=["subjects"])

```
In [3]: df1 = pd.melt(df, id_vars=["subjects"])
        df1
```

Out[3]:

	subjects	variable	value
0	maths	arun	72
1	physics	arun	92
2	chemistry	arun	55
3	biology	arun	82
4	computers	arun	68
5	maths	varun	88
6	physics	varun	74
7	chemistry	varun	69
8	biology	varun	77
9	computers	varun	71
10	maths	neha	87
11	physics	neha	81
12	chemistry	neha	78
13	biology	neha	89
14	computers	neha	76

With the above command we can see that the various columns of students have become rows with a new column name variable and the marks column have become value (by default).

The value (column name) passed to *'id_vars'* parameter keeps that row constant and melt down or reshape the other columns to fit into this.

13.1 Use of melt

With melt the column becomes row and now any value could be accessed as we access a row. For example.

df1[df1["subjects"]=="maths"]

```
In [4]: df1[df1["subjects"]=="maths"]
```

Out[4]:

	subjects	variable	value
0	maths	arun	72
5	maths	varun	88
10	maths	neha	87

13.2 Melt for only one column

Sometime we may need to convert a specific column to row. This could be done by passing that column to parameter *'value_vars'*. For example.

df1 = pd.melt(df, id_vars=["subjects"], value_vars=["arun"])

```
In [5]: df1 = pd.melt(df, id_vars=["subjects"], value_vars=["arun"])
        df1
```

Out[5]:

	subjects	variable	value
0	maths	arun	72
1	physics	arun	92
2	chemistry	arun	55
3	biology	arun	82
4	computers	arun	68

Now we can see that only column name "arun" has be converted to rows.

13.3 Melt multiple columns

We may even need multiple columns to be melted in rows. This can be done by passing a list of columns to parameter *'value_vars'*.

df1 = pd.melt(df, id_vars=["subjects"], value_vars=["arun", "neha"])

```
In [6]: df1 = pd.melt(df, id_vars=["subjects"], value_vars=["arun", "neha"])
        df1
```

Out[6]:

	subjects	variable	value
0	maths	arun	72
1	physics	arun	92
2	chemistry	arun	55
3	biology	arun	82
4	computers	arun	68
5	maths	neha	87
6	physics	neha	81
7	chemistry	neha	78
8	biology	neha	89
9	computers	neha	76

Here we can see that by passing the list of columns we can convert those column to rows.

13.4 Custom column name

The conversion of columns into rows is of great help sometimes but the new column name given by default may not be what we need. we have the flexibility to convert the default name to our own custom name too by the use of parameter *'var_name'* and *'value_name'*.

13.4.1 Custom variable name

The names of the students become variable (column) by default, which can be customised by the use of "var_name"

df1 = pd.melt(df, id_vars=["subjects"], var_name="name")

```
In [7]: df1 = pd.melt(df, id_vars=["subjects"], var_name="name")
        df1
```

Out[7]:

	subjects	name	value
0	maths	arun	72
1	physics	arun	92
2	chemistry	arun	55
3	biology	arun	82
4	computers	arun	68
5	maths	varun	88
6	physics	varun	74
7	chemistry	varun	69
8	biology	varun	77
9	computers	varun	71
10	maths	neha	87
11	physics	neha	81
12	chemistry	neha	78
13	biology	neha	89
14	computers	neha	76

13.4.2 Custom value name

The marks of the students becomes marks (column) by default, which can be customised by the use of parameter *'value_name'*.

df1 = pd.melt(df, id_vars=["subjects"], var_name="name", value_name="marks")

In [9]:
```
df1 = pd.melt(df, id_vars=["subjects"], var_name="name", value_name="marks")
df1
```

Out[9]:

	subjects	name	marks
0	maths	arun	72
1	physics	arun	92
2	chemistry	arun	55
3	biology	arun	82
4	computers	arun	68
5	maths	varun	88
6	physics	varun	74
7	chemistry	varun	69
8	biology	varun	77
9	computers	varun	71
10	maths	neha	87
11	physics	neha	81
12	chemistry	neha	78
13	biology	neha	89
14	computers	neha	76

In the next chapter we'll be looking into another helpful way of reshaping the DataFrame using stack and unstack.

14 Reshaping using stack and unstack

Many a time we get data with multilevel headers (columns). This data is little difficult to analyse a it is. We may have to break the multilevel columns into some reasonable format. Stack reshapes the DataFrame to have one or more levels of index.

Let's understand this with an example.

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_excel("students_stack.xlsx", header=[0,1,2], index_col=0)
        df
```

Out[2]:

	name					
	arun			varun		
	maths	physics	chemistry	maths	physics	chemistry
sem1	60	63	62	58	66	65
sem2	58	61	60	56	64	63
sem3	62	65	64	60	68	67
sem4	67	70	69	65	73	72

The data we have taken has three level of columns. The level are indexed from zero. The 0^{th} level (header 0) is name. The 1^{st} level (header 1) contains the name of the students (arun and varun) and the 2^{nd} level (header 2) has the name of the subjects (maths, physics and chemistry). The semesters are the set as the index for the DataFrame.

14.1 Stack the DataFrame

Let's see what happens when we stack them.

df.stack()

```
In [3]: # by default stacks the last level of header

        df.stack()
```

Out[3]:

		name	
		arun	varun
sem1	chemistry	62	65
	maths	60	58
	physics	63	66
sem2	chemistry	60	63
	maths	58	56
	physics	61	64
sem3	chemistry	64	67
	maths	62	60
	physics	65	68
sem4	chemistry	69	72
	maths	67	65
	physics	70	73

Once running the stack on the DataFrame we can see that the subjects column (level 2) have become the row and the data for other columns are aligned accordingly.

14.2 Stack custom level of column

We can see, in the above output that the last level of the column has been taken for stack by default. We can customise to get our choice of column by using parameter *'level'*.

df_stacked = df.stack(level=1)

```
In [4]: df_stacked = df.stack(level=1)
        df_stacked
```

Out[4]:

	name	chemistry	maths	physics
sem1	arun	62	60	63
	varun	65	58	66
sem2	arun	60	58	61
	varun	63	56	64
sem3	arun	64	62	65
	varun	67	60	68
sem4	arun	69	67	70
	varun	72	65	73

Here we ran the stack for level 1 i.e. on name of the students and the students name became the row and the data has been adjusted accordingly.

The same can be done as:
df_stacked = df.stack(1)

```
In [5]: df_stacked = df.stack(1)
        df_stacked
```

Out[5]:

	name	chemistry	maths	physics
sem1	arun	62	60	63
	varun	65	58	66
sem2	arun	60	58	61
	varun	63	56	64
sem3	arun	64	62	65
	varun	67	60	68
sem4	arun	69	67	70
	varun	72	65	73

14.3 Stack on multiple levels of column

We may sometime need that data where we may have to convert multiple level of columns into rows. This can be done by passing a list of level of columns to *'level'* parameter.

df_stacked = df.stack(level=[1,2])

OR

df_stacked = df.stack([1,2])

```
In [6]: df_stacked = df.stack(level=[1,2])
        df_stacked
```

Out[6]:

			name
sem1	arun	chemistry	62
		maths	60
		physics	63
	varun	chemistry	65
		maths	58
		physics	66
sem2	arun	chemistry	60
		maths	58
		physics	61
	varun	chemistry	63
		maths	56
		physics	64
sem3	arun	chemistry	64
		maths	62
		physics	65
	varun	chemistry	67
		maths	60
		physics	68
sem4	arun	chemistry	69
		maths	67
		physics	70
	varun	chemistry	72
		maths	65
		physics	73

14.4 Dropping missing values

There is always a chance of having some missing values in the data and we must be able to control that. Stack by default removes the row with all

missing values but still we can control the behaviour by parameter called *'dropna'*.

Let's take an example first

columns = pd.MultiIndex.from_tuples([('weight', 'kilogram'), ('height', 'meter')])

animals_df = pd.DataFrame([[1.3, None], [8, 2.8]], index=['rat', 'dog'], columns=columns)

```
In [7]: columns = pd.MultiIndex.from_tuples([('weight', 'kilogram'),
                                             ('height', 'meter')])
        animals_df = pd.DataFrame([[1.3, None], [8, 2.8]],
                                  index=['rat', 'dog'],
                                  columns=columns
                                 )
        animals_df
```

Out[7]:

	weight	height
	kilogram	meter
rat	1.3	NaN
dog	8.0	2.8

The data contains the name of the animals as index and 2 levels of columns.

Let's stack the data and see the result.

```
In [8]: animals_df.stack()
```

Out[8]:

		height	weight
rat	**kilogram**	NaN	1.3
dog	**kilogram**	NaN	8.0
	meter	2.8	NaN

This is the default stack where dropna will be true (*dropna*=True).
Let's look at the result making the parameter false.

```
In [9]: animals_df.stack(dropna=False)
```

Out[9]:

		height	weight
rat	kilogram	NaN	1.3
	meter	NaN	NaN
dog	kilogram	NaN	8.0
	meter	2.8	NaN

We can see that one extra row has been found with all the values as NaN. This row was removed by stack by default. This is how we can control the missing values too.

14.5 Unstack the stacked DataFrame

This is a reverse process of the stack. Here, we can convert the multilevel indexes to multilevel columns.

Let's look at this with some examples:

14.5.1 Default unstack

By default as stack takes the innermost level of column, unstack also takes the innermost layer of index.

df_stacked.unstack()

```
In [10]: df_stacked.unstack()
```

Out[10]:

		name		
		chemistry	maths	physics
sem1	arun	62	60	63
	varun	65	58	66
sem2	arun	60	58	61
	varun	63	56	64
sem3	arun	64	62	65
	varun	67	60	68
sem4	arun	69	67	70
	varun	72	65	73

Here we can see that the innermost index for the df_stacked (in above example) was, name of the subjects which is now the column of the DataFrame.

14.5.2 Converting other index levels to column

We can provide *'level'* as parameter to specify which level of index to be transformed to column.

For example:
df_stacked = df.stack(level=0)

OR

df_stacked = df.stack(0)

```
In [11]: df_stacked.unstack(level=0)
```

Out[11]:

		name			
		sem1	sem2	sem3	sem4
arun	chemistry	62	60	64	69
	maths	60	58	62	67
	physics	63	61	65	70
varun	chemistry	65	63	67	72
	maths	58	56	60	65
	physics	66	64	68	73

df_stacked = df.stack(level=1)

OR

df_stacked = df.stack(1)

```
In [12]: df_stacked.unstack(1)
```

Out[12]:

		name	
		arun	varun
sem1	chemistry	62	65
	maths	60	58
	physics	63	66
sem2	chemistry	60	63
	maths	58	56
	physics	61	64
sem3	chemistry	64	67
	maths	62	60
	physics	65	68
sem4	chemistry	69	72
	maths	67	65
	physics	70	73

14.5.3 Unstack multiple indexes

We can unstack multiple indexes also by providing a list to *'level'* parameter.

df_stacked.unstack(level=[1,2])

OR

df_stacked.unstack([1,2])

```
In [13]: df_stacked.unstack([1,2])
```

Out[13]:

	name					
	arun			varun		
	chemistry	maths	physics	chemistry	maths	physics
sem1	62	60	63	65	58	66
sem2	60	58	61	63	56	64
sem3	64	62	65	67	60	68
sem4	69	67	70	72	65	73

This is all about converting columns to rows and rows to columns by stack and unstack respectively and working with multilevel rows and columns. Let's look at the process of frequency distribution of the data in the next chapter.

15 Frequency distribution of DataFrame column

we may encounter some data where we may need the frequent of occurrence of some column values i.e. frequency distribution of the column value. This is very common in statistics and is known as contingency table or cross tabulation (crosstab). crosstab is heavily used in survey research, business intelligence, engineering and scientific research. It provides a basic picture of the interrelation between two variables and can help find interactions between them.

Pandas allows us to calculate the frequency distribution with the function named “crosstab”.
Let’s look at this with some examples.

We are taking an example data of some people from various countries (India, UK and USA) along with their age, gender and hair colour.

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv("haircolor.csv")
        df
```

Out[2]:

	name	country	gender	age	hair_color
0	Ram	India	M	23	black
1	Mathew	UK	M	27	brown
2	Gillian	UK	F	43	brown
3	Tom	USA	M	33	brown
4	Anna	USA	F	25	blonde
5	Sophia	USA	F	27	blonde
6	Emma	UK	F	52	blonde
7	Sweta	India	F	23	black
8	Mohan	India	M	44	black
9	Amelia	UK	F	24	blonde

15.1 Apply crosstab

Let’s apply crosstab on the above data and try to find the hair colour frequency distribution as per the countries.

pd.crosstab(df.country, df.hair_color)

The 1st parameter to crosstab function is the *'index'* and the 2nd parameter is the *'column'* on which we want to apply the frequency distribution.

```
In [3]: pd.crosstab(df.country, df.hair_color)
```

Out[3]:

country	black	blonde	brown
India	3	0	0
UK	0	2	2
USA	0	2	1

We can see that a distribution has been shown of hair colour as per the country i.e.

