PROJECT REPROT

THAPAR INSTITUTE
OF ENGINEERING & TECHNOLOGY
(Deemed to be University)

Knowledge Solutions India

GROUP 43

Members:

- Devaangini Mehta
- Ritvik Sharma
- Manan Kapila

Table of content

Table of figures/graphs

ABSTRACT

The title of the project is “COVID-19 Analysis”. The main objective of the project is to analyze the COVID cases of each state. In this project, we have imported two files of different format i.e., csv and excel files. In csv file, we are provided with the COVID cases data on the daily basis of all the states. And in excel file, we are provided with state and union territories name and their latitude and longitude.

In this report we have explained Data Collection, Data Cleaning, Data Modelling and Data Visualization. To explain these parts, we have used multiple libraries of Python.

The libraries we used in this project are-Pandas, NumPy, fbprophet, plotly.graph_objects, folium, plotly.express, Matplotlib.pyplot, Warnings, Sklearn.metrics,

Further in this report we have made pie charts to analyze the cases. Also, we visualized the number of cases in India geographically which showed us the worst affected areas of India by COVID. This given data is from Jan to end of May.

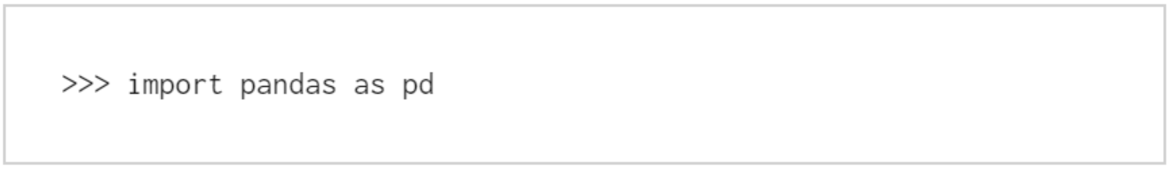
LIBRARIES USED IN DATA COLLECTION

- Pandas library
- Numpy library

PANDAS :-

Pandas is one of the most widely used python libraries in data science. It provides high-performance, easy to use structures and data analysis tools. Pandas provides in-memory 2d table object called Dataframe. It is like a spreadsheet with column names and row labels. Hence, with 2d tables, pandas is capable of providing many additional functionalities like creating pivot tables, computing columns based on other columns and plotting graphs.

Pandas can be imported into Python using:



```
>>> import pandas as pd
```

Some commonly used data structures in pandas are:-

- **Series objects**: 1D array, similar to a column in a spreadsheet
- **DataFrame objects:** 2D table, similar to a spreadsheet
- **Panel objects:** Dictionary of DataFrames, similar to sheet in MS Excel

Pandas Series object is created using pd.Series function. Each row is provided with an index and by defaults is assigned numerical values starting from 0.

Pandas dataframe object represents a spreadsheet with cell values, column names, and row index labels. Dataframe can be visualized as dictionaries of Series.

We can create pandas series object and pandas dataframe object as given below:

```
>>> people_dict = { "weight": pd.Series([68, 83, 112],index=
["alice", "bob", "charles"]),   "birthyear": pd.Series([1984,
1985, 1992], index=["bob", "alice", "charles"], name="year"),
"children": pd.Series([0, 3], index=["charles", "bob"]),
"hobby": pd.Series(["Biking", "Dancing"], index=["alice",
"bob"]),}
```

```
>>> people = pd.DataFrame(people_dict)
>>> people
```

Some other essential methods that are present in dataframes are:

- **head():** returns the top 5 rows in the dataframe object
- **tail():** returns the bottom 5 rows in the dataframe
- **info():** prints the summary of the dataframe
- **describe():** gives a nice overview of the main aggregated values over each column

NUMPY :-

NumPy stands for ‘Numerical Python’ or ‘Numeric Python’. It is an open source module of Python which provides fast mathematical computation on arrays and matrices. Since, arrays and matrices are an essential part of the Machine Learning ecosystem, NumPy along with Machine Learning modules like Scikit-learn, Pandas, Matplotlib, TensorFlow, etc. complete the Python Machine Learning Ecosystem.

NumPy provides the essential multi-dimensional array-oriented computing functionalities designed for high-level mathematical functions and scientific computation.

Numpy can be imported into Python using:

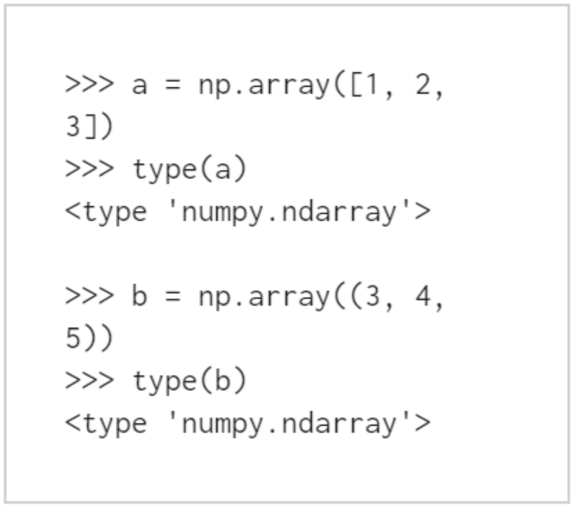
```
>>> import numpy as np
```

NumPy's main object is the homogeneous multidimensional array. It is a table with same type elements, i.e, integers or string or characters (homogeneous), usually integers. In NumPy, dimensions are called axes. The number of axes is called the rank.

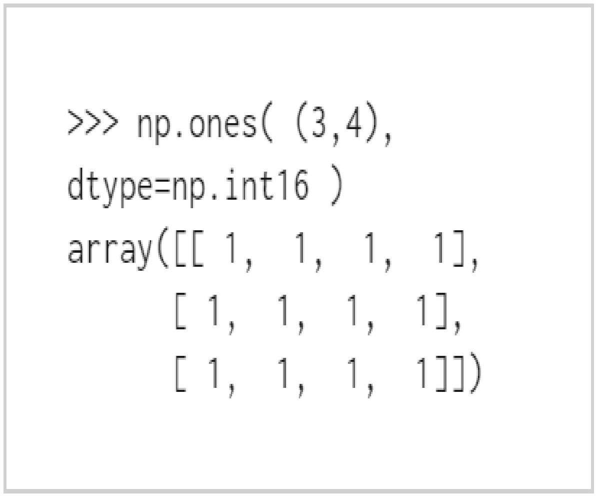
<u>There are several ways to create an array in NumPy like:-</u>

- np.array
- np.zeros
- np.ones
- np.arange
- np.linspace
- np.random.rand(2,3)
- np.empty((2,3))

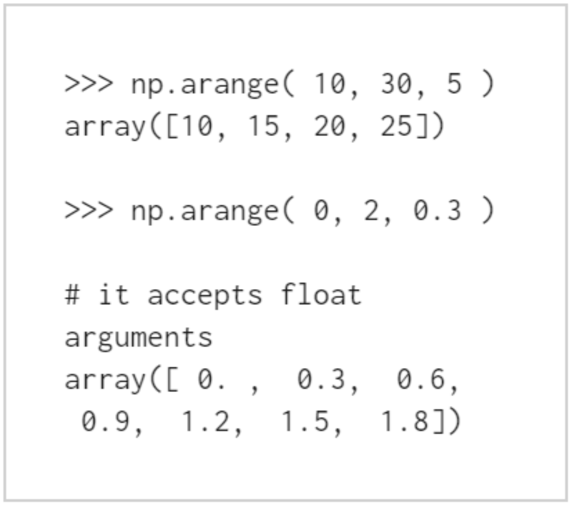
<u>Some examples to create an array in numpy:-</u>



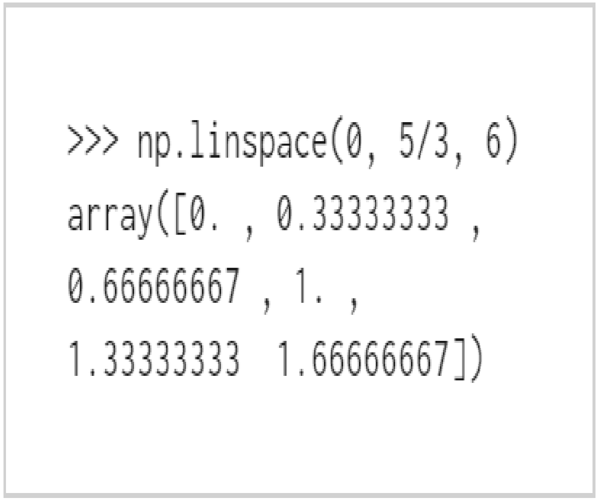
```
>>> a = np.array([1, 2,
3])
>>> type(a)
<type 'numpy.ndarray'>

>>> b = np.array((3, 4,
5))
>>> type(b)
<type 'numpy.ndarray'>
```



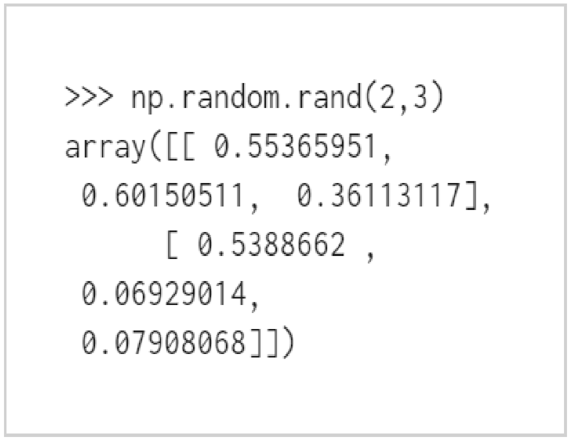
```
>>> np.ones( (3,4),
dtype=np.int16 )
array([[ 1,  1,  1,  1],
       [ 1,  1,  1,  1],
       [ 1,  1,  1,  1]])
```



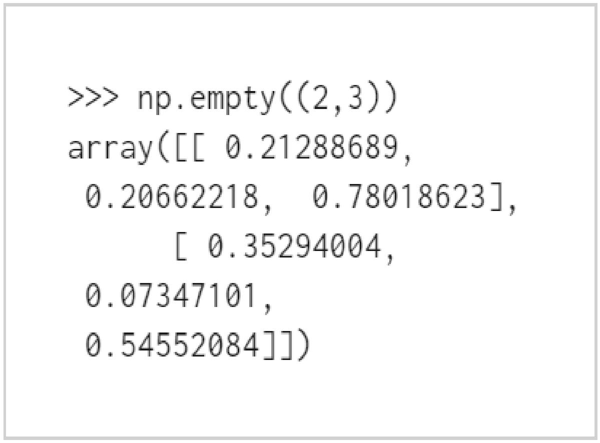
```
>>> np.arange( 10, 30, 5 )
array([10, 15, 20, 25])

>>> np.arange( 0, 2, 0.3 )

# it accepts float
arguments
array([ 0. ,  0.3,  0.6,
 0.9,  1.2,  1.5,  1.8])
```



```
>>> np.linspace(0, 5/3, 6)
array([0. , 0.33333333 ,
0.66666667 , 1. ,
1.33333333  1.66666667])
```



```
>>> np.random.rand(2,3)
array([[ 0.55365951,
 0.60150511,  0.36113117],
      [ 0.5388662 ,
 0.06929014,
 0.07908068]])
```



```
>>> np.empty((2,3))
array([[ 0.21288689,
 0.20662218,  0.78018623],
      [ 0.35294004,
 0.07347101,
 0.54552084]])
```

Some of the important attributes of a NumPy object are:

- **Ndim:** displays the dimension of the array
- **Shape:** returns a tuple of integers indicating the size of the array
- **Size:** returns the total number of elements in the NumPy array
- **Dtype**: returns the type of elements in the array, i.e., int64, character
- **Itemsize:** returns the size in bytes of each item
- **Reshape**: Reshapes the NumPy array

Plotly.express

Plotly.express is a new high-level Python visualization library: it's a wrapper for Plotly.py that exposes a simple syntax for complex charts. Inspired by Seaborn and ggplot2, it was specifically designed to have a terse, consistent and easy-to-learn API: with just a single import, you can make richly interactive plots in just a single function call, including faceting, maps, animations, and trendlines. It comes with on-board datasets, color scales and themes, and just like Plotly.py, Plotly Express is *totally free.*

Matplotlib.pyplot

Matplotlib.pyplot is a collection of command style functions that make matplotlib work like MATLAB. Each pyplot function makes some change to a figure: e.g., creates a figure, creates a plotting area in a figure, plots some lines in a plotting area, decorates the plot with labels, etc.

In matplotlib.pyplot various states are preserved across function calls, so that it keeps track of things like the current figure and plotting area, and the plotting functions are directed to the current axes (please note that "axes" here and in most places in the text refers to the *axes* part of a figure and not the strict mathematical term for more than one axis).

Warnings

The warnings module was introduced in PEP 230 as a way to warn programmers about changes in language or library features in anticipation of backwards incompatible changes coming with Python 3.0. Since warnings are not fatal, a program may encounter the same warn-able situation many times in the course of running. The warnings module suppresses repeated warnings from the same source to cut down on the annoyance of seeing the same message over and over. You can control the messages printed on a case-by-case basis using the *-W* option to the interpreter or by calling functions found in warnings from your code.

Sklearn.metrics

The module sklearn.metrics also exposes a set of simple functions measuring a prediction error given ground truth and prediction:

- functions ending with _score return a value to maximize, the higher the better.
- functions ending with _error or _loss return a value to minimize, the lower the better. When converting into a scorer object using make_scorer, set the greater_is_better parameter to False (True by default)

<u>plotly.graph_objects</u>

The figures created, manipulated and rendered by the plotly Python library are represented by tree-like data structures which are automatically serialized to JSON for rendering by the Plotly.js JavaScript library. These trees are composed of named nodes called "attributes", with their structure defined by the Plotly.js figure schema, which is available in machine-readable form.
The plotly.graph_objects module (typically imported as go) contains an automatically-generated hierarchy of Python classes which represent non-leaf nodes in this figure schema. The term "graph objects" refers to instances of these classes. The primary classes defined in the plotly.graph_objects module are Figure and an ipywidgets-compatible variant called FigureWidget, which both represent entire figures. Instances of these classes have many convenience methods for Pythonically manipulating their attributes (e.g. .update_layout() or .add_trace(), which all accept "magic underscore" notation) as well as rendering them (e.g. .to_json() or .write_image() or .write_html()).

The primary classes defined in the plotly.graph_objects module are Figure and an ipywidgets-compatible variant called FigureWidget, which both represent entire figures. Instances of these classes have many convenience methods for Pythonically manipulating their attributes (e.g. .update_layout() or .add_trace(), which all accept "magic underscore" notation) as well as rendering them (e.g. .show()) and exporting them to various formats (e.g. .to_json() or .write_image() or .write_html()).

The recommended way to create figures is using the functions in the **plotly.express** module, collectively known as Plotly Express, which all return instances of **plotly.graph_objects.Figure**, so every figure produced with the plotly library, actually uses graph objects under the hood, unless manually constructed out of dictionaries.
That said, certain kinds of figures are not yet possible to create with **Plotly Express**, such as figures that use certain 3D trace-types like mesh or iso surface. In addition, certain figures are cumbersome to create by starting from a figure created with Plotly Express, for example figures with subplots of different types, dual-axis plots, or faceted plots with multiple different types of traces. To construct such figures, it can be easier to start from an empty plotly.graph_objects.Figure object (or one configured with subplots via the make_subplots() function) and progressively add traces and update attributes.

Fbprophet

Implements a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well.

Prophet is an open source library published by Facebook that is based on **decomposable (trend+seasonality+holidays) models**. It provides us with the ability to make time series predictions with good accuracy using simple intuitive parameters and has support for including impact of custom seasonality and holidays!

We use a decomposable time series model with three main model components: trend, seasonality, and holidays. They are combined in the following equation:

$$y(t) = g(t) + s(t) + h(t) + \epsilon_t$$

- **g(t)**: piecewise linear or logistic growth curve for modelling non-periodic changes in time series
- **s(t)**: periodic changes (e.g. weekly/yearly seasonality)
- **h(t)**: effects of holidays (user provided) with irregular schedules
- **ε_t**: error term accounts for any unusual changes not accommodated by the model

Using time as a regressor, Prophet is trying to fit several linear and nonlinear functions of time as components. Modeling seasonality as an additive component is the same approach taken by exponential smoothing in Holt-Winters technique . We are, in effect, framing the forecasting problem as a curve-fitting exercise rather than looking explicitly at the time-based dependence of each observation within a time series. Trend is modelled by fitting a piece wise linear curve over the trend or the non-periodic part of the time series. The linear fitting exercise ensures that it is least affected by spikes/missing data.

folium

Folium is a powerful data visualisation library in Python that was built primarily to help people visualize geospatial data. With Folium, one can create a map of any location in the world as long as its latitude and longitude values are known. Also, the maps created by Folium are interactive in nature, so one can zoom in and out after the map is rendered, which is a super useful feature.

Folium builds on the data wrangling strengths of the Python ecosystem and the mapping strengths of the Leaflet.js library. The data is manipulated in Python and then visualised in a Leaflet map via folium. Folium provides the **folium.Map()** class which takes location parameter in terms of latitude and longitude and generates a map around it. These maps are interactive. We can zoom in and out by clicking the positive and negative buttons in the top-left corner of the map. We can also drag the map and see different regions.

Layers and Tiles in Folium: A tileset is a collection of raster and vector data broken up into a uniform grid of square tiles. Each tileset has a different way of representing data in the map. Folium allows us to create maps with different tiles like Stamen Terrain, Stamen Toner, Stamen Water Color, CartoDB Positron, and many more. By default, the tiles are set to OpenStreetMap.

sklearn.model_selection

Scikit-learn provides a range of supervised and unsupervised learning algorithms via a consistent interface in Python. sklearn.model_selection.train_test_split split arrays or matrices into random train and test subsets, the library is focused on modelling data.

Parameters

arrays** ***sequence of indexables with same length / shape[0]
Allowed inputs are lists, numpy arrays, scipy-sparse matrices or pandas dataframes.

test_size ***float or int, default=None***
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the test split. If int, represents the absolute number of test samples. If None, the value is set to the complement of the train size. If train_size is also None, it will be set to 0.25.

train_size ***float or int, default=None***
If float, should be between 0.0 and 1.0 and represent the proportion of the dataset to include in the train split. If int, represents the absolute number of train samples. If None, the value is automatically set to the complement of the test size.

random_state ***int or RandomState instance, default=None***
Controls the shuffling applied to the data before applying the split. Pass an int for reproducible output across multiple function calls.

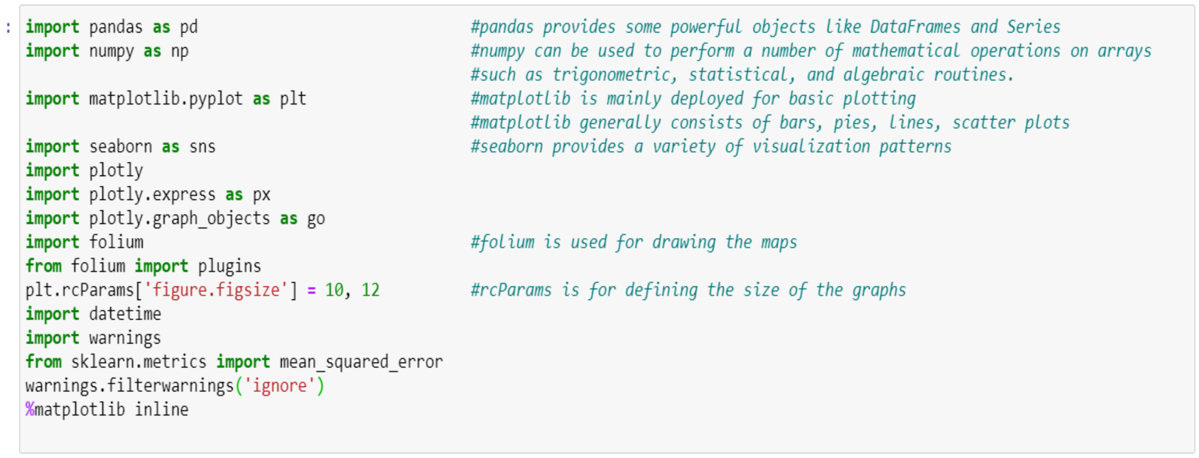
Shuffle ***bool, default=True***
Whether or not to shuffle the data before splitting. If shuffle=False then stratify must be None.

stratify***array-like, default=None***
If not None, data is split in a stratified fashion, using this as the class labels.

DATA COLLECTION

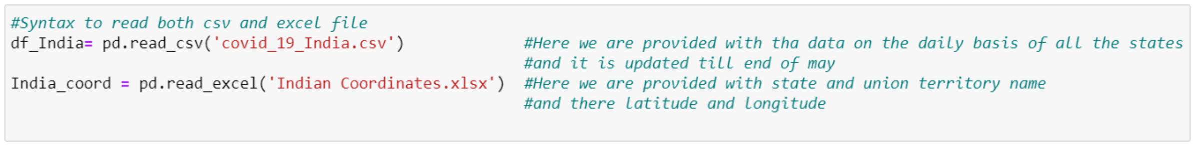
Data collection is the process of gathering and measuring information on variables of interest, in an established systematic fashion that enables one to answer stated research questions, test hypotheses, and evaluate outcomes.
The objective behind data collection is to capture quality evidence that allows analysis to lead to the formulation of convincing and credible answers to the questions that have been posed.

➢ Brief introduction of libraries that are used:-



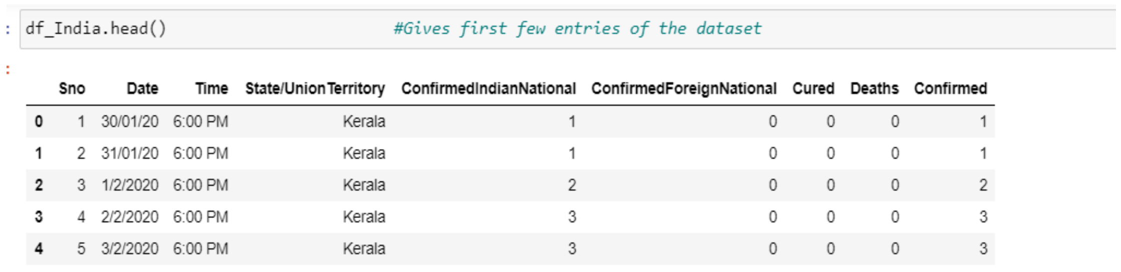
```
: import pandas as pd                          #pandas provides some powerful objects like DataFrames and Series
  import numpy as np                           #numpy can be used to perform a number of mathematical operations on arrays
                                               #such as trigonometric, statistical, and algebraic routines.
  import matplotlib.pyplot as plt              #matplotlib is mainly deployed for basic plotting
                                               #matplotlib generally consists of bars, pies, lines, scatter plots
  import seaborn as sns                        #seaborn provides a variety of visualization patterns
  import plotly
  import plotly.express as px
  import plotly.graph_objects as go
  import folium                                #folium is used for drawing the maps
  from folium import plugins
  plt.rcParams['figure.figsize'] = 10, 12      #rcParams is for defining the size of the graphs
  import datetime
  import warnings
  from sklearn.metrics import mean_squared_error
  warnings.filterwarnings('ignore')
  %matplotlib inline
```

➢ Ways to read files which are in different format:-



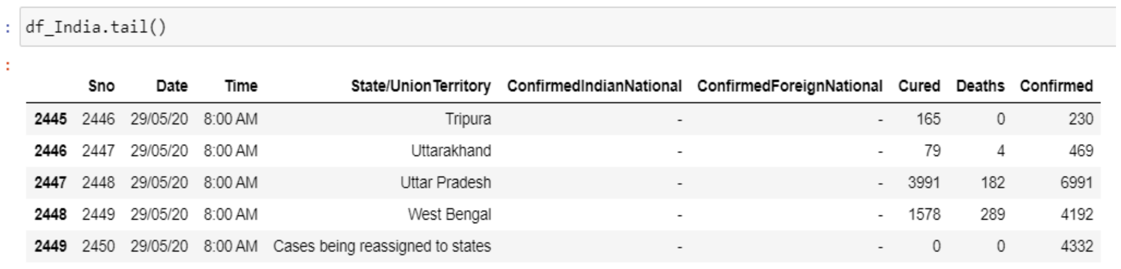
```
#Syntax to read both csv and excel file
df_India= pd.read_csv('covid_19_India.csv')              #Here we are provided with tha data on the daily basis of all the states
                                                         #and it is updated till end of may
India_coord = pd.read_excel('Indian Coordinates.xlsx')   #Here we are provided with state and union territory name
                                                         #and there latitude and longitude
```

➢ **Some ways to fetch data from data set:**



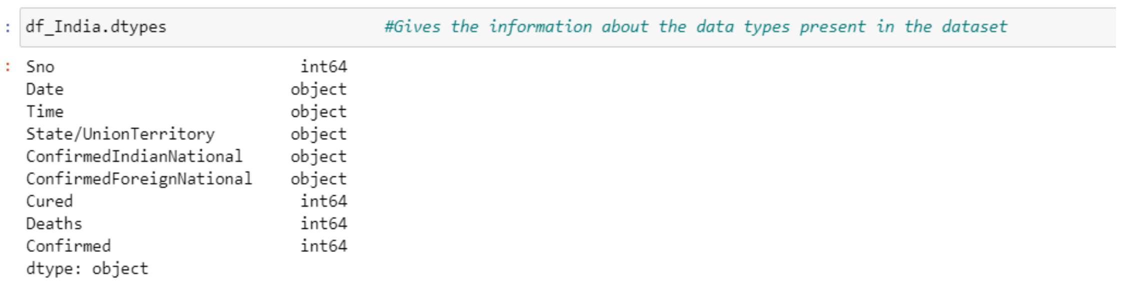
```
df_India.head()                    #Gives first few entries of the dataset
```

	Sno	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed
0	1	30/01/20	6:00 PM	Kerala	1	0	0	0	1
1	2	31/01/20	6:00 PM	Kerala	1	0	0	0	1
2	3	1/2/2020	6:00 PM	Kerala	2	0	0	0	2
3	4	2/2/2020	6:00 PM	Kerala	3	0	0	0	3
4	5	3/2/2020	6:00 PM	Kerala	3	0	0	0	3



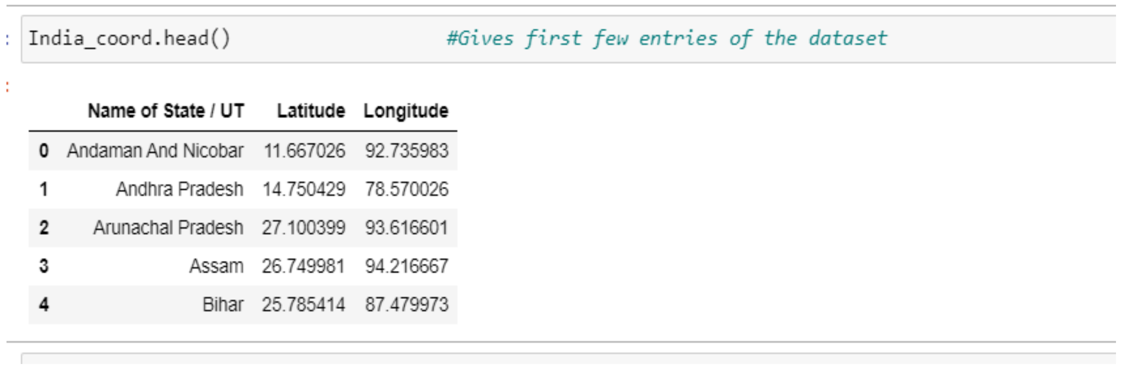
```
df_India.tail()
```

	Sno	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed
2445	2446	29/05/20	8:00 AM	Tripura	-	-	165	0	230
2446	2447	29/05/20	8:00 AM	Uttarakhand	-	-	79	4	469
2447	2448	29/05/20	8:00 AM	Uttar Pradesh	-	-	3991	182	6991
2448	2449	29/05/20	8:00 AM	West Bengal	-	-	1578	289	4192
2449	2450	29/05/20	8:00 AM	Cases being reassigned to states	-	-	0	0	4332



```
df_India.dtypes                    #Gives the information about the data types present in the dataset
```

```
Sno                          int64
Date                        object
Time                        object
State/UnionTerritory        object
ConfirmedIndianNational     object
ConfirmedForeignNational    object
Cured                        int64
Deaths                       int64
Confirmed                    int64
dtype: object
```



```
India_coord.head()                 #Gives first few entries of the dataset
```

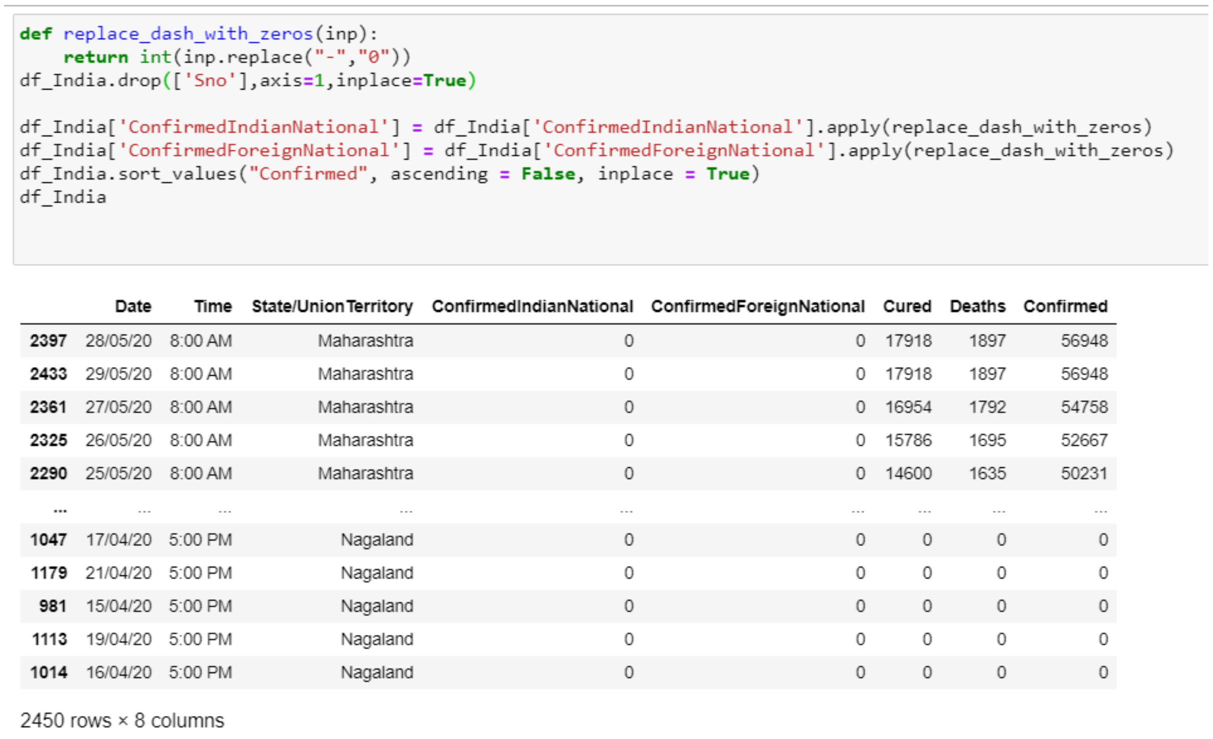
	Name of State / UT	Latitude	Longitude
0	Andaman And Nicobar	11.667026	92.735983
1	Andhra Pradesh	14.750429	78.570026
2	Arunachal Pradesh	27.100399	93.616601
3	Assam	26.749981	94.216667
4	Bihar	25.785414	87.479973

DATA CLEANING

Data cleaning or cleansing is the process of detecting and correcting (or removing) corrupt or inaccurate records from a record set, table, or database and refers to identifying incomplete, incorrect, inaccurate or irrelevant parts of the data and then replacing, modifying, or deleting the dirty or coarse data.

- Missing Data
- Irregular Data (Outliers)
- Unnecessary Data — Repetitive Data, Duplicates and more
- Inconsistent Data — Capitalization, Addresses and more

Here we have to replace (“-“) symbol with (“0”) in order to clean the data set.In data set they have calculated only total no. of cases without specifying confirmed_indian_national and confirmed_foreign_national. In there place they have put (“-“) sign in the columns. So, in order to clean the data write this code.