- From India, 3 person are having black coloured hair
- From UK, 2 persons are with brown and blonde hairs each
- From USA, 2 persons are with blonde hair and 1 with brown

Similar analysis can also be done using gender just by changing the *index* parameter to gender.

pd.crosstab(df.gender, df.hair_color)

```
In [4]: pd.crosstab(df.gender, df.hair_color)
```

Out[4]:

gender	black	blonde	brown
F	1	4	1
M	2	0	2

The important point to note here is that crosstab is a function of Pandas and the parameter passed are the DataFrame column (as df.gender etc.).

15.2 Get total of rows/columns

We can get the total sum of rows/columns values also. This may be very handy sometimes and can be done by a parameter names *'margins'*.

pd.crosstab(df.gender, df.hair_color, margins=True)

```
In [5]: pd.crosstab(df.gender, df.hair_color, margins=True)
```

Out[5]:

hair_color	black	blonde	brown	All
gender				
F	1	4	1	6
M	2	0	2	4
All	3	4	3	10

This is how we get the total of rows and columns.

15.3 Multilevel columns

We may sometime need to have the distribution with respect to other factors in column too. We can have multilevel column for this reason and this can be done by passing a list to the *column* parameter.

pd.crosstab(df.country, [df.gender, df.hair_color])

```
In [6]: pd.crosstab(df.country, [df.gender, df.hair_color])
```

Out[6]:

gender	F			M	
hair_color	black	blonde	brown	black	brown
country					
India	1	0	0	2	0
UK	0	2	1	0	1
USA	0	2	0	0	1

Just by passing a list of gender and hair_colour we get a multilevel column frequency distribution.

15.4 Multilevel indexes

Similar to multilevel column we can have multilevel indexes also and the process is also the same. We just have to pass a list to *index* parameter.

pd.crosstab([df.gender, df.country], df.hair_color)

```
In [7]: pd.crosstab([df.gender, df.country], df.hair_color)
```

Out[7]:

	hair_color	black	blonde	brown
gender	country			
F	India	1	0	0
	UK	0	2	1
	USA	0	2	0
M	India	2	0	0
	UK	0	0	1
	USA	0	0	1

15.5 Custom name to rows/columns

We have created multilevel indexes/columns but what if we need to give a custom name to those indexes and columns. We can do this using the parameter '*rownames*' and '*colnames*'.

pd.crosstab([df.gender, df.country], df.hair_color, rownames=["Gender", "Country"], colnames=["HairColor"])

```
In [8]: pd.crosstab([df.gender, df.country],
                   df.hair_color,
                   rownames=["Gender", "Country"],
                   colnames=["HairColor"])
```

Out[8]:

	HairColor	black	blonde	brown
Gender	Country			
F	India	1	0	0
	UK	0	2	1
	USA	0	2	0
M	India	2	0	0
	UK	0	0	1
	USA	0	0	1

We have provided two names to rownames as we have two indexes and 1 to colnames as we have only one column.

15.6 Normalize (percentage) of the frequency

What if we need to know how much percentage of females having blonde hairs and percentage of males having black hair.
This percentage of the distribution can be obtained using a *'normalize'* parameter.

pd.crosstab([df.gender], df.hair_color, normalize="index")

```
In [9]: pd.crosstab([df.gender], df.hair_color, normalize="index")
```

Out[9]:

hair_color	black	blonde	brown
gender			
F	0.166667	0.666667	0.166667
M	0.500000	0.000000	0.500000

The above command results the percentage distribution of the data.

Let's validate this.

```
In [10]: pd.crosstab([df.gender], df.hair_color, margins=True)
```

Out[10]:

hair_color	black	blonde	brown	All
gender				
F	1	4	1	6
M	2	0	2	4
All	3	4	3	10

Here we can see that the frequency distribution. Let's check the percentage

- Females with black hairs: 1/6 = 0.166667 = ~16%
- Females with blonde hairs: 4/6 = 0.6666 = ~66%
- Males with black hairs: 2/4 = 0.5 = 50%

15.7 Analysis using custom function

Till now we have been working with gender, hair colour and countries but what if we need to know or analyse the data in terms of age (or any other column available).

We can do this with the help of parameters '*values*' and '*aggfunc*'.

Let's take a scenario where we want to know the average age of females having black/blonde/brown hair. This can be done as:

pd.crosstab([df.gender], df.hair_color, values=df.age, aggfunc="mean")

```
In [11]: pd.crosstab([df.gender],
                     df.hair_color,
                     values=df.age,
                     aggfunc="mean")
```

Out[11]:

gender	black	blonde	brown
F	23.0	32.0	43.0
M	33.5	NaN	30.0

The *'value'* parameter takes the column that we need to analyse and *'aggfunc'* provides the function that we need to apply on the value column.
For example in our case we needed the average age of females having black/blonde/brown hair. Average has been done by '*aggfunc*' parameter and the age has been taken care by *'value'* parameter.

This is all about frequency distribution. Let's learn the ways to delete the unwanted rows/columns in the next chapter.

16 Drop unwanted rows/columns

Sometimes it's possible that we may not need all the data that's available to us. It may sometime cause confusion to have so many unwanted rows and columns in our analysis. We have an option to delete the rows/columns that we don't need.

Let's take an example of hair colours of people in various country to understand the various operations.

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_csv("haircolor.csv")
        df
```

Out[2]:

	name	country	gender	age	hair_color
0	Ram	India	M	23	black
1	Mathew	UK	M	27	brown
2	Gillian	UK	F	43	brown
3	Tom	USA	M	33	brown
4	Anna	USA	F	25	blonde
5	Sophia	USA	F	27	blonde
6	Emma	UK	F	52	blonde
7	Sweta	India	F	23	black
8	Mohan	India	M	44	black
9	Amelia	UK	F	24	blonde

We'll convert this data to multilevel index.

df_drop = pd.crosstab([df.gender, df.country], df.hair_color)

```
In [3]: df_drop = pd.crosstab([df.gender, df.country],
                              df.hair_color)
        df_drop
```

Out[3]:

	hair_color	black	blonde	brown
gender	country			
F	India	1	0	0
	UK	0	2	1
	USA	0	2	0
M	India	2	0	0
	UK	0	0	1
	USA	0	0	1

All the deletion process is done with *drop()*. This function can be used with various parameters to delete the desired rows/columns or both.

16.1 Delete row

The parameter *'axis'* decides the row/column. If the value to *'axis'* is 0 then operation is performed on rows. By default the *'axis'* Is set to zero (row) in *drop()*.

df_drop.drop('M')

```
In [4]: df_drop.drop('M')
```

Out[4]:

	hair_color	black	blonde	brown
gender	country			
F	India	1	0	0
	UK	0	2	1
	USA	0	2	0

Here, we have just provided the name of the row index that we need to delete Alternative to specifying axis (labels, axis=0 is equivalent to index=labels). i.e.

df_drop.drop(index = 'M')

```
In [5]: df_drop.drop(index = 'M')
```

Out[5]:

	hair_color	black	blonde	brown
gender	country			
F	India	1	0	0
	UK	0	2	1
	USA	0	2	0

Here, we have removed all the rows that corresponds to index ‘M’.

16.1.1 Delete rows of custom index level

By default the level that’s deleted is level=0, but if we need to delete a row with any other level of index, then we can pass the level number to the parameter *‘level’*.

df_drop.drop(index = 'India', level=1)

```
In [6]: df_drop.drop(index = 'India', level=1)
```

Out[6]:

	hair_color	black	blonde	brown
gender	country			
F	UK	0	2	1
	USA	0	2	0
M	UK	0	0	1
	USA	0	0	1

Here, we have deleted the row with index ‘India’ at level 1.

16.1.2 Delete multiple rows

In order to delete multiple rows we have to pass a list of index to the *drop()* function.

df_drop.drop(index = ['India', 'UK'], level=1)

```
In [7]: df_drop.drop(index = ['India', 'UK'], level=1)
```

Out[7]:

	hair_color	black	blonde	brown
gender	country			
F	USA	0	2	0
M	USA	0	0	1

Here, we have deleted rows with ‘India’ and ‘UK’ at level 1.

16.2 Drop column

As we have deleted the row, we can delete the columns too. To delete the columns we have to pass the value (column label) to parameter *‘columns’*.

df_drop.drop(columns = 'brown')

```
In [8]: df_drop.drop(columns = 'brown')
```

Out[8]:

	hair_color	black	blonde
gender	country		
F	India	1	0
	UK	0	2
	USA	0	2
M	India	2	0
	UK	0	0
	USA	0	0

Alternative to specifying axis (labels, axis=1 is equivalent to columns=labels).
i.e.

df_drop.drop('brown', axis=1)

Here, we can see that we have successfully removed the column ‘brown’.

16.2.1 Delete multiple columns

To delete multiple columns we have to pass a list to the *‘columns’* parameter.

df_drop.drop(columns = ['brown', 'black'])

```
In [9]: df_drop.drop(columns = ['brown', 'black'])
```

Out[9]:

		hair_color	blonde
gender	country		
F	India		0
	UK		2
	USA		2
M	India		0
	UK		0
	USA		0

With this we have removed two columns (‘brown’ and ‘black’).

16.2.2 Delete multilevel columns

We can even specify the levels to be deleted in columns. Let’s take a multilevel column data by applying crosstab on the existing data.

df_drop = pd.crosstab(df.country, [df.gender, df.hair_color])

```
In [10]: df_drop = pd.crosstab(df.country,
                               [df.gender, df.hair_color])
         df_drop
```

Out[10]:

gender	F			M	
hair_color	black	blonde	brown	black	brown
country					
India	1	0	0	2	0
UK	0	2	1	0	1
USA	0	2	0	0	1

Let’s drop some columns at level 1.

df_drop.drop(columns = ['black', 'blonde'], level=1)

```
In [11]: df_drop.drop(columns = ['black', 'blonde'], level=1)
```

Out[11]:

gender	F	M
hair_color	brown	brown
country		
India	0	0
UK	1	1
USA	0	1

We have removed two columns (black & blonde) at level 1.

16.3 Delete both rows & columns

We have the facility to pass both *'index'* (rows) and *'columns'* as parameter to delete both rows and columns together.

df_drop.drop(index='UK', columns='M')

```
In [12]: df_drop.drop(index='UK', columns='M')
```

Out[12]:

gender	F		
hair_color	black	blonde	brown
country			
India	1	0	0
USA	0	2	0

Here we have removed a row (UK) and a column (M) simultaneously.

This is all about dropping the rows and columns but what about the duplicate values in the Dataframe? Let's learn the ways to deleting the duplicate data, in next chapter.

17 Remove duplicate values

Many times it's important to handle the duplicate values in the data so that we can proceed with the further analysis.
Let's take an example to understand this and look at various operations.

We are taking a data of various people, various instruments they play and the date they started to learn them.

df = pd.DataFrame({
"name":['arun', 'varun', 'neha', 'varun', 'varun', 'arun'],
'instruments':['violin', 'drum', 'flute', 'guitar', 'bongo','tabla'],
'start_date': ['Jan 10, 2020', 'Mar 3, 2003', 'Feb 6, 2005', 'Dec 8, 2008', 'Nov 5, 2011', 'Mar 10, 2011']
})
df.start_date = pd.to_datetime(df.start_date)

In [1]:
```
import pandas as pd
```

In [2]:
```
df = pd.DataFrame({
    "name":['arun', 'varun', 'neha', 'varun', 'varun', 'arun'],
    'instruments':['violin', 'drum', 'flute', 'guitar', 'bongo','tabla']
    'start_date': ['Jan 10, 2020',
                   'Mar 3, 2003',
                   'Feb 6, 2005',
                   'Dec 8, 2008',
                   'Nov 5, 2011',
                   'Mar 10, 2011']
})
df.start_date = pd.to_datetime(df.start_date)
df
```

Out[2]:

	name	instruments	start_date
0	arun	violin	2020-01-10
1	varun	drum	2003-03-03
2	neha	flute	2005-02-06
3	varun	guitar	2008-12-08
4	varun	bongo	2011-11-05
5	arun	tabla	2011-03-10

17.1 Remove duplicate

Pandas makes it very easy to handle the duplicate values with the use of *drop_duplicate()*.

df.drop_duplicates(subset = 'name')

OR

df.drop_duplicates('name')