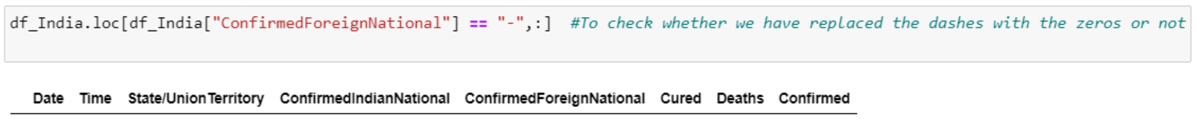


```
def replace_dash_with_zeros(inp):
    return int(inp.replace("-","0"))
df_India.drop(['Sno'],axis=1,inplace=True)

df_India['ConfirmedIndianNational'] = df_India['ConfirmedIndianNational'].apply(replace_dash_with_zeros)
df_India['ConfirmedForeignNational'] = df_India['ConfirmedForeignNational'].apply(replace_dash_with_zeros)
df_India.sort_values("Confirmed", ascending = False, inplace = True)
df_India
```

	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed
2397	28/05/20	8:00 AM	Maharashtra	0	0	17918	1897	56948
2433	29/05/20	8:00 AM	Maharashtra	0	0	17918	1897	56948
2361	27/05/20	8:00 AM	Maharashtra	0	0	16954	1792	54758
2325	26/05/20	8:00 AM	Maharashtra	0	0	15786	1695	52667
2290	25/05/20	8:00 AM	Maharashtra	0	0	14600	1635	50231
...	...	...	...	...	...	...	...	...
1047	17/04/20	5:00 PM	Nagaland	0	0	0	0	0
1179	21/04/20	5:00 PM	Nagaland	0	0	0	0	0
981	15/04/20	5:00 PM	Nagaland	0	0	0	0	0
1113	19/04/20	5:00 PM	Nagaland	0	0	0	0	0
1014	16/04/20	5:00 PM	Nagaland	0	0	0	0	0

2450 rows × 8 columns

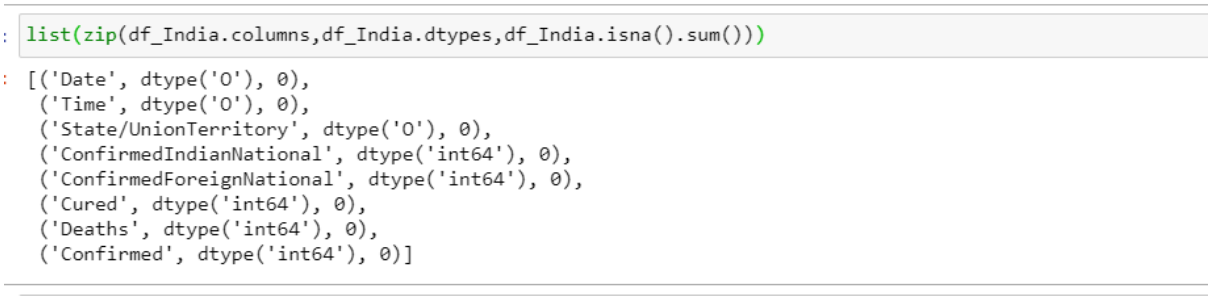
➢ To check whether code is working or not: -



```
df_India.loc[df_India["ConfirmedForeignNational"] == "-",:]  #To check whether we have replaced the dashes with the zeros or not
```

Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed

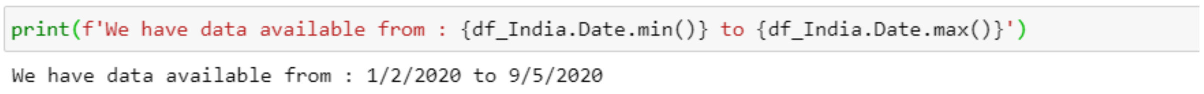
➢ To merge the data and to check whether there is null value or not:-



```
list(zip(df_India.columns,df_India.dtypes,df_India.isna().sum()))
```

```
[('Date', dtype('O'), 0),
 ('Time', dtype('O'), 0),
 ('State/UnionTerritory', dtype('O'), 0),
 ('ConfirmedIndianNational', dtype('int64'), 0),
 ('ConfirmedForeignNational', dtype('int64'), 0),
 ('Cured', dtype('int64'), 0),
 ('Deaths', dtype('int64'), 0),
 ('Confirmed', dtype('int64'), 0)]
```

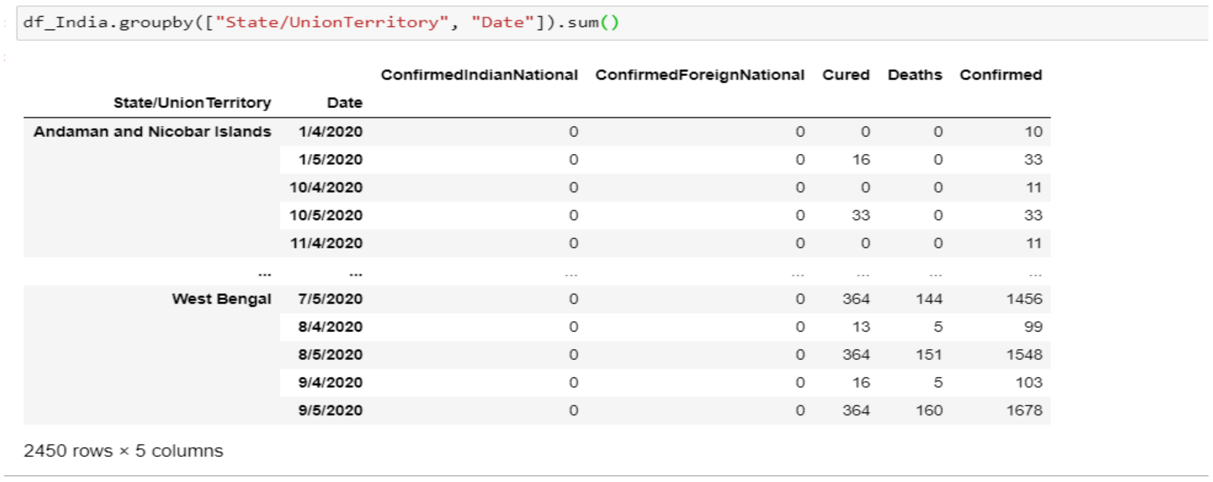
➢ To check the range of the data:-



```
print(f'We have data available from : {df_India.Date.min()} to {df_India.Date.max()}')
```

```
We have data available from : 1/2/2020 to 9/5/2020
```

➢ To see the data of different columns:

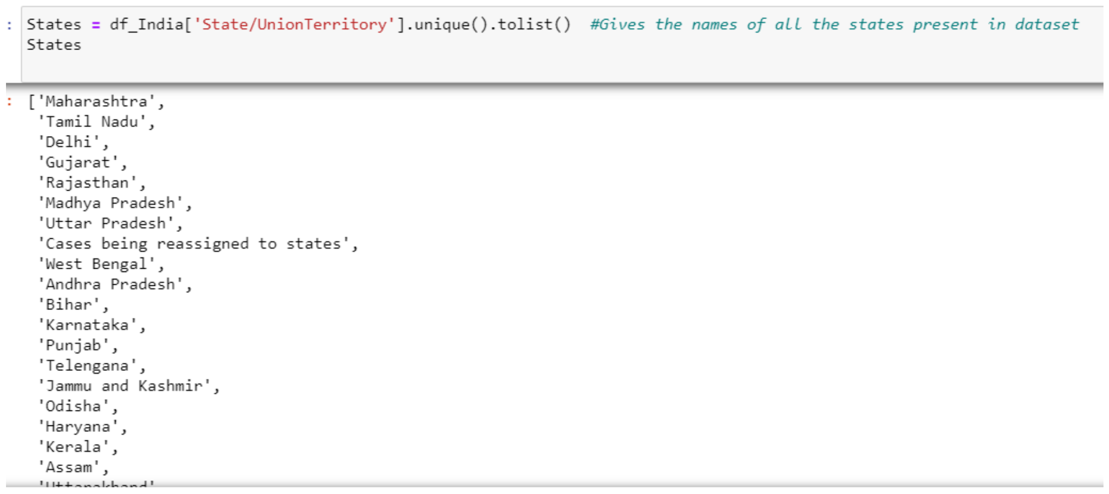


```
df_India.groupby(["State/UnionTerritory", "Date"]).sum()
```

		ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed
State/UnionTerritory	Date					
Andaman and Nicobar Islands	1/4/2020	0	0	0	0	10
	1/5/2020	0	0	16	0	33
	10/4/2020	0	0	0	0	11
	10/5/2020	0	0	33	0	33
	11/4/2020	0	0	0	0	11
...	...	...	...	...	...	...
West Bengal	7/5/2020	0	0	364	144	1456
	8/4/2020	0	0	13	5	99
	8/5/2020	0	0	364	151	1548
	9/4/2020	0	0	16	5	103
	9/5/2020	0	0	364	160	1678

2450 rows × 5 columns

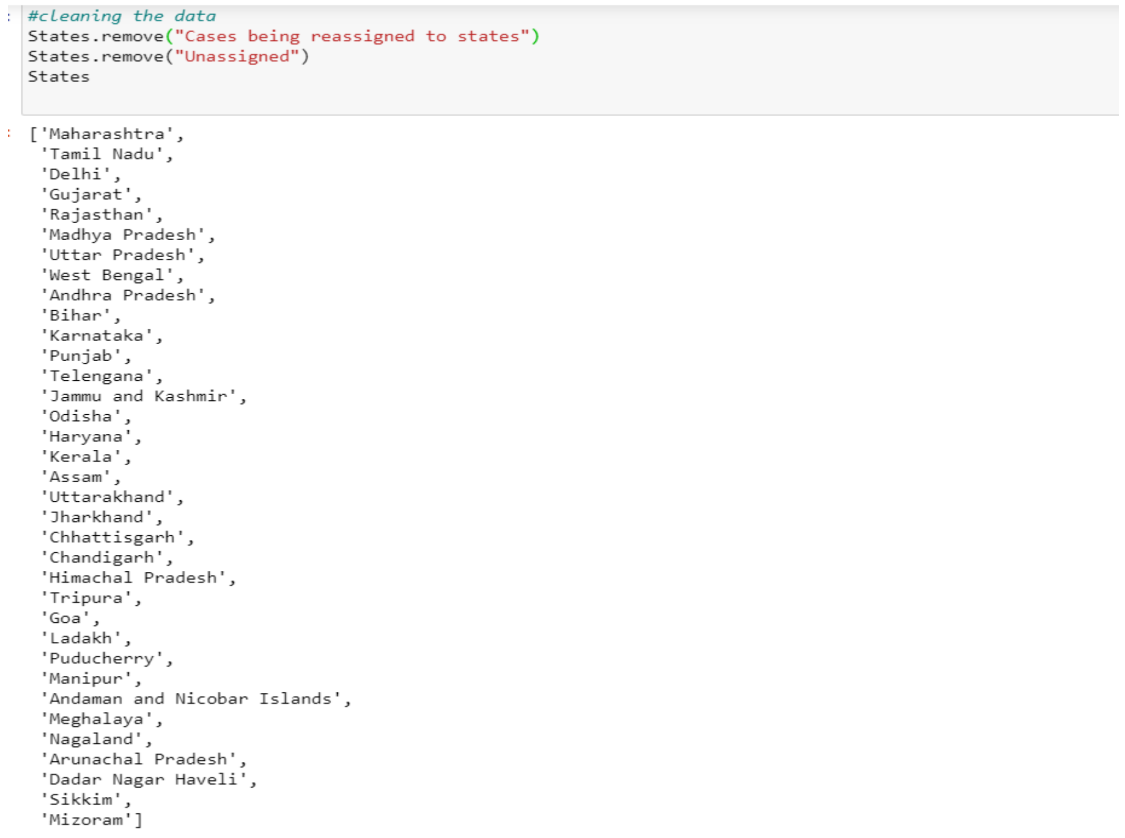
➢ To remove the Unnecessary data or duplicate from the data set:-



```
States = df_India['State/UnionTerritory'].unique().tolist()  #Gives the names of all the states present in dataset
States
```

```
['Maharashtra',
 'Tamil Nadu',
 'Delhi',
 'Gujarat',
 'Rajasthan',
 'Madhya Pradesh',
 'Uttar Pradesh',
 'Cases being reassigned to states',
 'West Bengal',
 'Andhra Pradesh',
 'Bihar',
 'Karnataka',
 'Punjab',
 'Telengana',
 'Jammu and Kashmir',
 'Odisha',
 'Haryana',
 'Kerala',
 'Assam',
 'Uttarakhand'
```

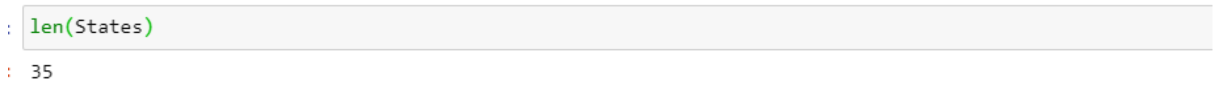
➢ To remove the useless data:



```
#cleaning the data
States.remove("Cases being reassigned to states")
States.remove("Unassigned")
States
```

```
['Maharashtra',
 'Tamil Nadu',
 'Delhi',
 'Gujarat',
 'Rajasthan',
 'Madhya Pradesh',
 'Uttar Pradesh',
 'West Bengal',
 'Andhra Pradesh',
 'Bihar',
 'Karnataka',
 'Punjab',
 'Telengana',
 'Jammu and Kashmir',
 'Odisha',
 'Haryana',
 'Kerala',
 'Assam',
 'Uttarakhand',
 'Jharkhand',
 'Chhattisgarh',
 'Chandigarh',
 'Himachal Pradesh',
 'Tripura',
 'Goa',
 'Ladakh',
 'Puducherry',
 'Manipur',
 'Andaman and Nicobar Islands',
 'Meghalaya',
 'Nagaland',
 'Arunachal Pradesh',
 'Dadar Nagar Haveli',
 'Sikkim',
 'Mizoram']
```

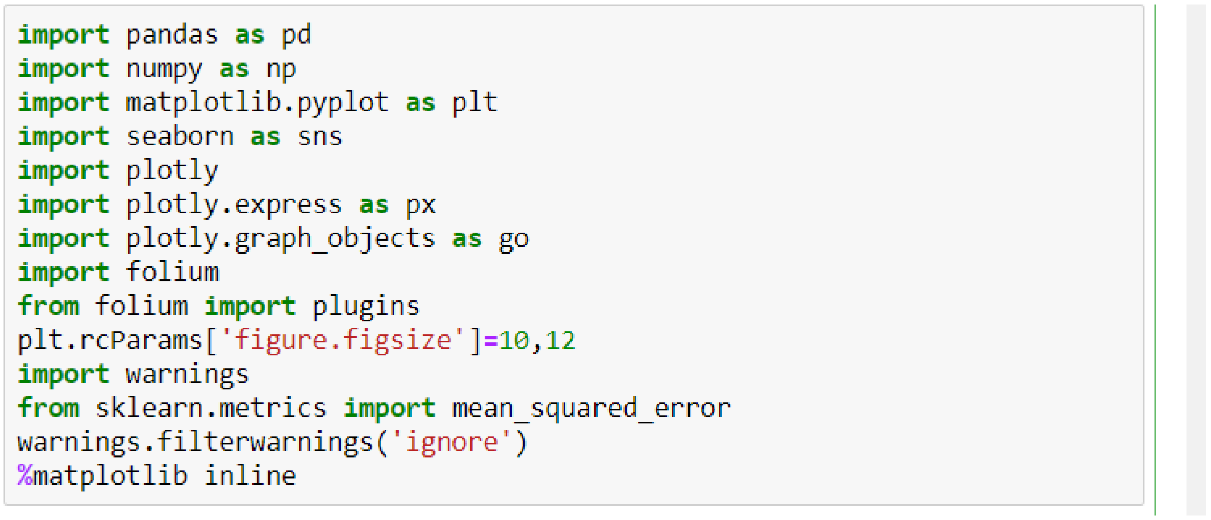
➢ To find the length of number of states:-



```
len(States)
```

```
35
```