```
In [3]: df.drop_duplicates('name')
```

Out[3]:

	name	instruments	start_date
0	arun	violin	2011-01-10
1	varun	drum	2003-03-03
2	neha	flute	2005-02-06

This provides a result without the duplicates. Here the important point to note is that the remaining rows are the rows with their first occurrence.

17.2 Fetch custom occurrence of data

By default *drop_duplicates()* keeps the first occurrence of the duplicate value and deletes other. We can opt to choose our own occurrence of value (first or last). These all are controlled by the parameter *'keep'*.

17.2.1 First occurrence

This is a by default function and can be achieved as:

df.drop_duplicates('name')

OR

df.drop_duplicates('name', keep='first')

This kind of operation helps us to analyse the first instrument learned by respective individual and their start data.

17.2.2 Last occurrence

This can be achieved by passing value 'last' to *'keep'* parameter.

df.drop_duplicates('name', keep='last')

```
In [5]: df.drop_duplicates('name', keep='last')
```

Out[5]:

	name	instruments	start_date
2	neha	flute	2005-02-06
4	varun	bongo	2011-11-05
5	arun	tabla	2020-03-10

This kind of operation helps us to analyse the last instrument learned by respective individual and their start data.

17.2.3 Remove all duplicates

There is functionality to remove the duplicates completely.

df.drop_duplicates('name', keep=False)

```
In [6]: df.drop_duplicates('name', keep=False)
```

Out[6]:

	name	instruments	start_date
2	neha	flute	2005-02-06

There is only one entry of ‘neha’ and thus remained in the result and those who are having duplicate values are removed completely just by passing ‘False’ to *‘keep’* parameter.

17.3 Ignore index

while deleting the duplicates the indexes becomes nonsequential. This can be handled by a parameter *‘ignore_index’*.

df.drop_duplicates('name', keep='last', ignore_index=True)

```
In [8]: df.drop_duplicates('name',
                           keep='last',
                           ignore_index=True)
```

Out[8]:

	name	instruments	start_date
0	neha	flute	2005-02-06
1	varun	bongo	2011-11-05
2	arun	tabla	2020-03-10

With this, the index seems to be in sequence.

We may have need to sort the data too which we'll learn in the next chapter.

18 Sort the data

Sorting is a basic and important operation in any data analysis. Pandas provides us with a dataframe function *sort_values()*. This function facilitates various sorting operations in Pandas.

Let's look at the various operations with an example.

df = pd.DataFrame({
'alphabet':list('dpbtbkc'),
'num1':[1,2,np.nan,4,3,7,2],
'num2':[3,4,3,4,2,5,4]
})

In [1]:
```
import pandas as pd
import numpy as np
```

In [2]:
```
df = pd.DataFrame({
    'alphabet':list('dpbtbkc'),
    'num1':[1,2,np.nan,4,3,7,2],
    'num2':[3,4,3,4,2,5,4]
})
df
```

Out[2]:

	alphabet	num1	num2
0	d	1.0	3
1	p	2.0	4
2	b	NaN	3
3	t	4.0	4
4	b	3.0	2
5	k	7.0	5
6	c	2.0	4

18.1 Sort columns

Sorting is generally done on columns and we have to provide a column name on which we have to do the operation. *Columns are on axis 0*, which is set by default. We rarely do any sorting operation on rows but if that's needed then we have to a pass a parameter *axis=1*.

df.sort_values(by='alphabet')

OR

df.sort_values(by=['alphabet'])

OR

df.sort_values('alphabet')

```
In [3]:
1 df.sort_values(by='alphabet')
2 #df.sort_values(by=['alphabet'])
3 #df.sort_values('alphabet')
```

Out[3]:

	alphabet	num1	num2
2	b	NaN	3
4	b	3.0	2
6	c	2.0	4
0	d	1.0	3
5	k	7.0	5
1	p	2.0	4
3	t	4.0	4

If we'll observe, we can see that the 'alphabet' column has been sorted.

18.2 Sorting multiple columns

We can sort multiple columns also. For this we just need to pass a list of the columns that needs to be sorted.

df.sort_values(by=['alphabet', 'num2'])

```
In [4]: df.sort_values(by=['alphabet', 'num2'])
        #df.sort_values(['alphabet', 'num2'])
```

Out[4]:

	alphabet	num1	num2
4	b	3.0	2
2	b	NaN	3
6	c	2.0	4
0	d	1.0	3
5	k	7.0	5
1	p	2.0	4
3	t	4.0	4

In multiple column sort, the first preference is given to the first column in the list and if there are same values in the first column then the sorting is performed on the second column in the list. Here in our case we can see that the sorting is done on the alphabet column and when ‘b’ occurs twice the other column (num2) has been sorted.

18.3 Sorting order

By default the sorting order is descending. If we need the ascending order, we have to pass a parameter *‘ascending=False’* (by default its True).

```
In [5]: df.sort_values(by='alphabet', ascending=False)
```

Out[5]:

	alphabet	num1	num2
3	t	4.0	4
1	p	2.0	4
5	k	7.0	5
0	d	1.0	3
6	c	2.0	4
2	b	NaN	3
4	b	3.0	2

With this we can see that the ‘alphabet’ column has been sorted in descending order.

18.4 Positioning missing value

We can see that the column ‘num1’ has a missing value. When we apply the sort on the column with the missing value, by default the mussing value goes to the end of the dataframe.

```
In [6]: # by default missing value is at the end

        df.sort_values(by='num1')
```

Out[6]:

	alphabet	num1	num2
0	d	1.0	3
1	p	2.0	4
6	c	2.0	4
4	b	3.0	2
3	t	4.0	4
5	k	7.0	5
2	b	NaN	3

We can bring that missing value to starting by passing value ‘first’ to a parameter *‘na_position’*.

df.sort_values(by='num1', na_position='first')

```
In [7]: df.sort_values(by='num1', na_position='first')
```

Out[7]:

	alphabet	num1	num2
2	b	NaN	3
0	d	1.0	3
1	p	2.0	4
6	c	2.0	4
4	b	3.0	2
3	t	4.0	4
5	k	7.0	5

With this the missing value in column ‘num1’ came to the beginning of the

dataframe.

With this we are done with the sorting operation with Pandas. One more point to know is that there are various ways of sorting (quicksort, mergesort and heapsort) that Pandas support. This can be selected by a parameter *'kind'*. This parameter decides the kind of the sort we need to perform. By default quicksort is selected.

Let's look at a the ways of working and handling the datetime series in coming chapter.

19 Working with date and time

In real life, managing date and time in data could be a big deal and sometime difficult too. Pandas have great ability to deal with the date and time series. Generally people who does financial analysis tasks like revenue reports, productivity reports etc. has to deal with time series a lot more.

Let's look at some of the functionalities of Pandas with time series.

19.1 Creation, working and use of DatetimeIndex

We have seen the ways to convert columns to index and various operations that can be done with indexes. When we convert the date as index, it's called "DatetimeIndex". This many times very useful for analysis purpose and mainly whenever we are dealing with financial data.

Let's take some data to understand this

We are here taking TCS data for the financial year 2019 - 2020. (you can go to google finance and get the data for companies from there).

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_csv('TCS_data.csv')
        df
```

Out[2]:

	Date	Open	High	Low	Close	Adj Close	Volume
0	2019-04-01	2010.000000	2039.949951	2008.250000	2031.650024	1986.672485	2095740.0
1	2019-04-02	2037.099976	2086.000000	2037.000000	2079.300049	2033.267822	3719663.0
2	2019-04-03	2085.000000	2089.600098	2058.100098	2079.300049	2033.267822	2939886.0
3	2019-04-04	2078.149902	2079.699951	2007.400024	2014.500000	1969.901978	4397518.0
4	2019-04-05	2028.650024	2054.399902	2018.800049	2048.300049	2002.953735	3152103.0
...	...	...	...	...	...	...	...
240	2020-03-24	1653.050049	1770.000000	1632.849976	1703.150024	1703.150024	6350865.0
241	2020-03-25	1700.000000	1810.000000	1680.000000	1750.300049	1750.300049	2765520.0
242	2020-03-26	1831.599976	1832.050049	1722.550049	1790.949951	1790.949951	4556067.0
243	2020-03-27	1820.000000	1850.000000	1750.400024	1824.500000	1824.500000	4331250.0
244	2020-03-30	1766.000000	1905.000000	1763.550049	1778.500000	1778.500000	8513547.0

245 rows × 7 columns

19.1.1 Converting date to timestamp and set as index

In the above data the date is a normal string.

```
In [3]: type(df["Date"][0])
Out[3]: str
```

Firstly, we have to convert this to timestamp format and then set it as index. These two processes can be done separately also as we have seen previously. i.e.
make “Date” as timestamp
df['Date'] = pd.to_datetime(df['Date'])

set the “Date” column as index
df.set_index('Date', inplace=True)

These two process can be done together while reading the data it’self

df = pd.read_csv('TCS_data.csv', parse_dates=["Date"], index_col="Date")

“parse_dates” makes the column as timestamp and “index_col” makes the column as index.

```
In [4]: df = pd.read_csv('TCS_data.csv', parse_dates=["Date"], index_col="Date")
        df.head()
```

Out[4]:

Date	Open	High	Low	Close	Adj Close	Volume
2019-04-01	2010.000000	2039.949951	2008.250000	2031.650024	1986.672485	2095740.0
2019-04-02	2037.099976	2086.000000	2037.000000	2079.300049	2033.267822	3719663.0
2019-04-03	2085.000000	2089.600098	2058.100098	2079.300049	2033.267822	2939886.0
2019-04-04	2078.149902	2079.699951	2007.400024	2014.500000	1969.901978	4397518.0
2019-04-05	2028.650024	2054.399902	2018.800049	2048.300049	2002.953735	3152103.0

Now, if we’ll check the index of the DataFrame we’ll observe that the index is formatted as *DatetimeIndex.*

df.index

```
In [5]: df.index
```

```
Out[5]: DatetimeIndex(['2019-04-01', '2019-04-02', '2019-04-03', '2019-04-04',
                       '2019-04-05', '2019-04-08', '2019-04-09', '2019-04-10',
                       '2019-04-11', '2019-04-12',
                       ...
                       '2020-03-17', '2020-03-18', '2020-03-19', '2020-03-20',
                       '2020-03-23', '2020-03-24', '2020-03-25', '2020-03-26',
                       '2020-03-27', '2020-03-30'],
                      dtype='datetime64[ns]', name='Date', length=245, freq=None)
```

Now, there are multiple advantages to having the data with DatetimeIndex

19.1.2 Access data for particular year

We can now easily access the data for any year as:

df["2020"]

```
In [6]: df["2020"]
```

Out[6]:

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908.0
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752.0
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761.0
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209.0
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317.0
...	...	...	...	...	...	...
2020-03-24	1653.050049	1770.000000	1632.849976	1703.150024	1703.150024	6350865.0
2020-03-25	1700.000000	1810.000000	1680.000000	1750.300049	1750.300049	2765520.0
2020-03-26	1831.599976	1832.050049	1722.550049	1790.949951	1790.949951	4556067.0
2020-03-27	1820.000000	1850.000000	1750.400024	1824.500000	1824.500000	4331250.0
2020-03-30	1766.000000	1905.000000	1763.550049	1778.500000	1778.500000	8513547.0

62 rows × 6 columns

19.1.3 Access data for particular month

We can easily access any month of the year and get the monthly data.

df["2020-01"]

```
In [7]: df["2020-01"].head()
```

Out[7]:

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908.0
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752.0
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761.0
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209.0
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317.0

19.1.4 Calculating average closing price for any month

As we can access any year, month etc, we can now apply function to those data directly.

df["2020-01"].Close.mean()

```
In [8]: df["2020-01"].Close.mean()
Out[8]: 2188.8934910434787
```

This will take the monthly data, get the Closing price and apply mean to that data and thus we get the mean of the closing price for that month.

19.1.5 Access a date range

With DatetimeIndex we can now easily access data for a range of dates too.

df["2020-01-01":"2020-01-07"]

```
In [9]: df["2020-01-01":"2020-01-07"]
```

Out[9]:

Date	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908.0
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752.0
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761.0
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209.0
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317.0

We have taken a range of data for the first week of January, 2020. Now we can apply any function to this as per our need.