DATA VISUALISATION



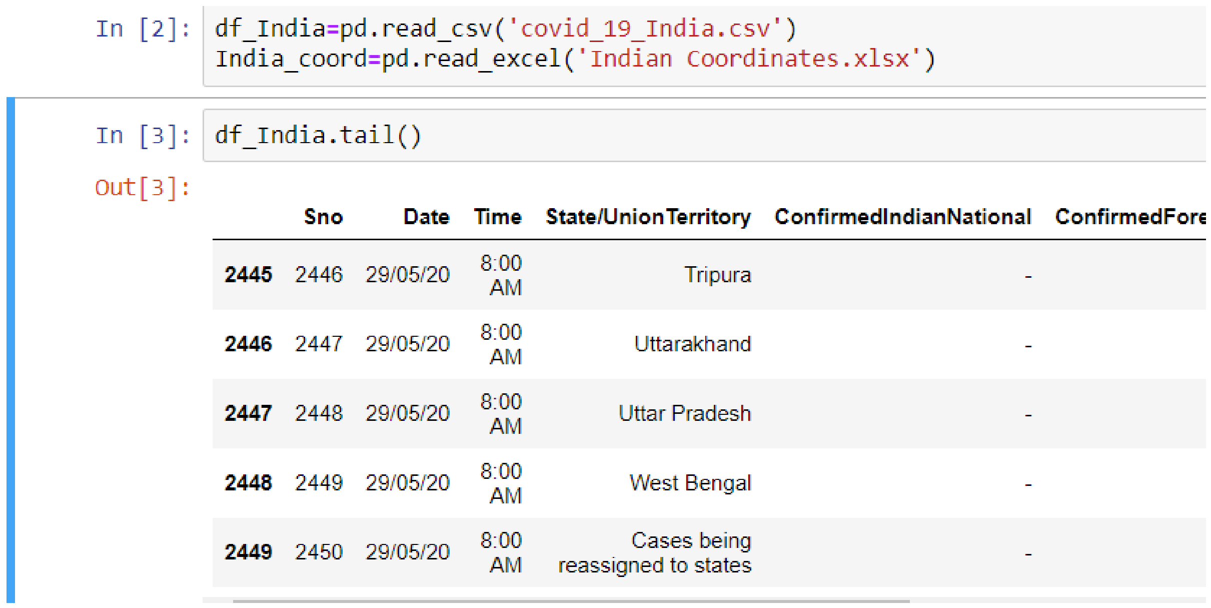
```
import pandas as pd
import numpy as np
import matplotlib.pyplot as plt
import seaborn as sns
import plotly
import plotly.express as px
import plotly.graph_objects as go
import folium
from folium import plugins
plt.rcParams['figure.figsize']=10,12
import warnings
from sklearn.metrics import mean_squared_error
warnings.filterwarnings('ignore')
%matplotlib inline
```

Plugins add to the ability of folium to make map

Plt.pcParams[‘figure.figsize’] is for setting figure size of the plot

Matplotlib inline for making the graph on same notebook

Folium library is used to make maps



```
In [2]: df_India=pd.read_csv('covid_19_India.csv')
        India_coord=pd.read_excel('Indian Coordinates.xlsx')
```

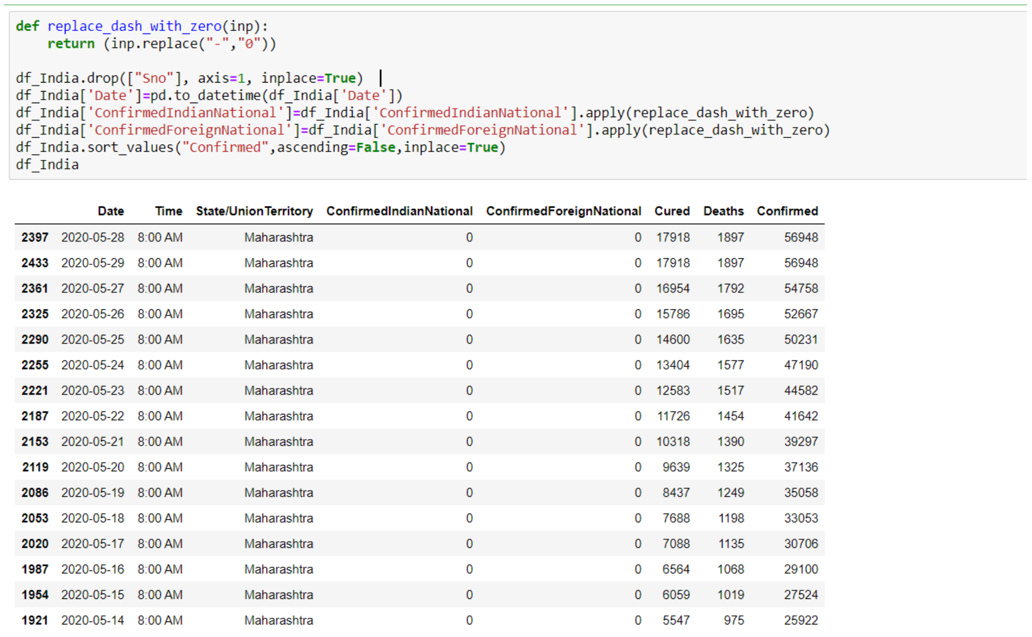
```
In [3]: df_India.tail()
```

Out[3]:

	Sno	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedFore
2445	2446	29/05/20	8:00 AM	Tripura	-	
2446	2447	29/05/20	8:00 AM	Uttarakhand	-	
2447	2448	29/05/20	8:00 AM	Uttar Pradesh	-	
2448	2449	29/05/20	8:00 AM	West Bengal	-	
2449	2450	29/05/20	8:00 AM	Cases being reassigned to states	-	

read_csv is used for importing the csv file and read_excel is used for importing excel file

df_India.tail() is used to print last few entries of the dataset

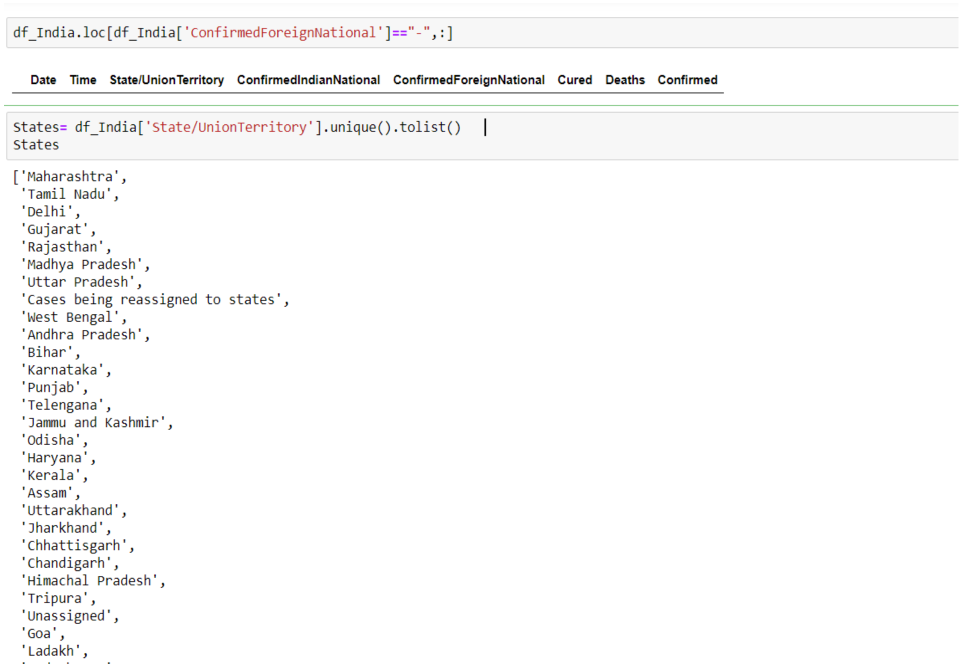


```python
def replace_dash_with_zero(inp):
    return (inp.replace("-","0"))

df_India.drop(["Sno"], axis=1, inplace=True)
df_India['Date']=pd.to_datetime(df_India['Date'])
df_India['ConfirmedIndianNational']=df_India['ConfirmedIndianNational'].apply(replace_dash_with_zero)
df_India['ConfirmedForeignNational']=df_India['ConfirmedForeignNational'].apply(replace_dash_with_zero)
df_India.sort_values("Confirmed",ascending=False,inplace=True)
df_India
```

	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed
2397	2020-05-28	8:00 AM	Maharashtra	0	0	17918	1897	56948
2433	2020-05-29	8:00 AM	Maharashtra	0	0	17918	1897	56948
2361	2020-05-27	8:00 AM	Maharashtra	0	0	16954	1792	54758
2325	2020-05-26	8:00 AM	Maharashtra	0	0	15786	1695	52667
2290	2020-05-25	8:00 AM	Maharashtra	0	0	14600	1635	50231
2255	2020-05-24	8:00 AM	Maharashtra	0	0	13404	1577	47190
2221	2020-05-23	8:00 AM	Maharashtra	0	0	12583	1517	44582
2187	2020-05-22	8:00 AM	Maharashtra	0	0	11726	1454	41642
2153	2020-05-21	8:00 AM	Maharashtra	0	0	10318	1390	39297
2119	2020-05-20	8:00 AM	Maharashtra	0	0	9639	1325	37136
2086	2020-05-19	8:00 AM	Maharashtra	0	0	8437	1249	35058
2053	2020-05-18	8:00 AM	Maharashtra	0	0	7688	1198	33053
2020	2020-05-17	8:00 AM	Maharashtra	0	0	7088	1135	30706
1987	2020-05-16	8:00 AM	Maharashtra	0	0	6564	1068	29100
1954	2020-05-15	8:00 AM	Maharashtra	0	0	6059	1019	27524
1921	2020-05-14	8:00 AM	Maharashtra	0	0	5547	975	25922

In the Function replace_dash_with_zero all the dashes present in the original dataset are being replaced by zero and are visible in the ConfirmedIndianNational and ConfirmedForeignNational columns of the output.

Drop is used to cut the column Sno in the original dataset , Inplace is used for sorting the data, In the next line we are converting date in date time format and then with the help of function replace_dash_with_zero we are replacing dashes with zeroes. Then we are sorting the values.



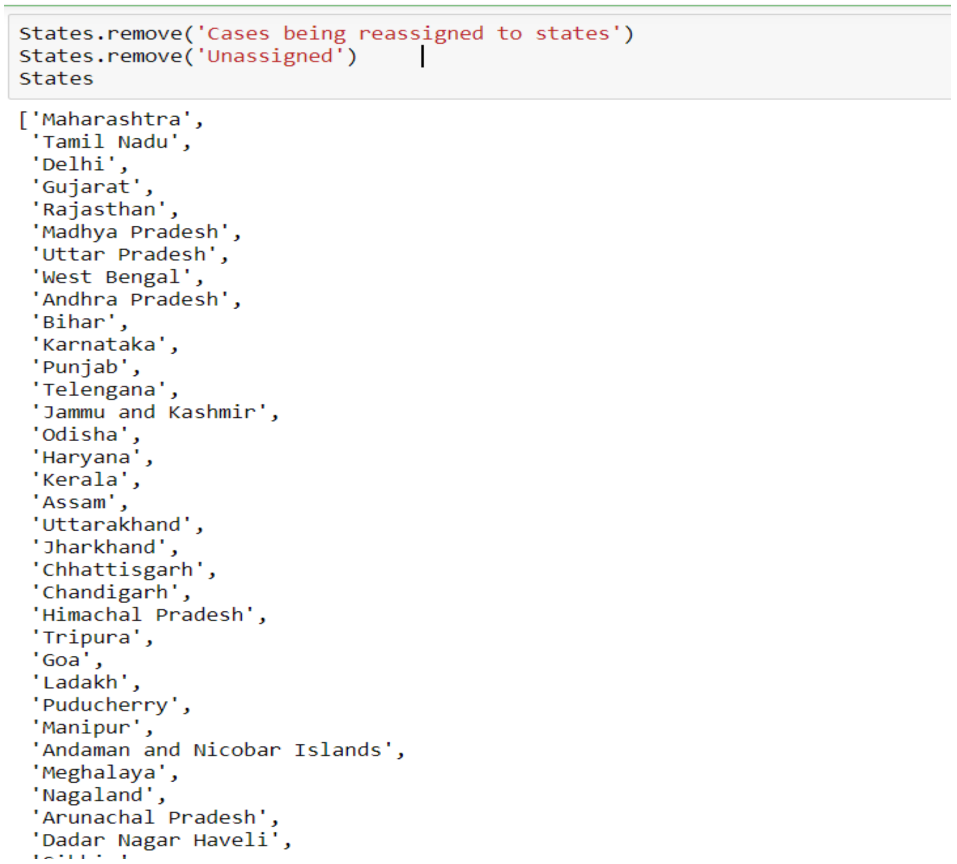
```python
df_India.loc[df_India['ConfirmedForeignNational']=="-",:]
```

Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed

```python
States= df_India['State/UnionTerritory'].unique().tolist()
States
```

```
['Maharashtra',
 'Tamil Nadu',
 'Delhi',
 'Gujarat',
 'Rajasthan',
 'Madhya Pradesh',
 'Uttar Pradesh',
 'Cases being reassigned to states',
 'West Bengal',
 'Andhra Pradesh',
 'Bihar',
 'Karnataka',
 'Punjab',
 'Telengana',
 'Jammu and Kashmir',
 'Odisha',
 'Haryana',
 'Kerala',
 'Assam',
 'Uttarakhand',
 'Jharkhand',
 'Chhattisgarh',
 'Chandigarh',
 'Himachal Pradesh',
 'Tripura',
 'Unassigned',
 'Goa',
 'Ladakh',
```

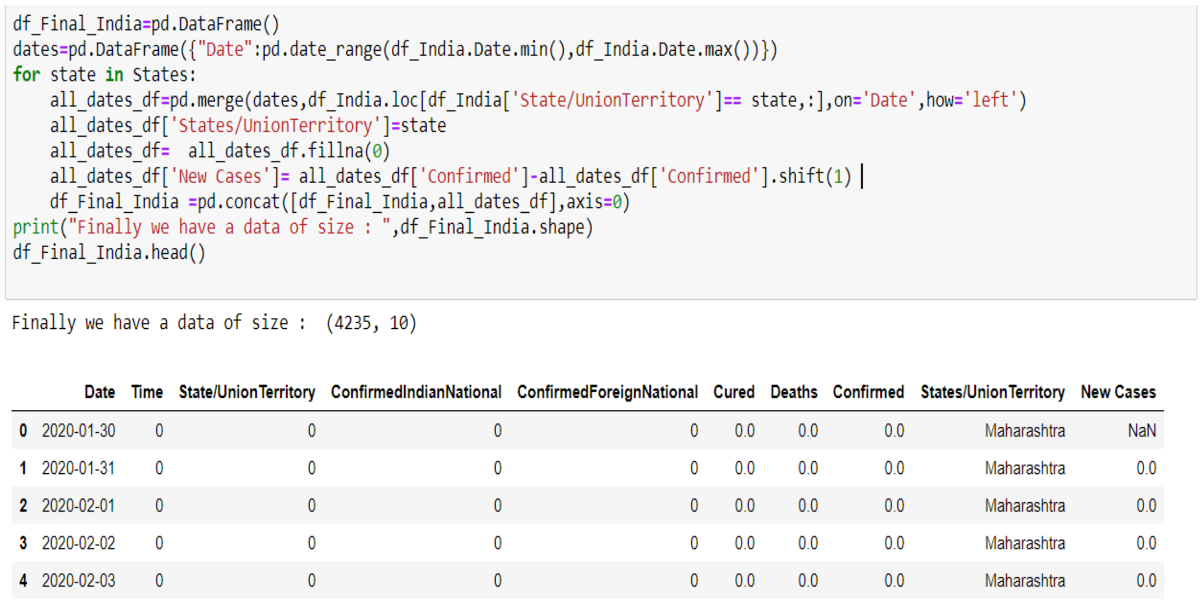
Line 5’s Output tells that we have successfully replaced all dashes with zeroes, code written in Line 6 is to make list of Unique entries in state column of Dataframe



```
States.remove('Cases being reassigned to states')
States.remove('Unassigned')
States
```

```
['Maharashtra',
 'Tamil Nadu',
 'Delhi',
 'Gujarat',
 'Rajasthan',
 'Madhya Pradesh',
 'Uttar Pradesh',
 'West Bengal',
 'Andhra Pradesh',
 'Bihar',
 'Karnataka',
 'Punjab',
 'Telengana',
 'Jammu and Kashmir',
 'Odisha',
 'Haryana',
 'Kerala',
 'Assam',
 'Uttarakhand',
 'Jharkhand',
 'Chhattisgarh',
 'Chandigarh',
 'Himachal Pradesh',
 'Tripura',
 'Goa',
 'Ladakh',
 'Puducherry',
 'Manipur',
 'Andaman and Nicobar Islands',
 'Meghalaya',
 'Nagaland',
 'Arunachal Pradesh',
 'Dadar Nagar Haveli',
```

Now removing the line ‘Cases being reassigned to states’ and ‘Unassigned’



```
df_Final_India=pd.DataFrame()
dates=pd.DataFrame({"Date":pd.date_range(df_India.Date.min(),df_India.Date.max())})
for state in States:
    all_dates_df=pd.merge(dates,df_India.loc[df_India['State/UnionTerritory']== state,:],on='Date',how='left')
    all_dates_df['States/UnionTerritory']=state
    all_dates_df=  all_dates_df.fillna(0)
    all_dates_df['New Cases']= all_dates_df['Confirmed']-all_dates_df['Confirmed'].shift(1)
    df_Final_India =pd.concat([df_Final_India,all_dates_df],axis=0)
print("Finally we have a data of size : ",df_Final_India.shape)
df_Final_India.head()
```

```
Finally we have a data of size :  (4235, 10)
```