19.1.6 Resampling the data

Many times we may have to access specific sample of the data and Pandas allows us to do this by a method called "resample()". resample() is a time-based groupby, followed by a reduction method on each of it's groups. The resample function is flexible and allows us to specify many different values to control the frequency conversion and resampling operation.
For example:

df.Close.resample("M").mean()

The above command:

- Takes the column(Close)
- Resamples on monthly frequency
- Takes mean of each month closing price

```
In [10]: df.Close.resample("M").mean()

Out[10]: Date
         2019-04-30    2109.397390
         2019-05-31    2120.722734
         2019-06-30    2241.686858
         2019-07-31    2145.432596
         2019-08-31    2219.017505
         2019-09-30    2131.200003
         2019-10-31    2078.810528
         2019-11-30    2131.205023
         2019-12-31    2128.897618
         2020-01-31    2188.893491
         2020-02-29    2145.097380
         2020-03-31    1841.204993
         Freq: M, Name: Close, dtype: float64
```

Here 'M' is monthly frequency. The detail of all the various frequencies are:

Frequency String	Description
None	Generic offset class, defaults to 1 calendar day
'B'	business day (weekday)

'C'	custom business day
'W'	one week, optionally anchored on a day of the week
'WOM'	the x-th day of the y-th week of each month
'LWOM'	the x-th day of the last week of each month
'M'	calendar month end
'MS'	calendar month begin
'BM'	business month end
'BMS'	business month begin
'CBM'	custom business month end
'CBMS'	custom business month begin
'SM'	15th (or other day_of_month) and calendar month end
'SMS'	15th (or other day_of_month) and calendar month begin
'Q'	calendar quarter end
'QS'	calendar quarter begin
'BQ	business quarter end
'BQS'	business quarter begin
'REQ'	retail (aka 52-53 week) quarter
'A'	calendar year end
'AS' or 'BYS'	calendar year begin
'BA'	business year end
'BAS'	business year begin
'RE'	retail (aka 52-53 week) year
None	Easter holiday
'BH'	business hour
'CBH'	custom business hour
'D'	one absolute day
'H'	one hour
'T' or 'min'	one minute

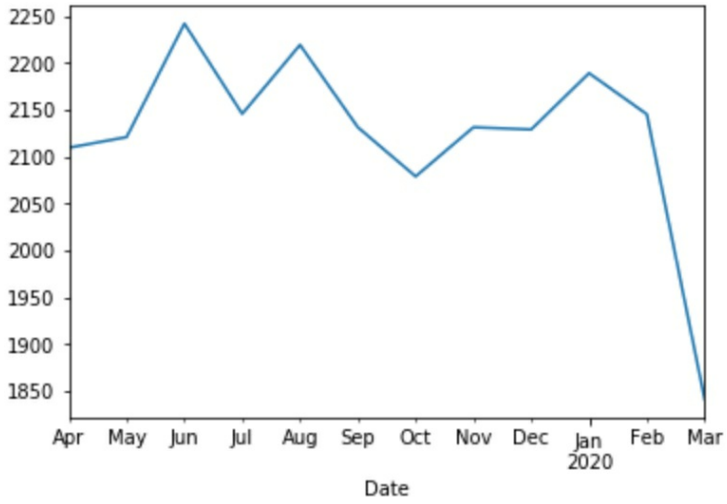
'S'	one second
'L' or 'ms'	one millisecond
'U' or 'us'	one microsecond
'N'	one nanosecond

19.1.7 Plotting the resampled data

We can even plot the above data and analyse easily.

df.Close.resample("M").mean().plot()

```
In [11]: df.Close.resample("M").mean().plot()
Out[11]: <matplotlib.axes._subplots.AxesSubplot at 0x116fcf7c0>
```



How easy it is to get the data and analyse.

19.1.8 Quarterly frequency

Generally companies are interested in quarterly data and it's performance. We can even resample the data with a quarterly frequency and analyse.

df.Close.resample("Q").mean()

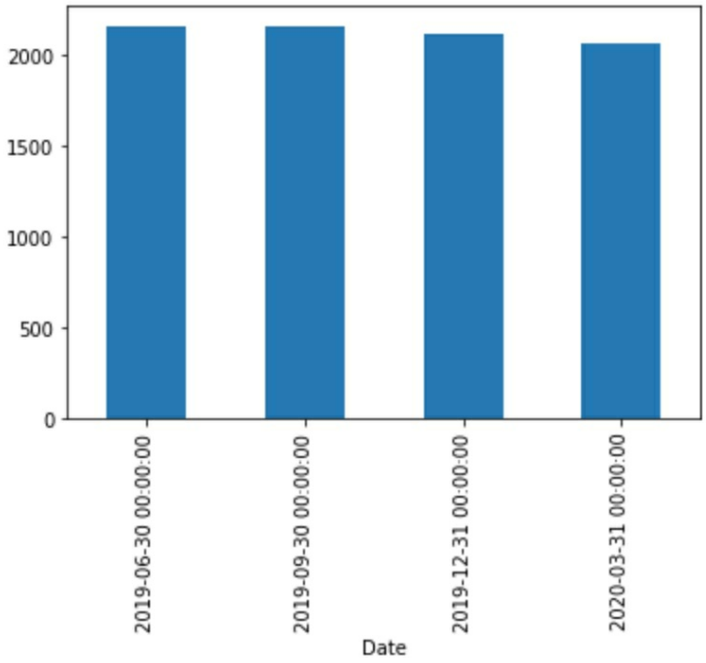
"Q" provides the quarterly frequency.

```
In [12]: df.Close.resample("Q").mean()

Out[12]: Date
         2019-06-30    2155.441681
         2019-09-30    2164.808062
         2019-12-31    2113.805841
         2020-03-31    2063.314522
         Freq: Q-DEC, Name: Close, dtype: float64
```

We can plot and analyse the performance easily and quickly.

df.Close.resample("Q").mean().plot(kind="bar")

```
In [13]: df.Close.resample("Q").mean().plot(kind="bar")

Out[13]: <matplotlib.axes._subplots.AxesSubplot at 0x11918dbe0>
```



'Kind' parameter in plot tells the type of plot we need. here we are taking a bar plot and this makes the job quick and easy.

19.2 Working with date ranges

Many a times we face issues where the data does not have dates and we may have to supply one. This would be a very difficult task without Pandas *date_range()* functionality.

With *date_range()* we can generate a range of dates needed for the task.

Let’s take some data to work with

We are here taking TCS data for the month of January, 2020. (you can go to google finance and get the data for companies from there).

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv("TCS_data_withoutdate.csv")
        df.head()
```

Out[2]:

	Open	High	Low	Close	Adj Close	Volume
0	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908
1	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752
2	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761
3	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209
4	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317

This data will be having dates, we have removed the date column for our exercise.

19.2.1 Adding dates to the data

Now let’s try to add the dates to this data. For this we’ll be using *date_range()* function from Pandas.

19.2.1.1 Date range between start and end dates

rng=pd.date_range(start="01/01/2020", end="01/31/2020", freq="B")

here the date_range() will be taking a start date, a end date and a frequency.
Start: this parameter will have the start date
End: this parameter will have the end date
Freq: this parameter will define the frequency on which we need the dates, i.e. if we want business days to be coming between the start and the end dates then we provide the value as ‘B’. Complete list of the frequencies are as in the above frequency table.

```
In [3]: rng = pd.date_range(start="01/01/2020", end="01/31/2020", freq="B")
        rng

Out[3]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
                       '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
                       '2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16',
                       '2020-01-17', '2020-01-20', '2020-01-21', '2020-01-22',
                       '2020-01-23', '2020-01-24', '2020-01-27', '2020-01-28',
                       '2020-01-29', '2020-01-30', '2020-01-31'],
                      dtype='datetime64[ns]', freq='B')
```

The above output generates January, 2020 week days.
We can see that day 4 and 5 are missing as those were Saturday and Sunday and similarly other week ends too.

19.2.2 Apply the above date range to our data

To apply the above range to our data and set that as the datetimeIndex we can:

df.set_index(rng, inplace=True)

```
In [4]: df.set_index(rng, inplace=True)
        df.head(10)
```

Out[4]:

	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317
2020-01-08	2205.000000	2260.000000	2202.050049	2255.250000	2233.819336	5197454
2020-01-09	2248.750000	2251.949951	2210.000000	2214.350098	2193.308105	3734173
2020-01-10	2228.000000	2234.000000	2208.000000	2213.550049	2192.515625	1915807
2020-01-13	2217.850098	2218.949951	2184.699951	2190.350098	2169.536133	2843893
2020-01-14	2195.000000	2229.800049	2195.000000	2206.899902	2185.928467	2948452

We can see that we set the date range generated as the DatetimeIndex to our data.
We are only getting the weekdays (business days) here. What if we need to have all the dates including the weekends.

One way would have to have generated the range with all the days by passing ‘freq’ and ‘D’

rng = pd.date_range(start="01/01/2020", end="01/31/2020", freq="D")

This would have given all the dates, but this can’t be applied because we have data for only working days. By above method we would have generated the dates but would have not able to map it with the data.

19.2.3 Generate the missing data with missing dates

To get the missing dates we can use ‘*asfreq()*’ function on the DataFrame.

df.asfreq("D")

The ‘D’ here is freq to be generated (as per the freq table above)

```
In [5]: df.asfreq("D").head(7)
```

Out[5]:

	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908.0
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752.0
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761.0
2020-01-04	NaN	NaN	NaN	NaN	NaN	NaN
2020-01-05	NaN	NaN	NaN	NaN	NaN	NaN
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209.0
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317.0

This generated the dates but the data is still missing. To fill the data we can do various methods as we learned in previous chapters. This filling can also be done with *asfreq()* directly by.

df.asfreq("D", method='pad')

‘method’ parameter provides the way to fill the data. ‘pad’ or ‘ffill’ is the way to fill the above values to the missing ones (as discussed in the previous chapters).

```
In [9]: df.asfreq("D", method='ffill').head(7)
```

Out[9]:

	Open	High	Low	Close	Adj Close	Volume
2020-01-01	2168.000000	2183.899902	2154.000000	2167.600098	2147.002441	1354908
2020-01-02	2179.949951	2179.949951	2149.199951	2157.649902	2137.146484	2380752
2020-01-03	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761
2020-01-04	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761
2020-01-05	2164.000000	2223.000000	2164.000000	2200.649902	2179.738037	4655761
2020-01-06	2205.000000	2225.949951	2187.899902	2200.449951	2179.540039	3023209
2020-01-07	2200.500000	2214.649902	2183.800049	2205.850098	2184.888672	2429317

We can get the weekly prices too,

```
In [7]: df.asfreq("W", method="pad")
```

Out[7]:

	Open	High	Low	Close	Adj Close	Volume
2020-01-05	2164.000000	2223.000000	2164.0	2200.649902	2179.738037	4655761
2020-01-12	2228.000000	2234.000000	2208.0	2213.550049	2192.515625	1915807
2020-01-19	2240.750000	2253.550049	2213.0	2219.100098	2198.012939	3281059
2020-01-26	2190.949951	2190.949951	2170.0	2183.399902	2167.562744	1319430

19.2.4 Date range with periods

We have one more way to generate the date range with *date_range()* function.

rng = pd.date_range(start="01/01/2020", periods=23, freq="B")

This will also give the same output but just in case you want the result (date range) for particular periods the we can pass *'periods'* parameter instead of *'end'* parameter

```
In [8]: rng = pd.date_range(start="01/01/2020", periods=23, freq="B")
        rng

Out[8]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-06',
                       '2020-01-07', '2020-01-08', '2020-01-09', '2020-01-10',
                       '2020-01-13', '2020-01-14', '2020-01-15', '2020-01-16',
                       '2020-01-17', '2020-01-20', '2020-01-21', '2020-01-22',
                       '2020-01-23', '2020-01-24', '2020-01-27', '2020-01-28',
                       '2020-01-29', '2020-01-30', '2020-01-31'],
                      dtype='datetime64[ns]', freq='B')
```

Here, we have passed periods as 23 which gives the 23 business days (as *freq* is 'B') from start date.

19.3 Working with custom holidays

In the previous topic we have seen various ways to create date ranges for various frequencies but in reality apart from the weekends we may have various holidays (like new year, birthday, etc.). we may need to add these holidays also into our data. Pandas provides us the functionality to create data ranges with these custom holidays too.

Let,s take an example of December, 2019.

pd.date_range(start="12/01/2019", end="12/31/2019", freq="B")

```
In [1]: import pandas as pd

In [2]: pd.date_range(start="12/01/2019", end="12/31/2019", freq="B")

Out[2]: DatetimeIndex(['2019-12-02', '2019-12-03', '2019-12-04', '2019-12-05',
                       '2019-12-06', '2019-12-09', '2019-12-10', '2019-12-11',
                       '2019-12-12', '2019-12-13', '2019-12-16', '2019-12-17',
                       '2019-12-18', '2019-12-19', '2019-12-20', '2019-12-23',
                       '2019-12-24', '2019-12-25', '2019-12-26', '2019-12-27',
                       '2019-12-30', '2019-12-31'],
                      dtype='datetime64[ns]', freq='B')
```

The above command provides all the working date (business days) of December, 2019.

19.3.1 Adding US holidays

In the above data we have the business days but 25th was a holiday (Christmas), which we may have to add.

from Pandas.tseries.holiday import USFederalHolidayCalendar

from Pandas.tseries.offsets import CustomBusinessDay

usb = CustomBusinessDay(calendar=USFederalHolidayCalendar())

```
In [3]: from pandas.tseries.holiday import USFederalHolidayCalendar
        from pandas.tseries.offsets import CustomBusinessDay

        usb = CustomBusinessDay(calendar=USFederalHolidayCalendar())
        usb

Out[3]: <CustomBusinessDay>
```

This creates a custom frequency for only US related holdays.