	Date	Time	State/UnionTerritory	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed	States/UnionTerritory	New Cases
0	2020-01-30	0	0	0	0	0.0	0.0	0.0	Maharashtra	NaN
1	2020-01-31	0	0	0	0	0.0	0.0	0.0	Maharashtra	0.0
2	2020-02-01	0	0	0	0	0.0	0.0	0.0	Maharashtra	0.0
3	2020-02-02	0	0	0	0	0.0	0.0	0.0	Maharashtra	0.0
4	2020-02-03	0	0	0	0	0.0	0.0	0.0	Maharashtra	0.0

In the first line we are using an Empty Data Set

In the second line we are Passing the min date and max date available in original dataset

In the For Loop

First line-> Here we have written what we are going to merge ,how left means we are going to merge left column with the right

Second line -> All states are being assigned in the empty dataset

Third line-> If no dates are there then fill it with zero

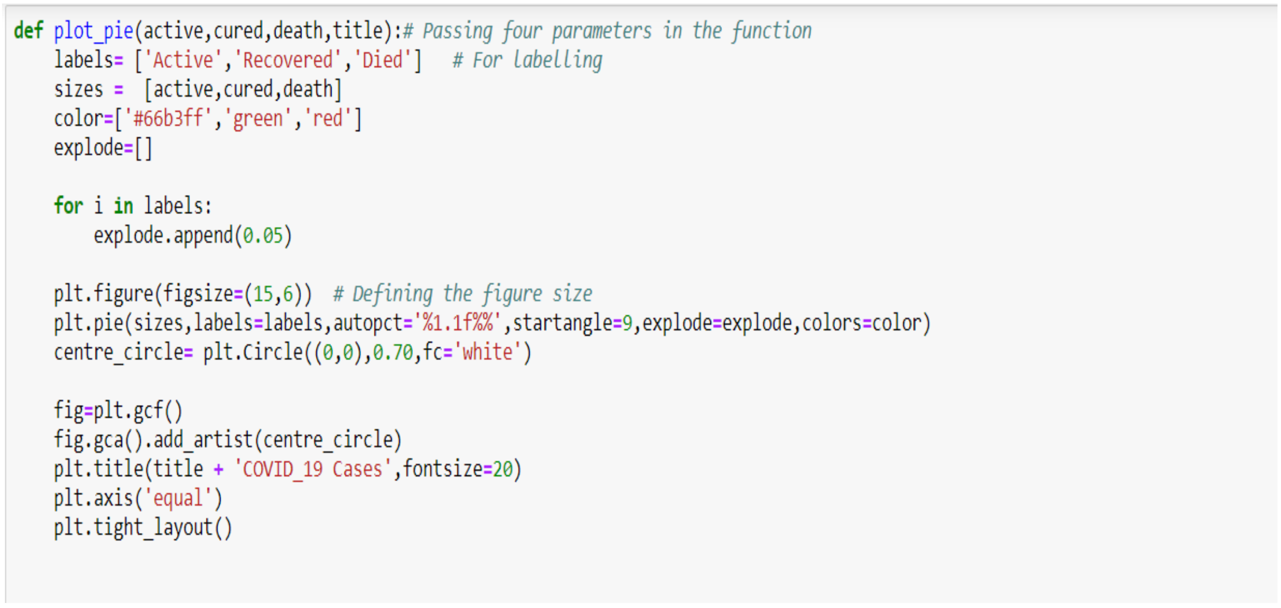
Fourth Line-> Creating a new column New cases and calculating it using The formula

(present day cases- yesterday cases)

Fifth Line-> Concatinating (combing) the df_Final_India and all_dates_df

Outside the For loop

Printing number of rows and columns using shape() command, Printing first few lines of df_Final_India using head() command



```python
def plot_pie(active,cured,death,title):# Passing four parameters in the function
    labels= ['Active','Recovered','Died']   # For labelling
    sizes =  [active,cured,death]
    color=['#66b3ff','green','red']
    explode=[]

    for i in labels:
        explode.append(0.05)

    plt.figure(figsize=(15,6))  # Defining the figure size
    plt.pie(sizes,labels=labels,autopct='%1.1f%%',startangle=9,explode=explode,colors=color)
    centre_circle= plt.Circle((0,0),0.70,fc='white')

    fig=plt.gcf()
    fig.gca().add_artist(centre_circle)
    plt.title(title + 'COVID_19 Cases',fontsize=20)
    plt.axis('equal')
    plt.tight_layout()
```

In this code we are making a function for plotting a pie chart with three labels :’Active’, ’Recovered’, ’Died’.

Explode is used for giving spaces in between the labels.

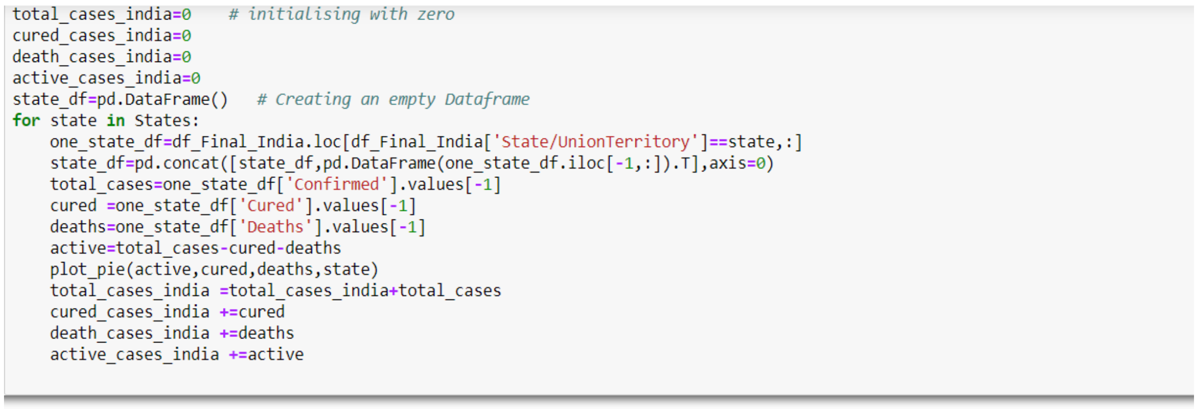
Gcf() is used to get the current figure.

Plt.title is used to give title to the pie chart.

Making the axis equal by using plt.axis()

Tight layout automatically adjusts subplot params so that the subplots(s) fits in to the figure area

STATEWISE COVID DATA IN INDIA



```
total_cases_india=0    # initialising with zero
cured_cases_india=0
death_cases_india=0
active_cases_india=0
state_df=pd.DataFrame()   # Creating an empty Dataframe
for state in States:
    one_state_df=df_Final_India.loc[df_Final_India['State/UnionTerritory']==state,:]
    state_df=pd.concat([state_df,pd.DataFrame(one_state_df.iloc[-1,:]).T],axis=0)
    total_cases=one_state_df['Confirmed'].values[-1]
    cured =one_state_df['Cured'].values[-1]
    deaths=one_state_df['Deaths'].values[-1]
    active=total_cases-cured-deaths
    plot_pie(active,cured,deaths,state)
    total_cases_india =total_cases_india+total_cases
    cured_cases_india +=cured
    death_cases_india +=deaths
    active_cases_india +=active
```

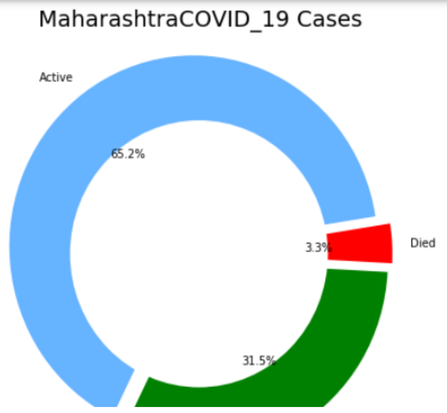


Figure 1.1

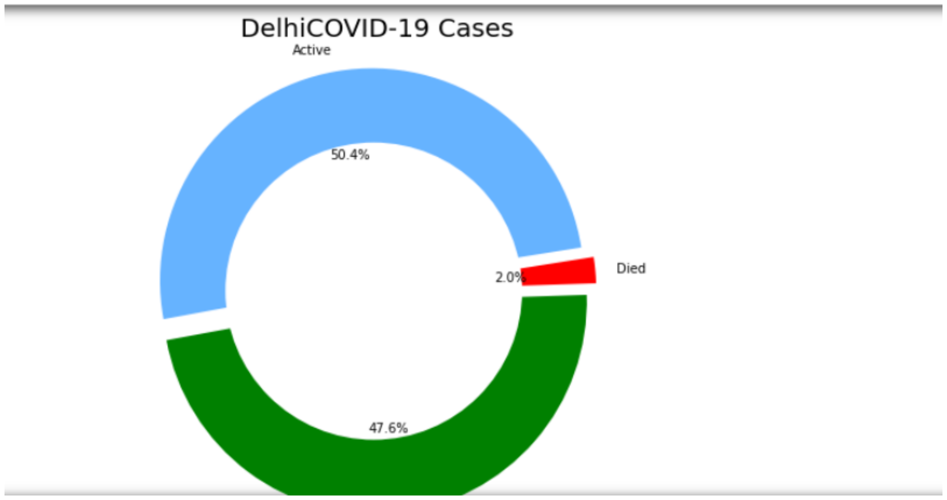


Figure 1.2

Here we are creating pie chart for every state in the dataset using For loop.

iloc[-1,:] means from the last value . T written here is a part of syntax

i

```
f ,ax =plt.subplots(figsize=(12,28))
data= state_df[['State/UnionTerritory','Confirmed','Cured','Deaths']]
data.sort_values('Confirmed',ascending=False,inplace=True)
sns.set_color_codes('pastel')
sns.barplot(x='Confirmed',y='State/UnionTerritory',data=data,label='Total',color='red')
sns.set_color_codes('muted')
sns.barplot(x='Cured',y='State/UnionTerritory',data=data,label='Cured',color='green')
ax.legend(ncol=5,loc='lower right',frameon=True)
ax.set(ylabel="",xlabel='Cases')
i=0
for p in ax.patches:
       x=p.get_x() +p.get_width()+ 3
       y=p.get_y() +p.get_height()/2
       if i<=len(States):
            ax.annotate("  "*10 + str(int(p.get_width())),(x,y))
       else:
           ax.annotate(int(p.get_width()),(x,y))
       i+=1
```

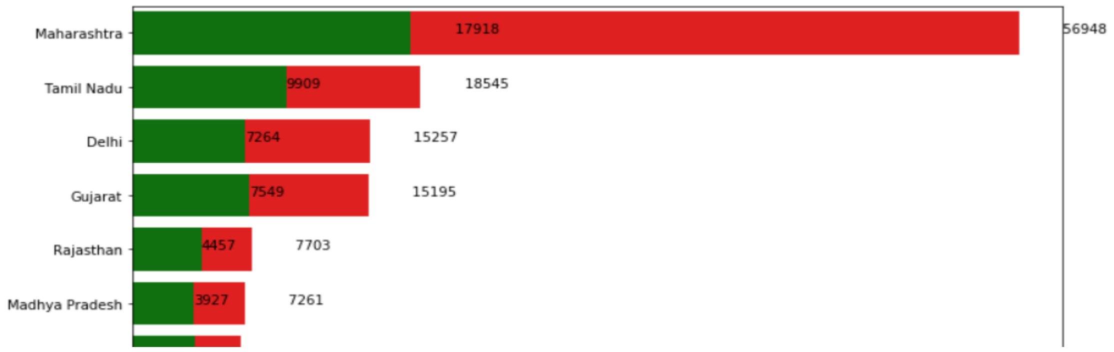


Figure 1.3

In the code we are setting figure size using figsize() command

Sorting values using sort_values()

Color code is used is pastel

In the next line we are creating a bar plot of Confirmed cases.

Then we are creating a bar plot of Cured Cases and are placing this bar plot above the previous barplot

OVERALL COVID19 STATUS IN INDIA

```
print("Total infected cases in India: ", total_cases_india)
print("Total cured cases in India: ", cured_cases_india)
print("Total active cases in India: ", active_cases_india)
print("Total death cases in India: ", death_cases_india)
plot_pie(active_cases_india, cured_cases_india, death_cases_india, "India")
```

```
Total infected cases in India:  154001.0
Total cured cases in India:  67692.0
Total active cases in India:  81778.0
Total death cases in India:  4531.0
```

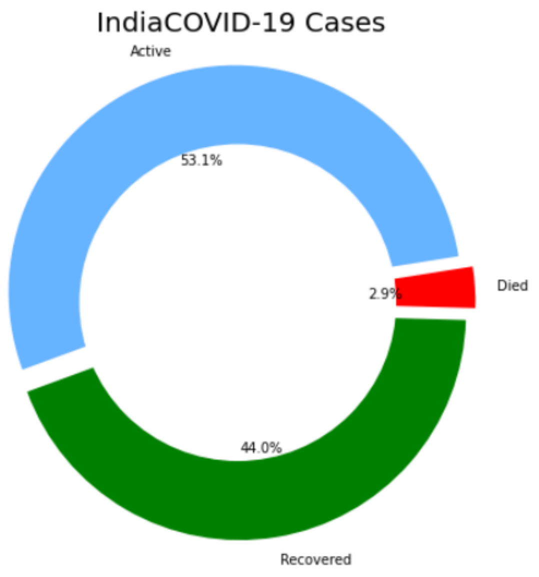
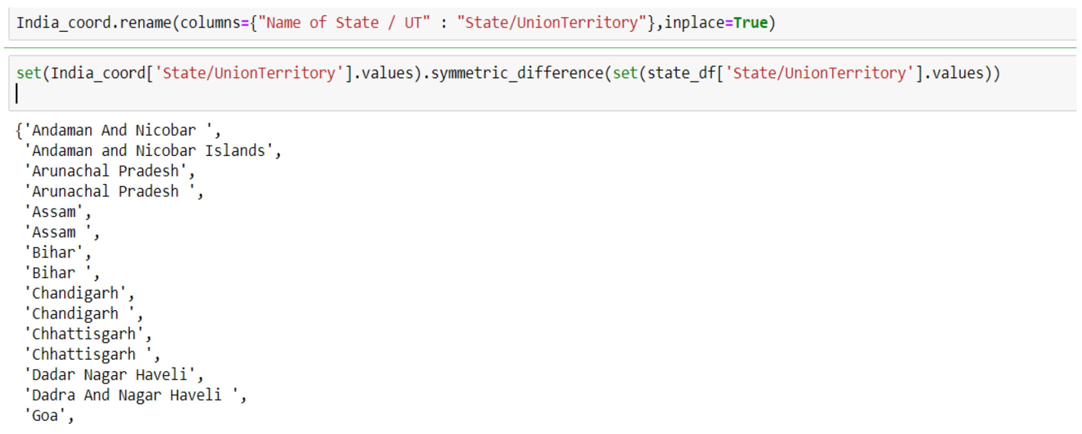


Figure 2.1

.



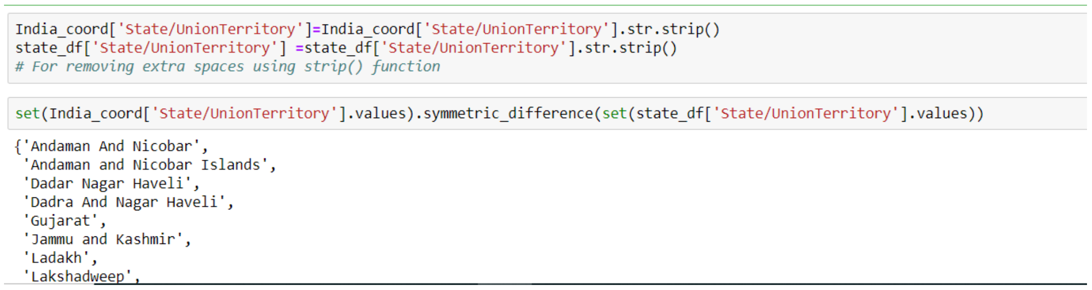
```
India_coord.rename(columns={"Name of State / UT" : "State/UnionTerritory"},inplace=True)
```

```
set(India_coord['State/UnionTerritory'].values).symmetric_difference(set(state_df['State/UnionTerritory'].values))
```

```
{'Andaman And Nicobar ',
 'Andaman and Nicobar Islands',
 'Arunachal Pradesh',
 'Arunachal Pradesh ',
 'Assam',
 'Assam ',
 'Bihar',
 'Bihar ',
 'Chandigarh',
 'Chandigarh ',
 'Chhattisgarh',
 'Chhattisgarh ',
 'Dadar Nagar Haveli',
 'Dadra And Nagar Haveli ',
 'Goa',
```

In the line 16 Code means Change of column's name in Indian Coordinate from Name of State/UT to State/UnionTerritory

Line 17 ‘s code is to compare symmetric differences between names like space etc.



```
India_coord['State/UnionTerritory']=India_coord['State/UnionTerritory'].str.strip()
state_df['State/UnionTerritory'] =state_df['State/UnionTerritory'].str.strip()
# For removing extra spaces using strip() function
```

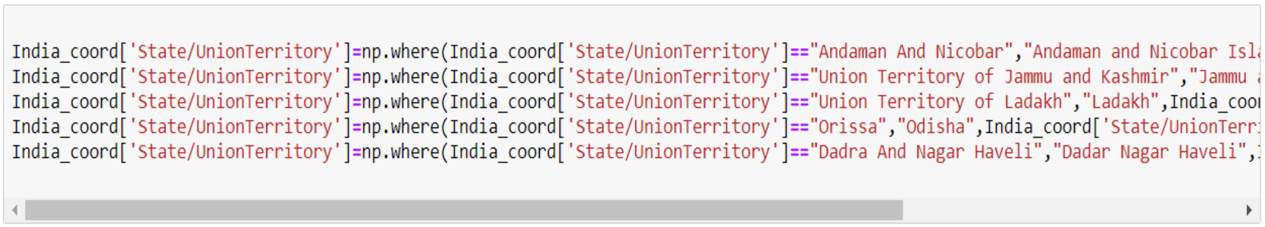
```
set(India_coord['State/UnionTerritory'].values).symmetric_difference(set(state_df['State/UnionTerritory'].values))
```

```
{'Andaman And Nicobar',
 'Andaman and Nicobar Islands',
 'Dadar Nagar Haveli',
 'Dadra And Nagar Haveli',
 'Gujarat',
 'Jammu and Kashmir',
 'Ladakh',
 'Lakshadweep',
```

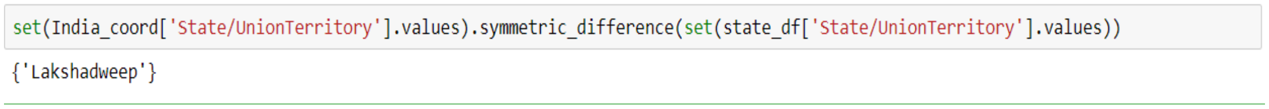
Strip() function is used to cut the spaces between the names.

```
India_coord.loc[India_coord.shape[0]]=['Gujarat','22.2587','71.1924']
India_coord
```

This is used to add Gujarat’s Latitude and Longitude in India_coord and after it is printed

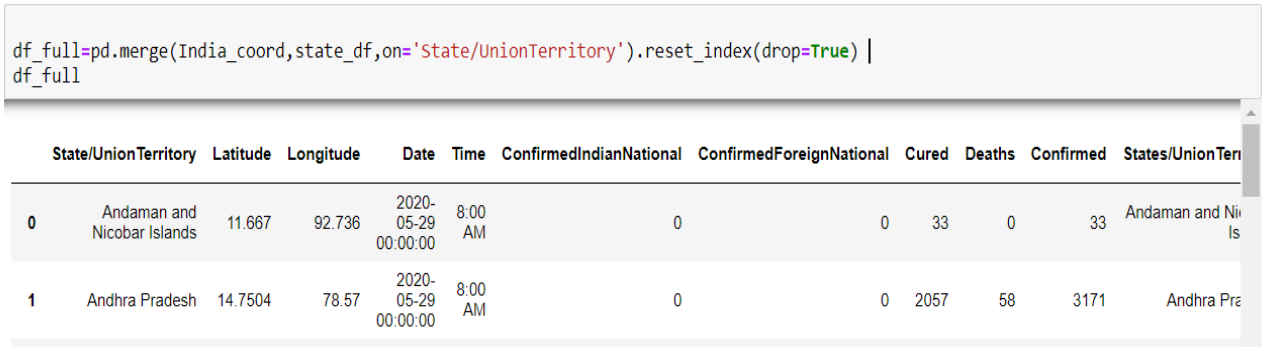


```
India_coord['State/UnionTerritory']=np.where(India_coord['State/UnionTerritory']=="Andaman And Nicobar","Andaman and Nicobar Isla
India_coord['State/UnionTerritory']=np.where(India_coord['State/UnionTerritory']=="Union Territory of Jammu and Kashmir","Jammu a
India_coord['State/UnionTerritory']=np.where(India_coord['State/UnionTerritory']=="Union Territory of Ladakh","Ladakh",India_coor
India_coord['State/UnionTerritory']=np.where(India_coord['State/UnionTerritory']=="Orissa","Odisha",India_coord['State/UnionTerri
India_coord['State/UnionTerritory']=np.where(India_coord['State/UnionTerritory']=="Dadra And Nagar Haveli","Dadar Nagar Haveli",I
```



```
set(India_coord['State/UnionTerritory'].values).symmetric_difference(set(state_df['State/UnionTerritory'].values))
```

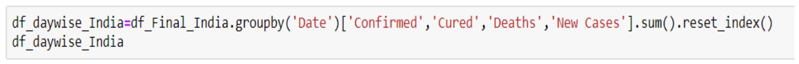
```
{'Lakshadweep'}
```



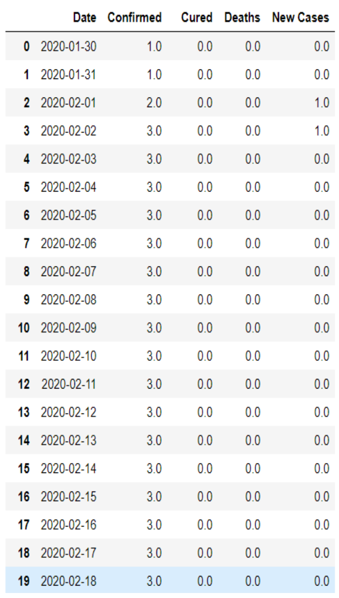
```
df_full=pd.merge(India_coord,state_df,on='State/UnionTerritory').reset_index(drop=True)
df_full
```

	State/UnionTerritory	Latitude	Longitude	Date	Time	ConfirmedIndianNational	ConfirmedForeignNational	Cured	Deaths	Confirmed	States/UnionTerr
0	Andaman and Nicobar Islands	11.667	92.736	2020-05-29 00:00:00	8:00 AM	0	0	33	0	33	Andaman and Ni Is
1	Andhra Pradesh	14.7504	78.57	2020-05-29 00:00:00	8:00 AM	0	0	2057	58	3171	Andhra Pra

In the line 22 Code is written to make names of the state's same, In the line 23 We again check the symmetric differences between the state's name, In the line 24 The one remaining i.e., Lakshadweep will get dropped using reset_index(drop=True) and after the differences between NAMES OF States are over, we have merged those.



```
df_daywise_India=df_Final_India.groupby('Date')['Confirmed','Cured','Deaths','New Cases'].sum().reset_index()
df_daywise_India
```



	Date	Confirmed	Cured	Deaths	New Cases
0	2020-01-30	1.0	0.0	0.0	0.0
1	2020-01-31	1.0	0.0	0.0	0.0
2	2020-02-01	2.0	0.0	0.0	1.0
3	2020-02-02	3.0	0.0	0.0	1.0
4	2020-02-03	3.0	0.0	0.0	0.0
5	2020-02-04	3.0	0.0	0.0	0.0
6	2020-02-05	3.0	0.0	0.0	0.0
7	2020-02-06	3.0	0.0	0.0	0.0
8	2020-02-07	3.0	0.0	0.0	0.0
9	2020-02-08	3.0	0.0	0.0	0.0
10	2020-02-09	3.0	0.0	0.0	0.0
11	2020-02-10	3.0	0.0	0.0	0.0
12	2020-02-11	3.0	0.0	0.0	0.0
13	2020-02-12	3.0	0.0	0.0	0.0
14	2020-02-13	3.0	0.0	0.0	0.0
15	2020-02-14	3.0	0.0	0.0	0.0
16	2020-02-15	3.0	0.0	0.0	0.0
17	2020-02-16	3.0	0.0	0.0	0.0
18	2020-02-17	3.0	0.0	0.0	0.0
19	2020-02-18	3.0	0.0	0.0	0.0

In this we have grouped the data by date and then we have printed that dataset

VISUALIZING THE SPREADS GEOGRAPHICALLY

Folium

folium makes it easy to visualize data that's been manipulated in Python on an interactive leaflet map. It enables both the binding of data to a map for choropleth visualizations as well as passing rich vector/raster/HTML visualizations as markers on the map. The library has a number of built-in tilesets from OpenStreetMap, MapQuest Open, MapQuest Open Aerial, Mapbox, and Stamen, and supports custom tilesets with Mapbox or Cloudmade API keys. Folium supports both GeoJSON and TopoJSON overlays, as well as the binding of data to those overlays to create choropleth maps with color-brewer color schemes.

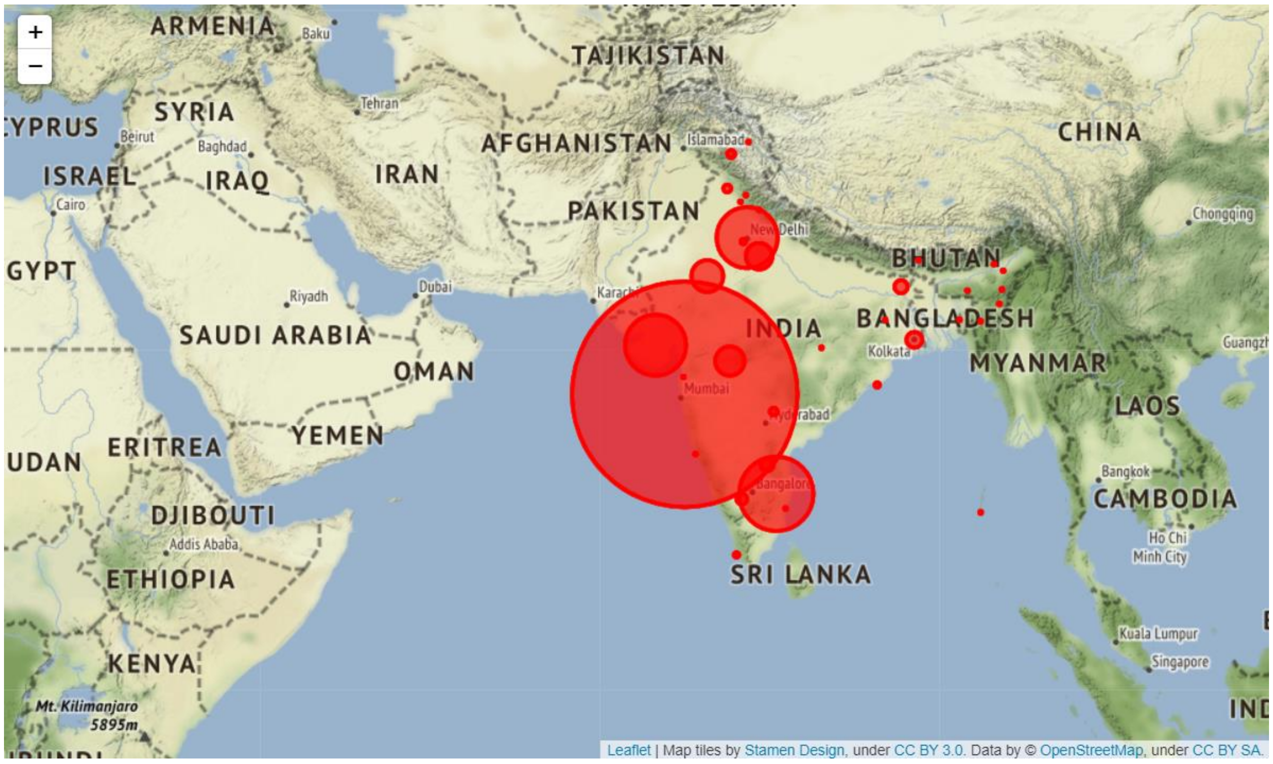
```
map = folium.Map(location=[20, 70], zoom_start=4,tiles='Stamenterrain')
```

The first dimension we wanted to show is geolocation. We set our map's latitude and longitude based on the results of a simple search for "latitude and longitude of India", we found it out to be around '20,70'. Then, we defined our map to initialize at these coordinates. The default zoom is set to 4, the design of the map has also been set as 'Stamen terrain', this type of map showcases advanced labelling and linework generalization of dual-carriageway roads.

```
for lat, lon, value, name in zip(df_full['Latitude'], df_full['Longitude'],
                                 df_full['Confirmed'], df_full['State/UnionTerritory']):
    folium.CircleMarker([lat, lon],
                        radius=value*0.0015,
                        popup = ('<strong>State</strong>: ' + str(name).capitalize()
                                 + '<br>''<strong>Total Cases</strong>: ' + str(value) + '<br>')
                        ,color='red',fill_color='red',fill_opacity=0.3 ).add_to(map)
map
```

The second dimension shows the number of COVID confirmed cases in a particular location so we grouped latitude, longitude, confirmed cases and state/union territory columns of df_full data frame and represented them on map with the help of folium.CiricleMarker which adds circles of the specific radius based on the number of the confirmed cases from df_full data frame, then to have much better visualization we added popup parameter in which we specified the format in which the data should be displayed when we click at a particular state, here html tags like <strong>, <br>

etc. has been used to specify the format, the rest of the parameters specify the design and colour of the marker which will appear over every state. All these details are added to the map by .add_to(map) object.

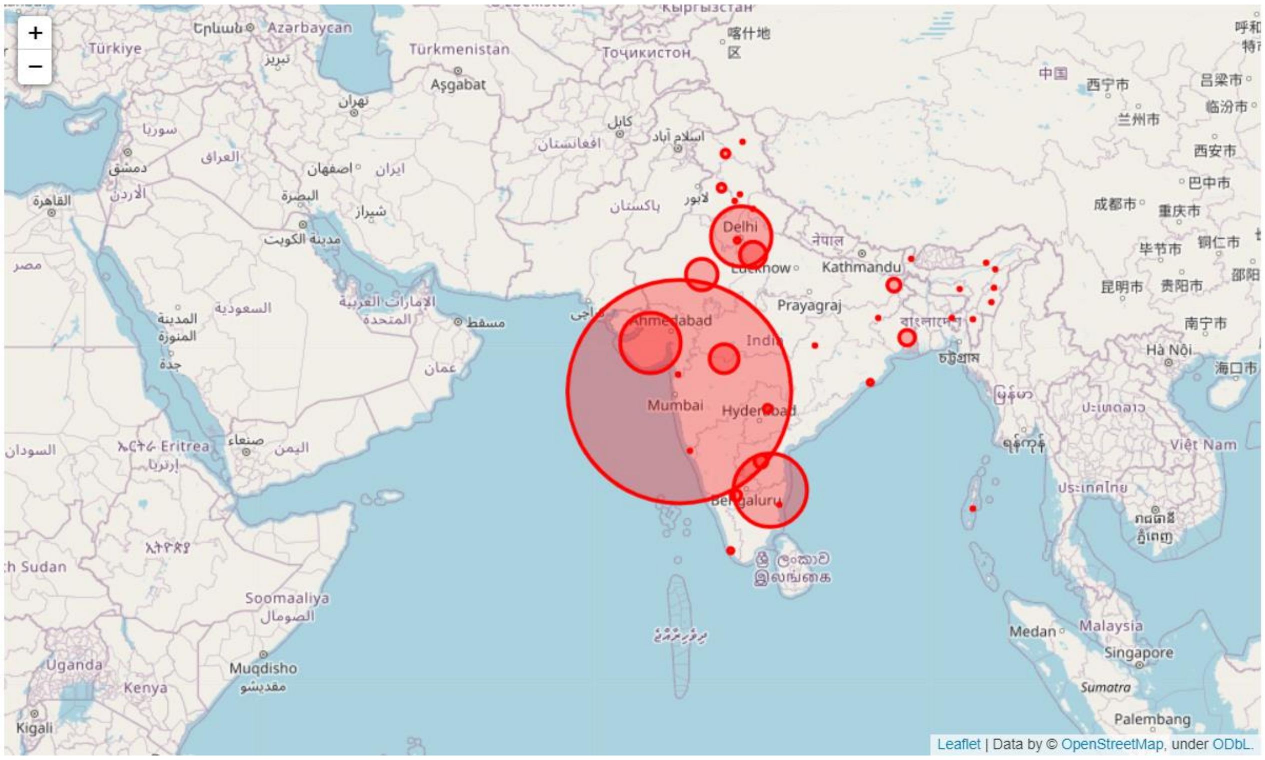


Map 3.1

Now we are changing the map design by using OpenStreetMap, it is a collaborative project to create a free editable map of the world. The data from OSM can be used in various ways including production of paper maps and electronic maps (similar to Google Maps, for example), geocoding of address and place names, and route planning.

```
map = folium.Map(location=[20, 70], zoom_start=4,tiles='OpenStreetMap')
for lat, lon, value, name in zip(df_full['Latitude'], df_full['Longitude'],
                                 df_full['Confirmed'], df_full['State/UnionTerritory']):
    folium.CircleMarker([lat, lon],
                        radius=value*0.0015,
                        popup = ('<strong>State</strong>: ' + str(name).capitalize() +
                                 '<br>''<strong>Total Cases</strong>: ' + str(value) + '<br>'),
                        color='red',fill_color='red',fill_opacity=0.3 ).add_to(map)
map
```

The output of the following code shows OpenStreetMap, rest of the parameters in the following code has been kept the same as before.