```
In [4]: # The 25th has been removed as it was Christmas in US

        rng = pd.date_range(start="12/01/2019", end="12/31/2019", freq=usb)
        rng

Out[4]: DatetimeIndex(['2019-12-02', '2019-12-03', '2019-12-04', '2019-12-05',
                       '2019-12-06', '2019-12-09', '2019-12-10', '2019-12-11',
                       '2019-12-12', '2019-12-13', '2019-12-16', '2019-12-17',
                       '2019-12-18', '2019-12-19', '2019-12-20', '2019-12-23',
                       '2019-12-24', '2019-12-26', '2019-12-27', '2019-12-30',
                       '2019-12-31'],
                      dtype='datetime64[ns]', freq='C')
```

We can pass the custom frequency created above as the part of the date_range() and can generate the US related holday's.

In the above result we can see that the 25th has been eliminated as it was Christmas Holiday in US.

19.3.2 Creating custom calendar

We have created the date range for US but what if we need other holidays or custom holidays like birthday or company holiday etc. This can be done by some changes to the existing class "USFederalHolidayCalendar". This class is present in *Pandas.tseries.holiday* file (https://github.com/Pandas-dev/Pandas/blob/master/Pandas/tseries/holiday.py).

We just have to edit the name of the class and the rules as per our own desired holidays.

from Pandas.tseries.holiday import Holiday, AbstractHolidayCalendar,

nearest_workday

```
class myHolidayCalender(AbstractHolidayCalendar):
    rules = [
        Holiday('ExampleHoliday1', month=1, day=2),
        Holiday('ExampleHoliday1', month=1, day=7)
    ]
myc = CustomBusinessDay(calendar=myHolidayCalender())
```

```
In [5]: from pandas.tseries.holiday import Holiday, \
        AbstractHolidayCalendar, \
        nearest_workday

        class myHolidayCalender(AbstractHolidayCalendar):
            rules = [
                Holiday('ExampleHoliday1', month=1, day=2),
                Holiday('ExampleHoliday1', month=1, day=7)
            ]
        myc = CustomBusinessDay(calendar=myHolidayCalender())
        myc

Out[5]: <CustomBusinessDay>
```

This will create a custom frequency (myc) which can be later passed to the date_range().

pd.date_range(start="1/1/2020", end="1/11/2020", freq=myc)

```
In [6]: pd.date_range(start="1/1/2020", end="1/11/2020", freq=myc)

Out[6]: DatetimeIndex(['2020-01-01', '2020-01-03', '2020-01-06', '2020-01-08',
                       '2020-01-09', '2020-01-10'],
                      dtype='datetime64[ns]', freq='C')
```

We can see that the dates that we have provided as holidays in myHolidayCalender (2nd and 7th) have been removed from the date range.

19.3.3 Observance rule

There are holidays that occurs on a fixed dates. We can define an observance rule to determine when that holiday should be observed if that falls on the weekend or some other non-observed day.

This can be done with a parameter '*observance*' and the value to be passed to the are:

Rule	Description
nearest_workday	move Saturday to Friday and Sunday to Monday
sunday_to_monday	move Sunday to following Monday
next_monday_or_tuesday	move Saturday to Monday and Sunday/Monday to Tuesday
previous_friday	move Saturday and Sunday to previous Friday"
next_monday	move Saturday and Sunday to following Monday

```
In [7]: from pandas.tseries.holiday import Holiday, \
        AbstractHolidayCalendar, nearest_workdayt_workday, \
        nearest_workday

        class myHolidayCalender(AbstractHolidayCalendar):
            rules = [
                Holiday('ExampleHoliday1', month=1, day=5, observance=nearest_workday),
                Holiday('ExampleHoliday1', month=1, day=9)
            ]
        myc = CustomBusinessDay(calendar=myHolidayCalender())
        myc

Out[7]: <CustomBusinessDay>

In [8]: # the 5th was sunday thus the nearest weekday was 6th(monday)
        # if we would have provided holiday on 4th then 3rd would have been the nearest holiday
        pd.date_range(start="1/1/2020", end="1/11/2020", freq=myc)

Out[8]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-03', '2020-01-07',
                       '2020-01-08', '2020-01-10'],
                      dtype='datetime64[ns]', freq='C')
```

It can be observed that 4th and 5th were week end but as our holiday falls on weekend(I.e. 5th), because of the observance rule we have provided, the holiday falls on the nearest available day i.e. 6th (Monday). If our holiday would have been on 4th (Saturday) then the nearest available day would be 3rd (Friday).

19.3.4 Custom week days

There might be cases where the weekends are different (Egypt where a Friday-Saturday weekend is observed). The other example would be companies which have week days as there weekend so that they can work on Saturdays and Sundays. To deal with such issues Pandas allows us to create custom week.

custom_weekdays = 'Sun Mon Tue Wed Thu'
custom_businessDays = CustomBusinessDay(weekmask=custom_weekdays)
pd.date_range(start="1/1/2020", end="1/11/2020", freq=custom_businessDays)

```
In [9]: custom_weekdays = 'Sun Mon Tue Wed Thu'
        custom_businessDays = CustomBusinessDay(weekmask=custom_weekdays)
        pd.date_range(start="1/1/2020", end="1/11/2020", freq=custom_businessDays)

Out[9]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-05', '2020-01-06',
                       '2020-01-07', '2020-01-08', '2020-01-09'],
                      dtype='datetime64[ns]', freq='C')
```

In This example Friday(3rd) and Saturday(4th) are the weekends.
Here we have passed a weekmask as our custom week days. And thus we get a custom week by passing the custom frequency.

19.3.5 Custom holiday

If we need to add some custom holiday to the above date range, then we just have to pass a *'holiday'* parameter and pass the list of dates which needs to be treated as holiday.

custom_weekdays = 'Sun Mon Tue Wed Thu'
custom_businessDays = CustomBusinessDay(weekmask=custom_weekdays, holidays=["2020-01-06"])
pd.date_range(start="1/1/2020", end="1/11/2020", freq=custom_businessDays)

```
In [10]: custom_weekdays = 'Sun Mon Tue Wed Thu'
         custom_businessDays = CustomBusinessDay(weekmask=custom_weekdays, holidays=["2020-01-06"])
         pd.date_range(start="1/1/2020", end="1/11/2020", freq=custom_businessDays)

Out[10]: DatetimeIndex(['2020-01-01', '2020-01-02', '2020-01-05', '2020-01-07',
                        '2020-01-08', '2020-01-09'],
                       dtype='datetime64[ns]', freq='C')
```

Here in this example we have passed Friday(3rd) and Saturday(4th) as

weekends and passed 6th as a holiday to *'holiday'* parameter and in result we can see that all the three dates are removed from the date range list.

19.4 Working with date formats

There are multiple ways to write a date ('2020-03-10', 'Mar 10, 2020', '03/10/2020', '2020.03.10','2020/03/10', '20200310') all of them are correct but to work efficiently we need to have a common date format. Pandas operation *'to_datetime()'* provides us the flexibility to convert all the formats to a common format.

19.4.1 Converting to a common format

Let's take a date (10th March, 2020) in various formats and try to look how Pandas helps us to brings all the dates to a common page.

dates = ['2020-03-10', 'Mar 10, 2020', '03/10/2020', '2020.03.10','2020/03/10', '20200310']
pd.to_datetime(dates)

```
In [1]: import pandas as pd

In [2]: # 10th March 2020
        dates = ['2020-03-10', 'Mar 10, 2020', '03/10/2020', '2020.03.10','2020/03/10', '20200310']
        pd.to_datetime(dates)

Out[2]: DatetimeIndex(['2020-03-10', '2020-03-10', '2020-03-10', '2020-03-10',
                       '2020-03-10', '2020-03-10'],
                      dtype='datetime64[ns]', freq=None)
```

Here, we can see that all the various formats of dates are converted to a. common date format.

19.4.2 Time conversion

Similar to dates we can bring all the time formats also to a common format.

dates = ['2020-03-10 04:30:00 PM', 'Mar 10, 2020 16:30:00', '03/10/2020', '2020.03.10','2020/03/10', '20200310']
pd.to_datetime(dates)

```
In [3]: dates = ['2020-03-10 04:30:00 PM',
                 'Mar 10, 2020 16:30:00',
                 '03/10/2020',
                 '2020.03.10',
                 '2020/03/10',
                 '20200310']
        pd.to_datetime(dates)

Out[3]: DatetimeIndex(['2020-03-10 16:30:00', '2020-03-10 16:30:00',
                       '2020-03-10 00:00:00', '2020-03-10 00:00:00',
                       '2020-03-10 00:00:00', '2020-03-10 00:00:00'],
                      dtype='datetime64[ns]', freq=None)
```

Here, we can see that both the different time formats (04:30 PM and 16:30) have became a single format (16:30)

19.4.3 Dayfirst formats

Let's take a date (say 10th march, 2020) and we get it as 2020-03-10. We may need this as 2020-10-03 (I.e. day first). This can be done by using *'dayfirst'* parameter in to_datetime().

pd.to_datetime(date, dayfirst=True)

```
In [4]: # 10 Mar, 2020

        date = "10/03/2020"
        pd.to_datetime(date)

Out[4]: Timestamp('2020-10-03 00:00:00')

In [5]: pd.to_datetime(date, dayfirst=True)

Out[5]: Timestamp('2020-03-10 00:00:00')
```

19.4.4 Remove custom delimiter in date

Many time in data the date format may have different delimiters (like ddmmyyyy or dd|mm|yyyy etc.). we can still make all these to a common format so that we can proceed with the analysis further.

date = "10@03@2020"
pd.to_datetime(date, format="%d@%m@%Y")

```
In [6]: date = "10@03@2020"
        pd.to_datetime(date, format="%d@%m@%Y")

Out[6]: Timestamp('2020-03-10 00:00:00')
```

Here, the ddd, mm, yyyy are separated with “@”. We have a parameter *‘format’* where we can supply the unwanted date format provided (like in this case we provided %d@%m@%Y). ‘%d’ is for day, ‘%m’ is for month and ‘%Y’ is for year and between them we provide the unwanted delimiter (@) and rest is taken care by Pandas. It converts the thing to a common timestamp.

19.4.5 Remove custom delimiter in time

Similar to date, time may also have some custom delimiters and they also can be handled as we did in dates.

date = "10@03@2020 04&30"

in this example dates has the delimiter “@” and time has “&”. This can be treated as:

pd.to_datetime(date, format="%d@%m@%Y %H&%M")

```
In [7]: date = "10@03@2020 04&30"
        pd.to_datetime(date, format="%d@%m@%Y %H&%M")

Out[7]: Timestamp('2020-03-10 04:30:00')
```

Here, %H is for hours and %M is for minutes.

19.4.6 Handling errors in datetime

While dealing with dates, we may sometime encounter garbage or invalid values in dates. We may have to handle them. Pandas does this for us.

19.4.6.1 None values

We may see some time a none values in datetime field. This by default is treated as NaT by Pandas.

dates = ['Mar 10, 2020', None]
pd.to_datetime(dates)

```
In [8]: dates = ['Mar 10, 2020', None]
        pd.to_datetime(dates)
```

```
Out[8]: DatetimeIndex(['2020-03-10', 'NaT'], dtype='datetime64[ns]', freq=None)
```

19.4.6.2 Ignore if errors

We have a parameter *'errors'* in to_datetime(). This is by default set to 'raise' which means if it finds any find of garbage value it will raise an error. This can be bypassed by passing 'ignore' to errors.

dates = ['Mar 10, 2020', "xyz"]
pd.to_datetime(dates, errors='ignore')