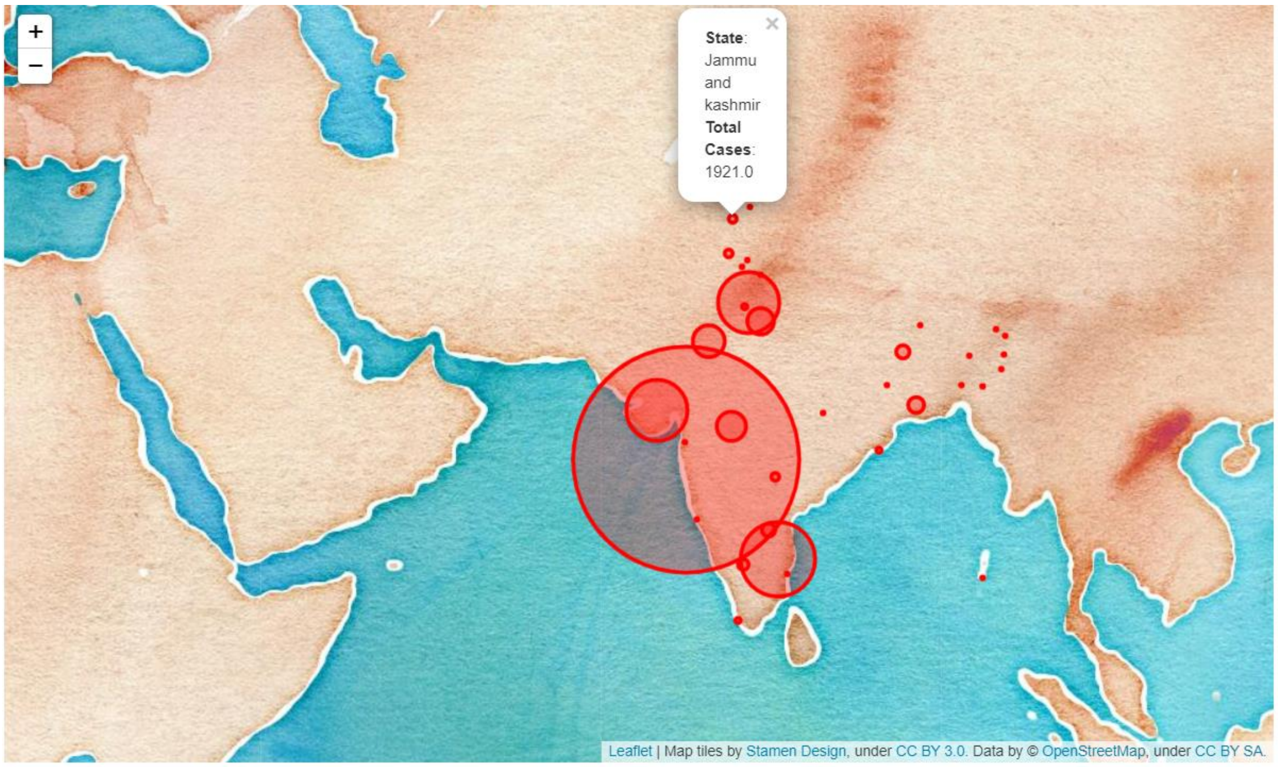


Map 3.2

Next we are again changing the design of the map , this time we are taking Stamenwatercolor , these maps apply raster effect area washes and organic edges over a paper texture to add warm pop to any map.

```
map = folium.Map(location=[20, 70], zoom_start=4,tiles='Stamenwatercolor')

for lat, lon, value, name in zip(df_full['Latitude'], df_full['Longitude'],
                                 df_full['Confirmed'], df_full['State/UnionTerritory']):
    folium.CircleMarker([lat, lon], radius=value*0.0015,
                        popup = ('<strong>State</strong>: ' + str(name).capitalize() + '<br>''<strong>Total Cases</strong>:'
                                 + str(value) + '<br>'),color='red',fill_color='red',fill_opacity=0.3 ).add_to(map)
map
```

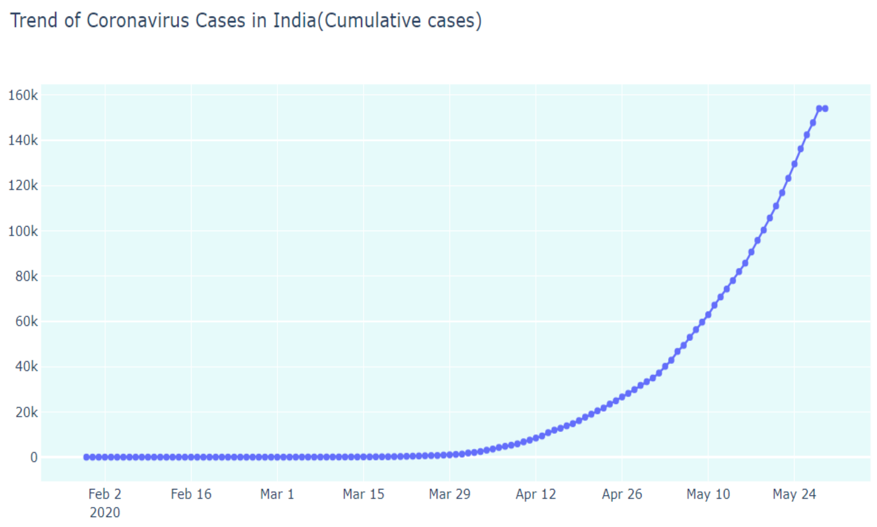
The rest of the parameters have been kept the same as before, the output of the above code is shown in the map below:


+
−
State: Jammu and kashmir
Total Cases: 1921.0
Leaflet | Map tiles by Stamen Design, under CC BY 3.0. Data by © OpenStreetMap, under CC BY SA.

Map 3.3

TREND OF CORONAVIRUS

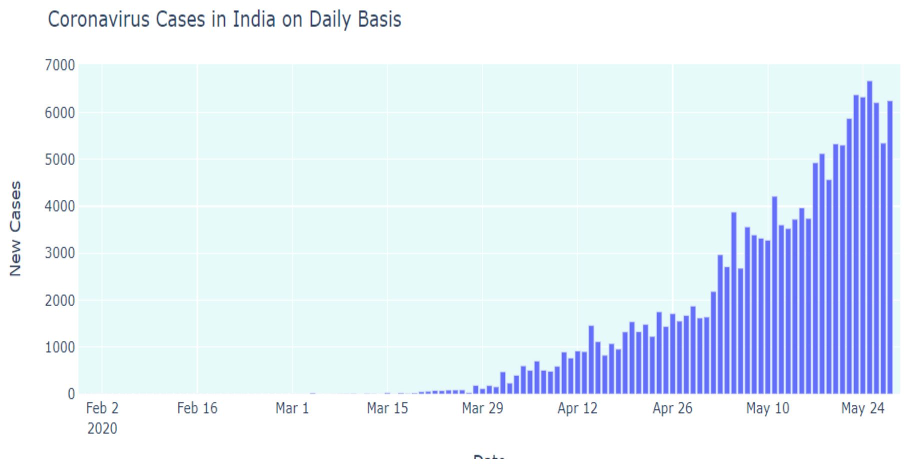
```
fig=go.Figure()

fig.add_trace(go.Scatter(x=df_daywise_India['Date'],y=df_daywise_India['Confirmed'],mode='lines+markers',name='Total Cases'))

fig.update_layout(title_text='Trend of Coronavirus Cases in India(Cumulative cases)',plot_bgcolor='rgb(230,250,250)')
fig.show()
```



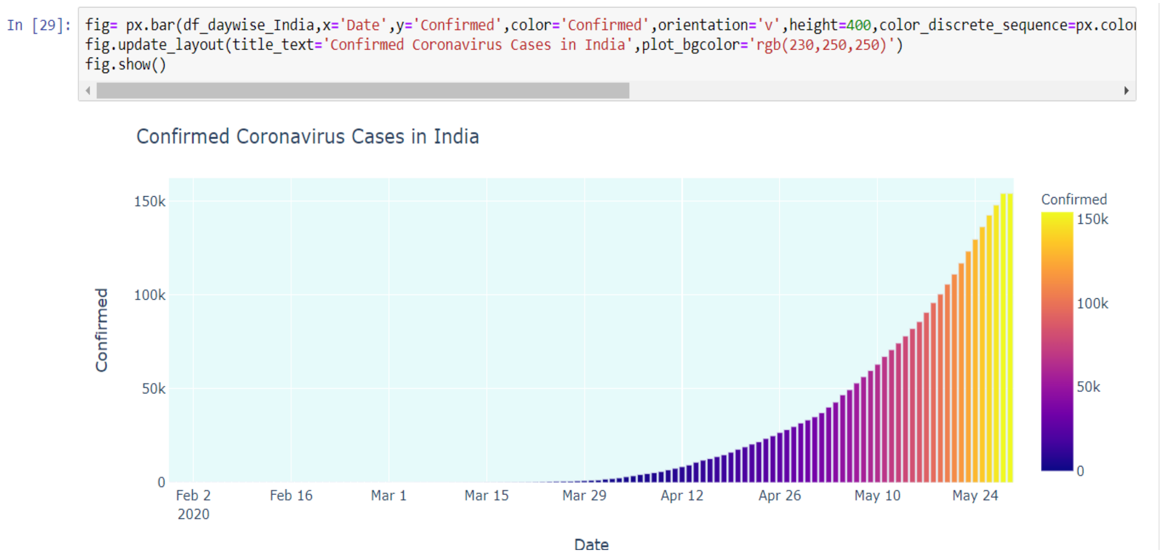
Graph 4.1

go.figure() is used to make plot, add_trace is used For giving parameters and plotting a graph with date on x-axis and Confirmed cases on y-axis ,plotting the graph with lines and markers, Update_layout() is for giving specifications of the graph like title , color etc, Fig.show() is for showing the graph

```
fig= px.bar(df_daywise_India,x='Date',y='New Cases',barmode='group',height=400)  #  Plotting a bargraph with x-axis as Date , y-
fig.update_layout(title_text='Coronavirus Cases in India on Daily Basis',plot_bgcolor='rgb(230,250,250)')
fig.show()
```



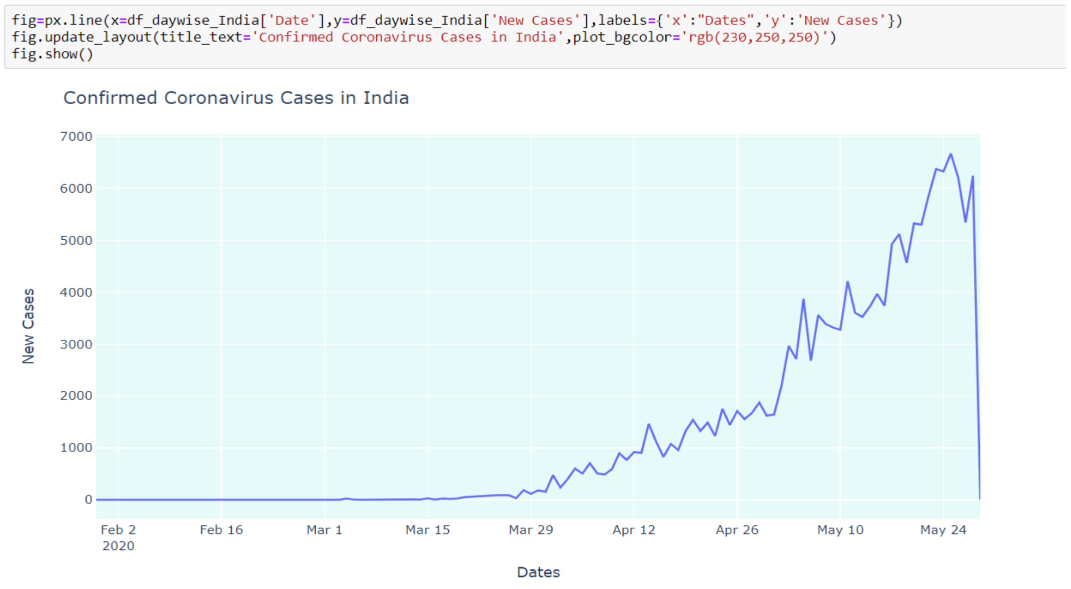
This is code for grouped graph with x-axis as Date , y-axis as New cases , Height of the graph as 400, Update_layout() is for giving specifications of the graph like title , color etc., Fig.show() is for showing the graph



```
In [29]: fig= px.bar(df_daywise_India,x='Date',y='Confirmed',color='Confirmed',orientation='v',height=400,color_discrete_sequence=px.colo
         fig.update_layout(title_text='Confirmed Coronavirus Cases in India',plot_bgcolor='rgb(230,250,250)')
         fig.show()
```

Graph 4.2

Orientation in the px.bar is for defining the orientation of the graph , 'v' means that it is vertical orientation ,color_discrete sequence is used for giving discrete colors, Update_layout() is for giving specifications of the graph like title , color etc, Fig.show() is for showing the graph.



```
fig=px.line(x=df_daywise_India['Date'],y=df_daywise_India['New Cases'],labels={'x':"Dates",'y':'New Cases'})
fig.update_layout(title_text='Confirmed Coronavirus Cases in India',plot_bgcolor='rgb(230,250,250)')
fig.show()
```

Graph 4.3

Px.line is used for making line charts with x-axis as dates and y-axis as New cases.

Labels is used to give titles to x and y axis .

Update_layout() is for giving specifications of the graph like title , color etc.

Fig.show() is for showing the graph.

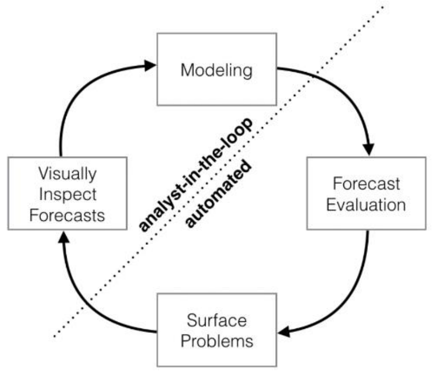
DATA PREDICTION

Prophet Model

Prophet is a procedure for forecasting time series data based on an additive model where non-linear trends are fit with yearly, weekly, and daily seasonality, plus holiday effects. It works best with time series that have strong seasonal effects and several seasons of historical data. Prophet is robust to missing data and shifts in the trend, and typically handles outliers well. It is provided by fbprophet library in python.

```
from fbprophet import Prophet
#conda install -c conda-forge fbprophet
```

Prophet is an open-source tool for business forecasting. In their own words, the motivation behind Prophet is to, make it easier for experts and non-experts to make high quality forecasts that keep up with demand.



Prophet forecast flow

How Prophet works

At its core, the Prophet procedure is an additive regression model with four main components:

- A piecewise linear or logistic growth curve trend. Prophet automatically detects changes in trends by selecting changepoints from the data.
- A yearly seasonal component modeled using Fourier series.
- A weekly seasonal component using dummy variables.
- A user-provided list of important holidays.

Dividing the model into Train and Test sets

```
df = df_daywise_India.iloc[:-1,]
df_train = df.loc[df['Date']<= "2020-05-23",:]
df_test = df.loc[df['Date'] > "2020-05-23",:]
```

The data stored in df_daywise_India Data Frame is as shown below

	Date	Confirmed	Cured	Deaths	New Cases
0	2020-01-30	1.0	0.0	0.0	0.0
1	2020-01-31	1.0	0.0	0.0	0.0
2	2020-02-01	2.0	0.0	0.0	1.0
3	2020-02-02	3.0	0.0	0.0	1.0
4	2020-02-03	3.0	0.0	0.0	0.0
...	...	...	...	...	...
116	2020-05-25	136203.0	57721.0	4021.0	6673.0
117	2020-05-26	142410.0	60491.0	4167.0	6207.0
118	2020-05-27	147754.0	64426.0	4337.0	5344.0
119	2020-05-28	154001.0	67692.0	4531.0	6247.0
120	2020-05-29	154001.0	67692.0	4531.0	0.0

121 rows × 5 columns

We are dividing this into two parts, in the first part we have taken the data for the dates before "23-05-2020" for the training set and dates after "23-05-2020" for the test set. To divide the data, we are initially storing data from df_daywise_India to a df data frame, then using the **pandas.DataFrame.loc** function provided by Pandas library which access a group of rows and columns by label(s), we are separating the data into test (df_test) and train data frames (df_train).

```
confirmed_train = df_train[['Date','Confirmed']]
confirmed_test = df_test[['Date','Confirmed']]
```

Now we are further dividing the data according to their labels, confirmed_train data frame consists of the COVID cases that were confirmed on each date before 23rd of May and in confirmed_test we are storing the confirmed cases after 23rd of May.

```
deaths_train = df_train[['Date','Deaths']]
deaths_test = df_test[['Date','Deaths']]
```

Similarly, the above data frames consist of the total no. of deaths across the country on each date

```
recovered_train = df_train[['Date','Cured']]
recovered_test = df_test[['Date','Cured']]
```

The above data frames consist of no. of people who recovered across the country on each date.