```
In [9]: # by default this will raise an error
        # to avoid we can pass errors=ignore
        # but none of the other conversions will be done

        dates = ['Mar 10, 2020', "xyz"]
        pd.to_datetime(dates, errors='ignore')
```

```
Out[9]: Index(['Mar 10, 2020', 'xyz'], dtype='object')
```

Here, we can see that by passing the 'ignore' value to *'errors'* parameter, no errors have been raised but the date format is also unchanged.

19.4.6.3 Coerce the error

Though we have some of the datetime as garbage but we may need to change the others so as to proceed further and we may deal with the garbage by some other way. This can be achieved by passing 'coerce' as value to parameter *'errors'*.

dates = ['Mar 10, 2020', "xyz"]
pd.to_datetime(dates, errors='coerce')

```
In [10]: # to make the conversion happen
         # errors='coerce'
         dates = ['Mar 10, 2020', "xyz"]
         pd.to_datetime(dates, errors='coerce')
```

```
Out[10]: DatetimeIndex(['2020-03-10', 'NaT'], dtype='datetime64[ns]', freq=None)
```

We can see that the conversion to the valid dates have been done but the garbage one has been converted to NaT.

19.4.7 Epoch time

The Unix epoch (or Unix time or POSIX time or Unix timestamp) is the number of seconds that have elapsed since January 1, 1970 (midnight UTC/GMT).
Multiple times we may need to work with epoch time. Epoch time also be dealt in Pandas.

19.4.7.1 Epoch to datetime

This is a very simple process and we just have to pass the epoch time to to_datetime().

t = 1583861574
pd.to_datetime(t)

```
In [11]: # by default in ns
         t = 1583861574
         pd.to_datetime(t)

Out[11]: Timestamp('1970-01-01 00:00:01.583861574')
```

By default the result is in nano seconds.

We can get the results in milli seconds or seconds by passing ‘ms’ or ‘s’ for milli seconds and seconds respectively to the *‘unit’* parameter.

dt = pd.to_datetime([t], unit='s')
dt

```
In [12]: # unit = 's' for sec
         # unit = 'ms' for milli sec
         dt = pd.to_datetime([t], unit='s')
         dt

Out[12]: DatetimeIndex(['2020-03-10 17:32:54'], dtype='datetime64[ns]', freq=None)
```

We passed ‘s’ and got result in seconds.

19.4.7.2 Datetime to epoch

We can reverse the above process i.e we can get epoch time from the normal date format. This is generally needed in databases to store epoch time.

We just have to use the function view().

dt.view('int64')

```
In [13]: dt.view('int64')
Out[13]: array([1583861574000000000])
```

This gives back, the epoch time.

19.5 Working with periods

Period is a span of time (a year, month, day, hour etc.), with the use of *'freq'* parameter we can specify what kind of period we need.
Let's look at this with some examples:

19.5.1 Annual period

We can have an annual period as:

y_2020 = pd.Period('2020')

OR

y_2020 = pd.Period('2020', freq='2Y')
we can even provide the frequency and the period become for 2 years (2020 & 2021) starting from 2020.

```
In [1]: import pandas as pd
        import numpy as np

In [2]: y_2020 = pd.Period('2020')
        y_2020
Out[2]: Period('2020', 'A-DEC')
```

Here in output we can see 'A-DEC' which means 'A' as annual and 'DEC' as ends in December.
So annual periods starts in January and ends in December.

19.5.1.1 Operations

Strat time:

y_2020.start_time

```
In [3]: y_2020.start_time
Out[3]: Timestamp('2020-01-01 00:00:00')
```

End time:

y_2020.end_time

```
In [4]: y_2020.end_time
Out[4]: Timestamp('2020-12-31 23:59:59.999999999')
```

Addition/Subtraction to annual period:
We can perform the addition and subtraction of periods to get the new year.

y_2019 = pd.Period("2019")
y_2019 + 1

```
In [5]: y_2019 = pd.Period("2019")
        y_2019 + 1
Out[5]: Period('2020', 'A-DEC')
```

19.5.2 Monthly period

We can have monthly periods as:

m_2020 = pd.Period('2020-1')

OR

m_2020 = pd.Period('2020-2', freq='3M')

with this we create a period of 3 months, starting from 2nd months of 2020 i.e. February, March and April

```
In [6]: m_2020 = pd.Period('2020-1')
        m_2020
Out[6]: Period('2020-01', 'M')
```

19.5.2.1 Operations

Start time:

m_2020.start_time

```
In [7]: m_2020.start_time
Out[7]: Timestamp('2020-01-01 00:00:00')
```

End time:

m_2020.end_time

```
In [8]: m_2020.end_time
Out[8]: Timestamp('2020-01-31 23:59:59.999999999')
```

Addition/Subtraction on monthly period:

m_2020 = pd.Period("2020-01")
m_2020 – 1

```
In [9]: m_2020 = pd.Period("2020-01")
        m_2020 - 1
Out[9]: Period('2019-12', 'M')
```

Subtracting a month from Jan, 2020 gives Dec, 2019

19.5.3 Daily period

We can even get the period on daily basis.

d_2020 = pd.Period('2020-03-01')

OR

d_2020 = pd.Period('2020-03-01', freq="3D")

This will create a frequency of 3 days, starting from 1st to 3rd March, 2020

```
In [10]: d_2020 = pd.Period('2020-03-01')
         d_2020
Out[10]: Period('2020-03-01', 'D')
```

19.5.3.1 Operations

Start time:

d_2020.start_time

```
In [11]: d_2020.start_time
Out[11]: Timestamp('2020-03-01 00:00:00')
```

End time:

d_2020.end_time

```
In [12]: d_2020.end_time
Out[12]: Timestamp('2020-03-01 23:59:59.999999999')
```

Adding/Subtracting days:

d_2020 – 1

```
In [13]: d_2020 - 1
Out[13]: Period('2020-02-29', 'D')
```

The above operating even tells that the operation is aware of the leap year also.

19.5.4 Hourly period

We can even generate hourly period

h_2020 = pd.Period('2020-03-01 04:00')

```
In [14]: h_2020 = pd.Period('2020-03-01 04:00')
         h_2020
Out[14]: Period('2020-03-01 04:00', 'T')
```

This will give us a time period rather than a hourly period. To get the hourly period we have to specify the frequency.

h_2020 = pd.Period('2020-03-01 04:00', freq="H")

```
In [15]: h_2020 = pd.Period('2020-03-01 04:00', freq="H")
         h_2020
Out[15]: Period('2020-03-01 04:00', 'H')
```

19.5.4.1 Operations

Adding/Subtracting hours

h_2020 - 1

```
In [16]: h_2020 - 1
Out[16]: Period('2020-03-01 03:00', 'H')
```

An hour gets reduced in the output.

19.5.5 Quarterly period

Generally companies deal their data per quarter. In such cases quarter period becomes much more important. A year has four quarters (Jan to Mar, Apr to Jun, Jul to Sept, Oct to Dec by default)

19.5.5.1 Only 1st Quarter

q1_2020 = pd.Period('2020', freq='Q')

By default, if we pass freq as “Q” it takes it as 1st quarter (i.e. “1Q”)

```
In [17]: q1_2020 = pd.Period('2020', freq='Q')
         q1_2020
Out[17]: Period('2020Q1', 'Q-DEC')

In [18]: q1_2020.start_time
Out[18]: Timestamp('2020-01-01 00:00:00')

In [19]: q1_2020.end_time
Out[19]: Timestamp('2020-03-31 23:59:59.999999999')
```

19.5.5.2 Multiple Quarters

q_2020 = pd.Period('2020', freq='3Q')

To get multiple Quarters we have to pass the number of quarters needed along with ‘Q’ in frequency. Like if we need three quarters then our *‘freq’* parameter will have “3Q” as value.

```
In [20]: q_2020 = pd.Period('2020', freq='3Q')
         q_2020
Out[20]: Period('2020Q1', '3Q-DEC')

In [21]: q_2020.start_time
Out[21]: Timestamp('2020-01-01 00:00:00')

In [22]: q_2020.end_time
Out[22]: Timestamp('2020-09-30 23:59:59.999999999')
```

19.5.5.3 Defined Quarter

To get a specific quarter we have to pass the quarter with year like:

q3_2020 = pd.Period('2020Q3', freq='Q')

Here, year ‘2020’ is passed along with ‘Q3’ which defined the quarter needed. With this we’ll get the 3rd quarter of 2020 i.e. Jul to Sept.

```
In [23]: q3_2020 = pd.Period('2020Q3', freq='Q')
         q3_2020
Out[23]: Period('2020Q3', 'Q-DEC')

In [24]: q3_2020.start_time
Out[24]: Timestamp('2020-07-01 00:00:00')

In [25]: q3_2020.end_time
Out[25]: Timestamp('2020-09-30 23:59:59.999999999')
```

19.5.5.4 Custom quarters

Some financial year starts with January and ends in December but some may start with April and ends in March. In such cases the quarter one starts in April. To get a period where the quarter starts with April we have to pass the *‘freq’* parameter as ‘Q-MAR’.

q1_2020 = pd.Period('2020Q1', freq='Q-MAR')

This will start the 1st quarter from April.

```
In [26]: q1_2020 = pd.Period('2020Q1', freq='Q-MAR')
         q1_2020
Out[26]: Period('2020Q1', 'Q-MAR')

In [27]: q1_2020.start_time
Out[27]: Timestamp('2019-04-01 00:00:00')
```

19.5.6 Converting one frequency to another

Many times the conversion of one frequency to another might be needed. A very good example is to convert the quarterly frequency to a monthly frequency so that we can break down our quarterly analysis to monthly.

Let's look at the example. We'll take the quarterly frequency we generated in above topic and convert that to monthly. This will done with the help od asfreq().

q1_2020.asfreq('M')

```
In [28]: q1_2020.asfreq('M')
Out[28]: Period('2019-06', 'M')
```

By default the output shows the end of the period.

We can control the frequency with the help of *'how'* parameter and can specify whether to return the starting or the ending month.

q1_2020.asfreq('M', how="start")
q1_2020.asfreq('M', 's')

```
In [29]: q1_2020.asfreq('M', how="start")
Out[29]: Period('2019-04', 'M')
```

OR

q1_2020.asfreq('M', how="end")
q1_2020.asfreq('M', ‘e’)

```
In [30]: q1_2020.asfreq('M', how="end")
Out[30]: Period('2019-06', 'M')
```

19.5.7 Arithmetic between two periods

Sometimes we may need to know the difference or the sum or two periods. for this we may just need to take the periods (of the same frequency) and do the operation between them as shown below:

q1_2020 = pd.Period('2020Q1', freq='Q-MAR')
q_new = pd.Period('2018Q3', freq='Q-MAR')
q1_2020 - q_new

```
In [31]: q1_2020
Out[31]: Period('2020Q1', 'Q-MAR')

In [32]: q_new = pd.Period('2018Q3', freq='Q-MAR')
         q_new
Out[32]: Period('2018Q3', 'Q-MAR')

In [33]: q1_2020 - q_new
Out[33]: <6 * QuarterEnds: startingMonth=3>
```

This shows the difference between is of 6 quarters.

19.6 Period Index

DataFrame indexes with periods are known as period index. There are very useful if we have period data with us.
We can get this as:

idx = pd.period_range('2015', '2020', freq='Q')

```
In [34]: idx = pd.period_range('2015', '2020', freq='Q')
         idx

Out[34]: PeriodIndex(['2015Q1', '2015Q2', '2015Q3', '2015Q4', '2016Q1', '2016Q2',
                      '2016Q3', '2016Q4', '2017Q1', '2017Q2', '2017Q3', '2017Q4',
                      '2018Q1', '2018Q2', '2018Q3', '2018Q4', '2019Q1', '2019Q2',
                      '2019Q3', '2019Q4', '2020Q1'],
                     dtype='period[Q-DEC]', freq='Q-DEC')
```

Here, we are having quarterly frequency from 2015 to 2020. Thus we'll get all the frequencies from 2015 Q1 till 2020 Q1.

19.6.1 Getting given number of periods

period_range() can be passed with a parameter *'periods'* which will provide the periods number of period from the start date.

idx = pd.period_range('2015', periods=5, freq='Q')

```
In [35]: idx = pd.period_range('2015', periods=5, freq='Q')
         idx

Out[35]: PeriodIndex(['2015Q1', '2015Q2', '2015Q3', '2015Q4', '2016Q1'], dtype='period[Q-DEC]', freq='Q-DEC')
```

This provides us 5 periods, starting from 2015.

19.6.2 Period index to DataFrame

Let's add these period indexes to the DataFrame.

ps = pd.Series(np.random.randn(len(idx)), idx)

```
In [36]: ps = pd.Series(np.random.randn(len(idx)), idx)
         ps

Out[36]: 2015Q1    0.297898
         2015Q2    1.928422
         2015Q3    0.194794
         2015Q4    0.922028
         2016Q1   -0.711151
         Freq: Q-DEC, dtype: float64
```

We have created a DataFrame with random numbers and added the period index to that.

19.6.3 Extract annual data

We can now extract the annual data from the period index as:

ps['2015']

```
In [37]: ps['2015']

Out[37]: 2015Q1    0.297898
         2015Q2    1.928422
         2015Q3    0.194794
         2015Q4    0.922028
         Freq: Q-DEC, dtype: float64
```

With period index of quarterly frequency we have extracted the annual data.

19.6.4 Extract a range of periods data

We can even extract a range of period data from the DataFrame using the period index.

ps['2015Q3':'2016']

```
In [38]: ps['2015Q3':'2016']

Out[38]: 2015Q3    0.194794
         2015Q4    0.922028
         2016Q1   -0.711151
         Freq: Q-DEC, dtype: float64
```

Here, we have extracted a range from 2015 Q3 to 2016 Q1.

19.6.5 Convert periods to datetime index

Sometimes the period index may be needed to convert to timestamp as that might be easy to proceed further. This is done by to_timestamp().

dti = ps.to_timestamp()

```
In [39]: dti = ps.to_timestamp()
         dti

Out[39]: 2015-01-01    0.297898
         2015-04-01    1.928422
         2015-07-01    0.194794
         2015-10-01    0.922028
         2016-01-01   -0.711151
         Freq: QS-OCT, dtype: float64
```

```
In [40]: dti.index

Out[40]: DatetimeIndex(['2015-01-01', '2015-04-01', '2015-07-01', '2015-10-01',
                        '2016-01-01'],
                       dtype='datetime64[ns]', freq='QS-OCT')
```

We can see that the period index has been converted to a DatetimeIndex.

19.6.6 Convert DatetimeIndex to PeriodIndex

Similar to the above case we might need the DatetimeIndex to be converted to a PeriodIndex as per the need of the analysis. This is done by to_period().

pi = dti.to_period()

```
In [41]: pi = dti.to_period()
         pi

Out[41]: 2015Q1    0.297898
         2015Q2    1.928422
         2015Q3    0.194794
         2015Q4    0.922028
         2016Q1   -0.711151
         Freq: Q-DEC, dtype: float64
```

```
In [42]: pi.index

Out[42]: PeriodIndex(['2015Q1', '2015Q2', '2015Q3', '2015Q4', '2016Q1'], dtype='period[Q-DEC]', freq='Q-DEC')
```

We can see that the DatetimeIndex that we created in the previous topic is now been converted to PeriodIndex.

19.7 Working with time zones

The time zone is an important feature of date time. There are two types of times one which is time zone aware and another which is time zone unaware (naïve time).

Let’s take some data to work with

```
In [1]: import pandas as pd
```

```
In [2]: df = pd.read_csv('timezone.csv',
                         index_col="date",
                         parse_dates=(['date']))
        df
```

Out[2]:

date	price
2020-10-03 01:00:00	57
2020-10-03 02:00:00	58
2020-10-03 03:00:00	59
2020-10-03 04:00:00	60
2020-10-03 05:00:00	61
2020-10-03 06:00:00	62
2020-10-03 07:00:00	63
2020-10-03 08:00:00	64
2020-10-03 09:00:00	65
2020-10-03 10:00:00	66

19.7.1 Make naïve time to time zone aware

With Pandas we can make a naïve time, a time zone aware time.
Let's look at an example

df1 = df.tz_localize(tz='US/Eastern')

```
In [3]: df1 = df.tz_localize(tz='US/Eastern')
        df1.index
```

```
Out[3]: DatetimeIndex(['2020-10-03 01:00:00-04:00', '2020-10-03 02:00:00-04:00',
                       '2020-10-03 03:00:00-04:00', '2020-10-03 04:00:00-04:00',
                       '2020-10-03 05:00:00-04:00', '2020-10-03 06:00:00-04:00',
                       '2020-10-03 07:00:00-04:00', '2020-10-03 08:00:00-04:00',
                       '2020-10-03 09:00:00-04:00', '2020-10-03 10:00:00-04:00'],
                      dtype='datetime64[ns, US/Eastern]', name='date', freq=None)
```

With the *tz_localize* operation we can convert a naïve time to time zone aware time. We have to pass *tz* as to which time zone the output is needed.

19.7.2 Available timezones

We may think, how to get the names of the time zones that has to be passed to the '*tz*' parameter above.
We can use *pytz* module of Python to get the list of all the common time

zones.

import pytz
pytz.all_timezones

```
In [4]: import pytz
        pytz.all_timezones[::30]

Out[4]: ['Africa/Abidjan',
         'Africa/Kinshasa',
         'America/Argentina/Catamarca',
         'America/Catamarca',
         'America/Guatemala',
         'America/Managua',
         'America/Port-au-Prince',
         'America/Thunder_Bay',
         'Asia/Ashkhabad',
         'Asia/Istanbul',
         'Asia/Qatar',
         'Asia/Yekaterinburg',
         'Australia/Queensland',
         'Etc/GMT+0',
         'Etc/UCT',
         'Europe/London',
         'Europe/Vienna',
         'Jamaica',
         'Pacific/Honolulu',
         'ROC']
```

We can obtain the time zone for specific country also:
Like for USA:
pytz.country_timezones('US')

For INDIA:
pytz.country_timezones('IN')

```
In [5]: pytz.country_timezones('IN')

Out[5]: ['Asia/Kolkata']
```

19.7.3 Convert on time zone to other

We can convert one time zone to another time zone by the help of *tz_convert* operation.

df = df1.tz_convert(tz="Europe/Guernsey")

here, df1 was in ‘US/Eastern’ time zone and is getting converted to

'*Europe/Guernsey*' time zone.

```
In [6]: df = df1.tz_convert(tz="Europe/Guernsey")
        df.index

Out[6]: DatetimeIndex(['2020-10-03 06:00:00+01:00', '2020-10-03 07:00:00+01:00',
                       '2020-10-03 08:00:00+01:00', '2020-10-03 09:00:00+01:00',
                       '2020-10-03 10:00:00+01:00', '2020-10-03 11:00:00+01:00',
                       '2020-10-03 12:00:00+01:00', '2020-10-03 13:00:00+01:00',
                       '2020-10-03 14:00:00+01:00', '2020-10-03 15:00:00+01:00'],
                      dtype='datetime64[ns, Europe/Guernsey]', name='date', freq=None)
```

19.7.4 Time zone in a date range

With date_range also we can pass the time zone, by default the date_range provides a naïve time but we can pass a *'tz'* parameter to specify the time zone.

rng = pd.date_range(start="10/3/2020", periods=10, freq='H', tz="Asia/Kolkata")

```
In [7]: rng = pd.date_range(start="10/3/2020", periods=10, freq='H', tz="Asia/Kolkata")
        rng

Out[7]: DatetimeIndex(['2020-10-03 00:00:00+05:30', '2020-10-03 01:00:00+05:30',
                       '2020-10-03 02:00:00+05:30', '2020-10-03 03:00:00+05:30',
                       '2020-10-03 04:00:00+05:30', '2020-10-03 05:00:00+05:30',
                       '2020-10-03 06:00:00+05:30', '2020-10-03 07:00:00+05:30',
                       '2020-10-03 08:00:00+05:30', '2020-10-03 09:00:00+05:30'],
                      dtype='datetime64[ns, Asia/Kolkata]', freq='H')
```

The above date range provides a date range with '*Asia/Kolkata*' time zone.

19.7.5 Time zone with dateutil

We can also provide the time zone by using '*dateutil*'. We just has to pass dateutil as a prefix to the time zone.

rng = pd.date_range(start="10/3/2020", periods=10, freq='H', tz="dateutil/Asia/Kolkata")

dateutil/Asia/Kolkata uses the dateutil to get the time zone.

```
In [8]: rng = pd.date_range(start="10/3/2020", periods=10, freq='H', tz="dateutil/Asia/Kolkata")
        rng

Out[8]: DatetimeIndex(['2020-10-03 00:00:00+05:30', '2020-10-03 01:00:00+05:30',
                       '2020-10-03 02:00:00+05:30', '2020-10-03 03:00:00+05:30',
                       '2020-10-03 04:00:00+05:30', '2020-10-03 05:00:00+05:30',
                       '2020-10-03 06:00:00+05:30', '2020-10-03 07:00:00+05:30',
                       '2020-10-03 08:00:00+05:30', '2020-10-03 09:00:00+05:30'],
                      dtype='datetime64[ns, tzfile('/usr/share/zoneinfo/Asia/Kolkata')]', freq='H')
```

Unlike pytz, dateutil uses the OS time zones, so there isn't any fixed list available but for common zones the names are same as in pytz.

19.8 Data shifts in DataFrame

We may sometime need to shift a particular column data back or forward in time. We can do this using Pandas by shift().

Let's look at this with examples:

```
In [1]: import pandas as pd

In [2]: df = pd.read_csv('shifting.csv', index_col="date", parse_dates=(['date']))
        df
```

Out[2]:

date	price
2020-03-10	43
2020-03-11	54
2020-03-12	73
2020-03-13	85
2020-03-14	53
2020-03-15	74
2020-03-16	76
2020-03-17	44
2020-03-18	62
2020-03-19	84

The data above has a DatetimeIndex and corresponding price for each date.

19.8.1 Shifting the price down

We can shift the data down by:

df.shift()

```
In [3]: df.shift()
```

Out[3]:

date	price
2020-03-10	NaN
2020-03-11	43.0
2020-03-12	54.0
2020-03-13	73.0
2020-03-14	85.0
2020-03-15	53.0
2020-03-16	74.0
2020-03-17	76.0
2020-03-18	44.0
2020-03-19	62.0

by default the data shifts down by one row because of which we can see that the price for date 10 March, 2020 has been shifted to 11 march, 2020 and so on.

19.8.2 Shifting by multiple rows

To shift the data by multiple rows we have to pass a value to the shift() as:

df.shift(3)

```
In [4]: df.shift(3)
```

Out[4]:

date	price
2020-03-10	NaN
2020-03-11	NaN
2020-03-12	NaN
2020-03-13	43.0
2020-03-14	54.0
2020-03-15	73.0
2020-03-16	85.0
2020-03-17	53.0
2020-03-18	74.0
2020-03-19	76.0

Here, we can see that we have provided a value 3 to shift() which shifted the price down by 3 dates and the price for 10 is now the price for 13. The prices for 10, 11 and 12 have become NaN.

19.8.3 Reverse shifting

We can reverse the shift as per our need by providing a negative integer to the shift() (example -3).

df.shift(-3)

```
In [5]: df.shift(-3)
```

Out[5]:

date	price
2020-03-10	85.0
2020-03-11	53.0
2020-03-12	74.0
2020-03-13	76.0
2020-03-14	44.0
2020-03-15	62.0
2020-03-16	84.0
2020-03-17	NaN
2020-03-18	NaN
2020-03-19	NaN

Here we can see that the price have been shifted up and the price which was for 19 is now for 17 and so on.

19.8.4 Use of shift

Let’s look at a use case of using shift in real life.
We’ll be calculating the percentage return from last 3 days.

New column with last day price:
We can create a column with the shifted price as:

df['previous_day_prices'] = df.shift()

```
In [6]: df['previous_day_prices'] = df.shift()
        df
```

Out[6]:

date	price	previous_day_prices
2020-03-10	43	NaN
2020-03-11	54	43.0
2020-03-12	73	54.0
2020-03-13	85	73.0
2020-03-14	53	85.0
2020-03-15	74	53.0
2020-03-16	76	74.0
2020-03-17	44	76.0
2020-03-18	62	44.0
2020-03-19	84	62.0

New column with change in the price from last day:

df['changed_price'] = df['price'] - df['previous_day_prices']

```
In [7]: df['changed_price'] = df['price'] - df['previous_day_prices']
        df
```

Out[7]:

date	price	previous_day_prices	changed_price
2020-03-10	43	NaN	NaN
2020-03-11	54	43.0	11.0
2020-03-12	73	54.0	19.0
2020-03-13	85	73.0	12.0
2020-03-14	53	85.0	-32.0
2020-03-15	74	53.0	21.0
2020-03-16	76	74.0	2.0
2020-03-17	44	76.0	-32.0
2020-03-18	62	44.0	18.0
2020-03-19	84	62.0	22.0

Now let’s get the return for the last three days.

*df['raturn_3days'] = (df['price']-df['price'].shift(3))*100/df['price'].shift(3)*

```
In [8]: df['raturn_3days'] = (df['price']-df['price'].shift(3))*100/df['price'].shift(3)
        df
```

Out[8]:

date	price	previous_day_prices	changed_price	raturn_3days
2020-03-10	43	NaN	NaN	NaN
2020-03-11	54	43.0	11.0	NaN
2020-03-12	73	54.0	19.0	NaN
2020-03-13	85	73.0	12.0	97.674419
2020-03-14	53	85.0	-32.0	-1.851852
2020-03-15	74	53.0	21.0	1.369863
2020-03-16	76	74.0	2.0	-10.588235
2020-03-17	44	76.0	-32.0	-16.981132
2020-03-18	62	44.0	18.0	-16.216216
2020-03-19	84	62.0	22.0	10.526316

With this we calculated the three day percentage return.

19.8.5 DatetimeIndex shift

We can even shift the DatetimeIndex by tshift().

df.tshift()

```
In [10]: df.tshift()
```

Out[10]:

date	price
2020-03-11	43
2020-03-12	54
2020-03-13	73
2020-03-14	85
2020-03-15	53
2020-03-16	74
2020-03-17	76
2020-03-18	44
2020-03-19	62
2020-03-20	84

Here we can see that the index has been shifted by one row down and day 10 has gone and taken over by 11. If we do not provide any parameter then the shifting will be done by one row (down).

19.8.6 Reverse DatetimeIndex shift

We can do the reverse index shifting by providing a negative integer to the tshift().

df.tshift(-1)

```
In [11]: df.tshift(-1)
```

Out[11]:

date	price
2020-03-09	43
2020-03-10	54
2020-03-11	73
2020-03-12	85
2020-03-13	53
2020-03-14	74
2020-03-15	76
2020-03-16	44
2020-03-17	62
2020-03-18	84

We can see that the index has been shifted a row up with the entry of 9 and vanished 19 from the index.

This is how time series works in Pandas. Let’s look at how to interact with various databases in the next chapter.

20 Database

Ultimately, any developer has to deal with database at some point of time. Handling database could be sometime tedious job. Pandas provides the facility to work with databases too and convert the database data to a dataframe. Once any data is in the form of dataframe then we know how to deal with them by now.
Let's look at two majorly used types of databases i.e. sql (MySQL) and no-sql (Mongo).

20.1 Working with MySQL

MySQL is one of the most famous databases used in the market. It's an open source, RDBMS (i.e. Relational Database Management System). let's see how we can work with this with some simple examples.

Let's create a table and insert some values to it assuming that's the data present in our database.

Create Database

CREATE DATABASE students_db

Create Table

```
create table students(
    id INT NOT NULL AUTO_INCREMENT,
    student_name VARCHAR(100) NOT NULL,
    age INT(10) NOT NULL,
    PRIMARY KEY ( id )
);
```

Insert Values

```
INSERT INTO students (student_name, age)
VALUES
("arun", 29),
("varun", 20),
("neha", 22),
("nisha", 35);
```

Now, we have a database as "students_db", a table as "students" and some

data into it.
Let’s proceed with working with the data bases.

20.1.1 Installations

We may need some packages to be installed before proceeding.

- Pandas: as we are working with pandas
- sqlalchemy (pip3 install sqlalchemy)
- PyMySQL (pip3 install PyMySQL)

There are the packages needed to connect to the database.

```
In [1]: import pandas as pd
        import sqlalchemy
```

20.1.2 Create connection

Now once we have installed and imported the packages, we need to connect to the database. The general syntax to create the connection is:

engine = sqlalchemy.create_engine('mysql+pymysql://<username>:<password>@<host>:<port>/<databaseName>)

for our example this is:

*engine = sqlalchemy.create_engine('mysql+pymysql://root:@localhost:3306/students_*

```
In [2]: engine = sqlalchemy.create_engine('mysql+pymysql://root:@localhost:3306/students_db')
        engine
Out[2]: Engine(mysql+pymysql://root:***@localhost:3306/students_db)
```

I don’t have any password to my MySQL database and thus the password field is blank

20.1.3 Read table data

Once the connect is established, we can now read the data in database.

df_students = pd.read_sql_table("students", engine)

```
In [3]: df_students = pd.read_sql_table("students", engine)
        df_students
```

Out[3]:

	id	student_name	age
0	1	arun	29
1	2	varun	20
2	3	neha	22
3	4	nisha	35

We can see that we are getting the data that we have inserted into database. This is done by “read_sql_table()”. This method takes argument as the table name and the connection and reads the table data.

20.1.4 Fetching specific columns from table

Many times, your table might have multiple columns that may not be needed to you. In such cases we can even fetch the specific columns that we need. This is done by a parameter ‘columns’ in read_sql_table().

df_students = pd.read_sql_table("students", engine, columns= ["student_name"])

```
In [4]: df_students = pd.read_sql_table("students",
                                       engine,
                                       columns=["student_name"])
        df_students
```

Out[4]:

	student_name
0	arun
1	varun
2	neha
3	nisha

We can see that we specified only “student_name” and thus the resultant dataframe does not contain the “id” or the “age” column.

20.1.5 Execute a query

It’s not just limited to reading the table data, we can even execute queries. Sometime a single table might not fulfil our need and we may have to join the

tables or any other type of query. This in pandas is done by read_sql_query(). This takes the query and the connection as basic parameters.

```
query='''
select student_name, age from students where student_name='arun'
'''
df_query = pd.read_sql_query(query, engine)
```

```
In [5]: query='''
        select student_name, age from students where student_name='arun'
        '''
        df_query = pd.read_sql_query(query, engine)
        df_query
```

Out[5]:

	student_name	age
0	arun	29

In addition, we can also specify ‘index_col’ if we need any specific index column and parse_dates if we have any dates related column (like date of birth) in the table.

20.1.6 Insert data to table

Database is not just about reading the data, we have to feed the data too sometimes. This is done with *“to_sql()”*.

Let’s take a dataframe with some students name and age.

```
students = {
    "robert":38,
    "michael":27,
    "gillian":52,
    "graeme":49,
    "rohan": 22,
}
df = pd.DataFrame(students.items(), columns=['name', 'Age'])
```

In [6]:
```
students = {
    "robert":38,
    "michael":27,
    "gillian":52,
    "graeme":49,
    "rohan": 22,
}
df = pd.DataFrame(students.items(), columns=['name', 'Age'])
df
```

Out[6]:

	name	Age
0	robert	38
1	michael	27
2	gillian	52
3	graeme	49
4	rohan	22

Before insert, we should make sure that the column name in the database should be matching with the column name in the dataframe. If this is not matching, we can change the column name in the dataframe.
In our case too, our database has column name as “students_name” and “age” whereas in dataframe the columns are “name” and “Age”. Thus, we have to rename the columns as:

```
df.rename(columns={
    'name':'student_name',
    'Age':'age'
}, inplace=True)
```

In [7]:
```
df.rename(columns={
    'name':'student_name',
    'Age':'age'
}, inplace=True)
df
```

Out[7]:

	student_name	age
0	robert	38
1	michael	27
2	gillian	52
3	graeme	49
4	rohan	22

Once the column names are matching then we can insert the data.

```
df.to_sql(
    name='students',
    con=engine,
    index=False,
    if_exists='append'
)
```

In [8]:
```
df.to_sql(
    name='students',
    con=engine,
    index=False,
    if_exists='append'
)
```

```
In [9]: df_students = pd.read_sql_table("students",
                                        engine,)
        df_students
```

Out[9]:

	id	student_name	age
0	1	arun	29
1	2	varun	20
2	3	neha	22
3	4	nisha	35
4	35	robert	38
5	36	michael	27
6	37	gillian	52
7	38	graeme	49
8	39	rohan	22

Here, have passed:

- Table name
- Connection name
- Index=False (as we don't need to pass the dataframe index to table as table has it's own index)
- if_exists='append' (as, if the table exists then the data should append to it).

20.1.7 Common function to read table and query

Pandas provides a common function which can work as both "read_sql_table" and "read_sql_query". It's a wrapper function which takes table name and works as "read_sql_table" or takes query and works as "read_sql_query". This function is "read_sql()".

20.1.7.1 read_sql() to fetch table data

df_students_table = pd.read_sql("students", engine,)

```
In [10]: df_students_table = pd.read_sql("students", engine,)
         df_students_table
```

Out[10]:

	id	student_name	age
0	1	arun	29
1	2	varun	20
2	3	neha	22
3	4	nisha	35
4	35	robert	38
5	36	michael	27
6	37	gillian	52
7	38	graeme	49
8	39	rohan	22

20.1.7.2 read_sql() to fetch query data

query='''
select student_name, age
from students
where student_name='arun'
'''
df_students_query = pd.read_sql(query, engine)

```
In [11]: query='''
         select student_name, age
         from students
         where student_name='arun'
         '''
         df_students_query = pd.read_sql(query, engine)
         df_students_query
```

Out[11]:

	student_name	age
0	arun	29

20.2 Working with MongoDB

MongoDB is a no-sql, document database. This is also open source. We can convert the result from MongoDB to a dataframe too.

20.2.1 Installations

We need to install “pymongo” to create the connection.

pip3 install pymongo

```
In [1]: import pymongo
        import pandas as pd
        from pymongo import MongoClient
```

20.2.2 Create connection

To create the connection, we need to know the database and the collection names. In our example the database name is “students_db” and the collections name is “students”.

client = MongoClient()
db = client.<database_name>
collection = db.<collection_name>

client = MongoClient()
db = client.students_db
collection = db.students

```
In [2]: client = MongoClient()
        db = client.students_db
        collection = db.students
```

20.2.3 Get records

Once the connection is established, we can use it for fetching the data as:

data = pd.DataFrame(list(collection.find()))

```
In [3]: data = pd.DataFrame(list(collection.find()))
        data
```

Out[3]:

	_id	student_name	age
0	5ed3a6a486565975a5e3bbb0	arun	29.0

20.2.4 Fetching specific columns

We can fetch the desired keys from the db by:

data = pd.DataFrame(list(collection.find({}, {"student_name":1})))

```
In [4]: data = pd.DataFrame(list(collection.find({}, {"student_name":1})))
        data
```

Out[4]:

	_id	student_name
0	5ed3a6a486565975a5e3bbb0	arun

We set the column (key) as 1, to fetch result corresponding to that key.

20.2.5 Insert records

We can insert the records too as:

students = [
{"student_name":"robert", "age":38},
{"student_name":"michael", "age":27},
{"student_name":"gillian", "age":52},
{"student_name":"graeme", "age":49},
{"student_name":"rohan", "age":22},
]
collection.insert_many(students)

```
In [5]: students = [
            {"student_name":"robert", "age":38},
            {"student_name":"michael", "age":27},
            {"student_name":"gillian", "age":52},
            {"student_name":"graeme", "age":49},
            {"student_name":"rohan", "age":22},
        ]
        collection.insert_many(students)
```

```
Out[5]: <pymongo.results.InsertManyResult at 0x117611780>
```

```
In [6]: data = pd.DataFrame(list(collection.find()))
        data
```

Out[6]:

	_id	student_name	age
0	5ed3a6a486565975a5e3bbb0	arun	29.0
1	5ed3b04c637f3a70ae4a20a3	robert	38.0
2	5ed3b04c637f3a70ae4a20a4	michael	27.0
3	5ed3b04c637f3a70ae4a20a5	gillian	52.0
4	5ed3b04c637f3a70ae4a20a6	graeme	49.0
5	5ed3b04c637f3a70ae4a20a7	rohan	22.0

We can see that the function insert_many() has inserted multiple records. We have insert_one() also to insert only one record.

20.2.6 Delete records

We can even delete the records from the database.

collection.delete_many({"student_name":{"$ne":"arun"}})

```
In [7]: collection.delete_many({"student_name":{"$ne":"arun"}})
        data = pd.DataFrame(list(collection.find()))
        data
```

Out[7]:

	_id	student_name	age
0	5ed3a6a486565975a5e3bbb0	arun	29.0

Here, we have deleted all the records whose student_name is not equal to “arun”.

21 About Author

Arun was born in Goa (India) and studied in multiple states across the country. Arun is a technical expert in problem solving and algorithm development. He has been exploring Artificial Intelligence since many years and has worked on multiple algorithms with main area of his expertise as Deep Learning. Arun has completed his Bachelor of Technology in Electronics and Communication Engineering and has found his interest in programming. Later worked with multiple organizations as developer in many technologies and languages.

Arun Has a great interest towards teaching and sharing his knowledge with friends, colleagues and interested once. This interest of him has motivated him to bend towards writing and sharing knowledge with society.