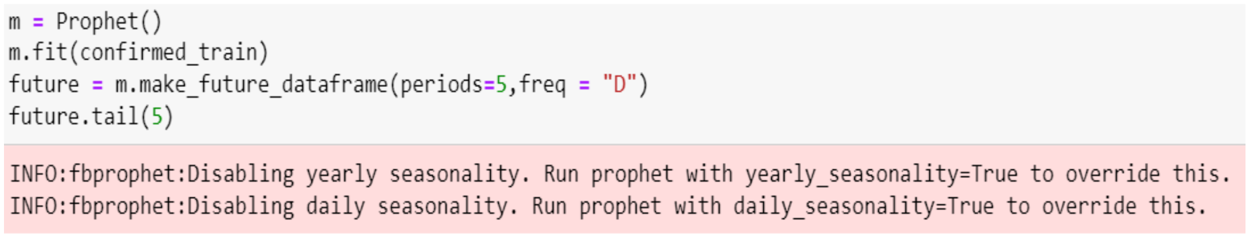
```
confirmed_train.columns = ['ds','y']
confirmed_train.tail()
```

	ds	y
110	2020-05-19	100325.0
111	2020-05-20	105654.0
112	2020-05-21	110956.0
113	2020-05-22	116827.0
114	2020-05-23	123202.0

The above figure shows the no. of confirmed cases in the country on 19th to 23rd May, Pandas **tail()** method is used to return bottom n (5 by default) rows of a data frame or series.

We are renaming columns as ds and y so that we can fit it in Prophet model. The input to Prophet is always a data frame with two columns: ds and y. The ds (date stamp) column should be of a format expected by Pandas, ideally YYYY-MM-DD for a date or YYYY-MM-DD HH:MM: SS for a timestamp. The y column must be numeric, and represents the measurement we wish to forecast.

Training the Model

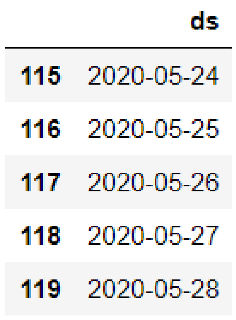


```
m = Prophet()
m.fit(confirmed_train)
future = m.make_future_dataframe(periods=5,freq = "D")
future.tail(5)
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

In the above shown code snippet we fit the model by instantiating a new **Prophet** object. Any settings to the forecasting procedure are passed into the constructor. Then you call its fit method and pass in the historical dataframe. Fitting should take 1-5 seconds. Basically by **.fit** function provided by scikit learn library we are training the model, In contrast to machine learning, fitting means training. There is a fit function in ML, that is used for training of model using data examples. Fit function adjusts weights according to data values so that better accuracy can be achieved. Prophet follows the **sklearn** model API wherein an instance of the Prophet class is created and then the fit and predict methods are called. The model is instantiated by a new Prophet object and followed by calling its fit method and passing in the historical data frame. By default, Prophet uses a linear model for its forecast but a logistic model can also be used by passing it as an attribute.

Predictions are then made on a data frame with a column ds containing the dates for which a prediction is to be made. We can get a suitable data frame that extends into the future a specified number of days using the helper method **Prophet.make_future_dataframe**. By default, it will also include the dates from the history, so we will see the model fit as well, **future.tail(5)** generates the following output:

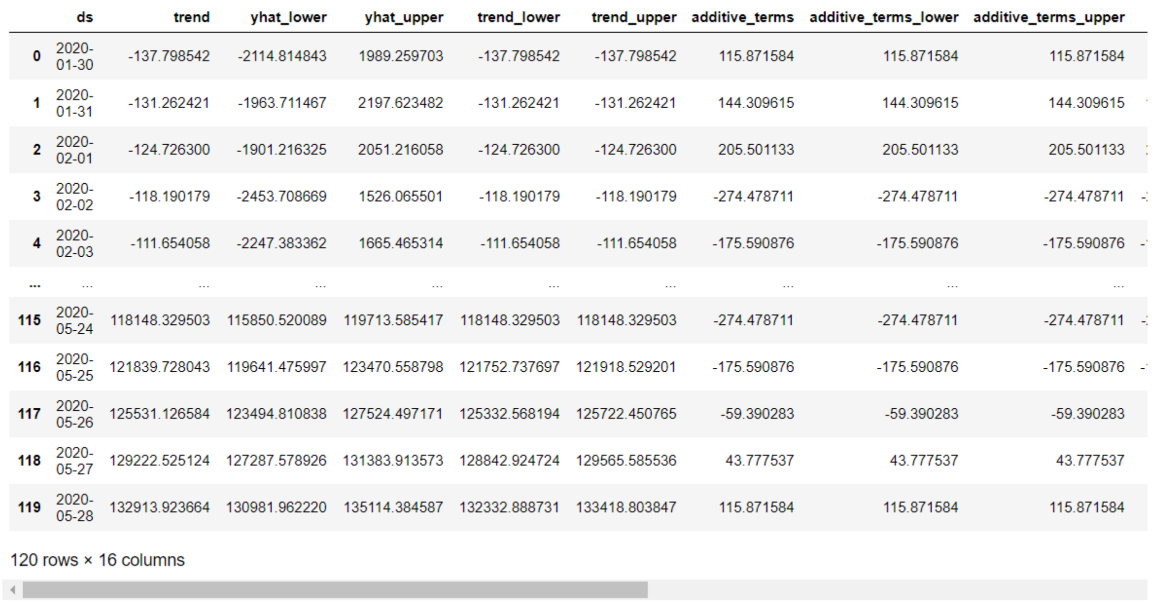


	ds
115	2020-05-24
116	2020-05-25
117	2020-05-26
118	2020-05-27
119	2020-05-28

Making predictions

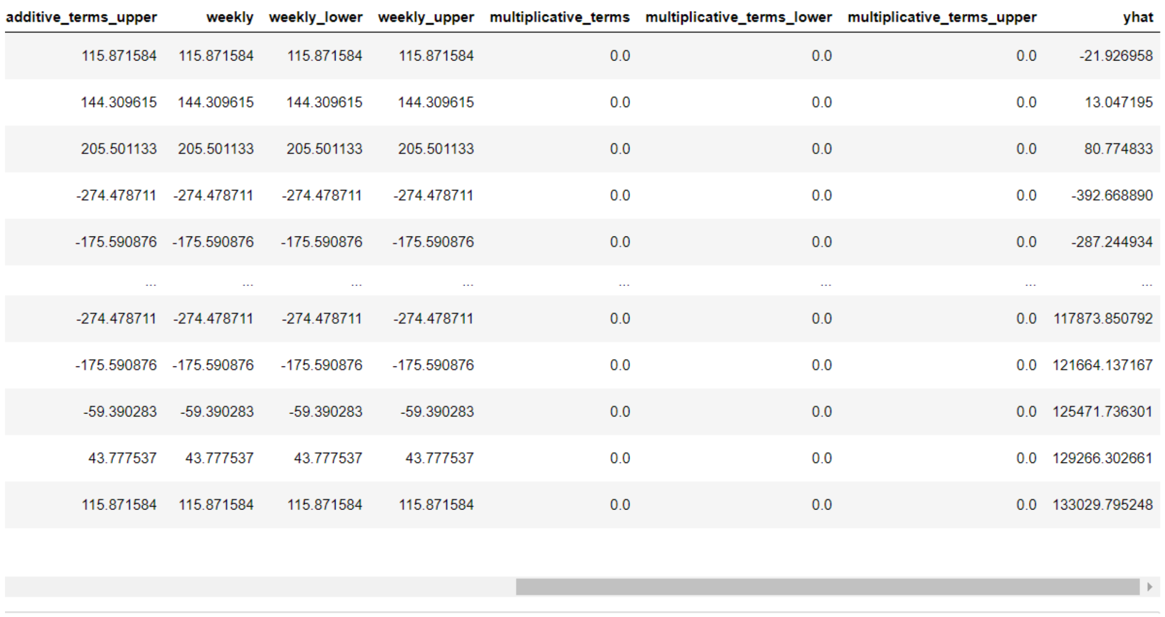
```
forecast = m.predict(future)
forecast
```

The predict method will assign each row in future a predicted value which it names **yhat**. Here we pass in historical dates, this provides an in-sample fit. The forecast object here is a new data frame that includes a column **yhat** with the forecast, as well as columns for components and uncertainty intervals (i.e yhat_lower, yhat_upper, trend_lower, trend_upper etc.)



	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additive_terms_lower	additive_terms_upper
0	2020-01-30	-137.798542	-2114.814843	1989.259703	-137.798542	-137.798542	115.871584	115.871584	115.871584
1	2020-01-31	-131.262421	-1963.711467	2197.623482	-131.262421	-131.262421	144.309615	144.309615	144.309615
2	2020-02-01	-124.726300	-1901.216325	2051.216058	-124.726300	-124.726300	205.501133	205.501133	205.501133
3	2020-02-02	-118.190179	-2453.708669	1526.065501	-118.190179	-118.190179	-274.478711	-274.478711	-274.478711
4	2020-02-03	-111.654058	-2247.383362	1665.465314	-111.654058	-111.654058	-175.590876	-175.590876	-175.590876
...	...	...	...	...	...	...	...	...	...
115	2020-05-24	118148.329503	115850.520089	119713.585417	118148.329503	118148.329503	-274.478711	-274.478711	-274.478711
116	2020-05-25	121839.728043	119641.475997	123470.558798	121752.737697	121918.529201	-175.590876	-175.590876	-175.590876
117	2020-05-26	125531.126584	123494.810838	127524.497171	125332.568194	125722.450765	-59.390283	-59.390283	-59.390283
118	2020-05-27	129222.525124	127287.578926	131383.913573	128842.924724	129565.585536	43.777537	43.777537	43.777537
119	2020-05-28	132913.923664	130981.962220	135114.384587	132332.888731	133418.803847	115.871584	115.871584	115.871584

120 rows × 16 columns



additive_terms_upper	weekly	weekly_lower	weekly_upper	multiplicative_terms	multiplicative_terms_lower	multiplicative_terms_upper	yhat
115.871584	115.871584	115.871584	115.871584	0.0	0.0	0.0	-21.926958
144.309615	144.309615	144.309615	144.309615	0.0	0.0	0.0	13.047195
205.501133	205.501133	205.501133	205.501133	0.0	0.0	0.0	80.774833
-274.478711	-274.478711	-274.478711	-274.478711	0.0	0.0	0.0	-392.668890
-175.590876	-175.590876	-175.590876	-175.590876	0.0	0.0	0.0	-287.244934
...	...	...	...	...	...	...	...
-274.478711	-274.478711	-274.478711	-274.478711	0.0	0.0	0.0	117873.850792
-175.590876	-175.590876	-175.590876	-175.590876	0.0	0.0	0.0	121664.137167
-59.390283	-59.390283	-59.390283	-59.390283	0.0	0.0	0.0	125471.736301
43.777537	43.777537	43.777537	43.777537	0.0	0.0	0.0	129266.302661
115.871584	115.871584	115.871584	115.871584	0.0	0.0	0.0	133029.795248

Comparing the predicted result with actual values

```
result_df = forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail(5)
result_df['Actual'] = confirmed_test['Confirmed']
result_df
```

Now we are comparing certain values from the predicted(forecast) data frame with their actual values (confirmed_test), result_df contains the last 5 values from forecast data frame, in the following line we are adding a new column to result_df data frame which contains the actual values. result_df data frame is shown below.

	ds	yhat	yhat_lower	yhat_upper	Actual
115	2020-05-24	117873.850792	115850.520089	119713.585417	129530.0
116	2020-05-25	121664.137167	119641.475997	123470.558798	136203.0
117	2020-05-26	125471.736301	123494.810838	127524.497171	142410.0
118	2020-05-27	129266.302661	127287.578926	131383.913573	147754.0
119	2020-05-28	133029.795248	130981.962220	135114.384587	154001.0

The above data frame shows us the comparison between actual and predicted values, the number of confirmed cases as predicted by our model is slightly lower than the actual cases that were confirmed on these dates, the accuracy of the model is fair enough to make suitable future predictions.

Comparing via Visualization

```
trace0 = go.Scatter(
        x = result_df['ds'],
        y = result_df['Actual'],
        mode = 'lines+markers',
        name='Actuals',
        line = dict(color = '#dd0000', shape = 'linear'),
        opacity = 0.3,
        connectgaps=True
)
trace1 = go.Scatter(
        x = result_df['ds'],
        y = result_df['yhat'],
        name='Predicted',
        mode = 'lines+markers',
        marker = dict(
            size = 10,
            color = '#44dd00'),
        opacity = 0.3
)
```

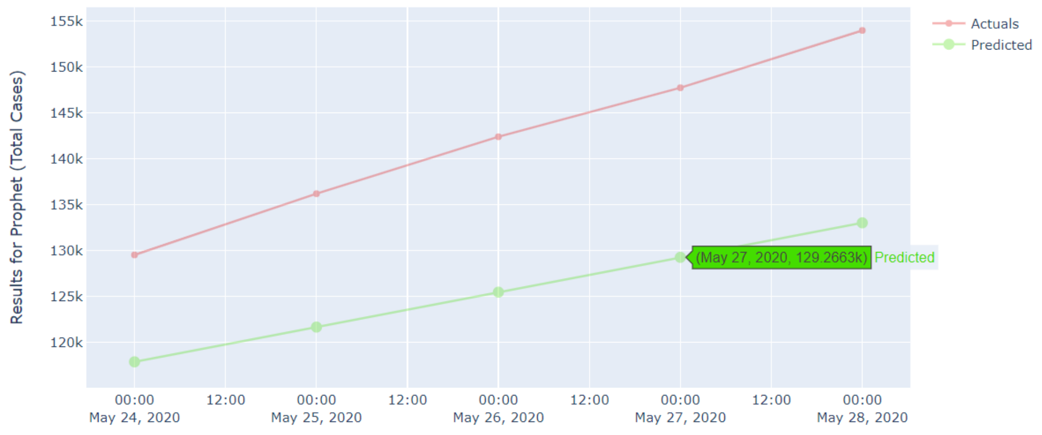
go.Scatter class from **plotly.graph_objects**, can be used both for plotting points (makers) or lines, depending on the value of mode here we are using lines + markers mode, on the x axis date stamp (ds) is taken for both trace0 and trace1 and on the y axis we are plotting the actual confirmed cases (in trace0) and yhat i.e. the predicted confirmed cases (in trace1), the name parameter adds labels to the graph trace0 is labelled as ‘Actual’ and trace1 as ‘Predicted’, the markers parameter specifies the dimension (size) , colour and opacity.

```
data = [trace0, trace1]
```

The above code combines both trace0 and trace1 into single variable so as to plot them on a single graph.

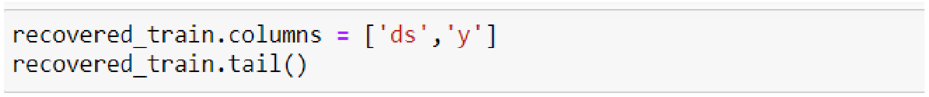
```
layout = go.Layout(
    yaxis=dict(
        title="Results for Prophet (Total Cases)"
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

Graph objects can be turned into their Python dictionary representation using the fig.to_dict() method. We can also retrieve the JSON string representation of a graph object using the fig.to_json() method,to plot the graph we use **go.Figure** it takes data and layout as its parameter which were specified before, **fig.show()** displays the graph

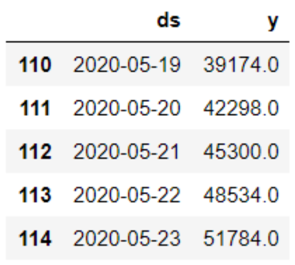


Graph 5.1

Similarly performing the same operations on recovered_train data frame to predict the number of recovered people on the date after 23rd of May.

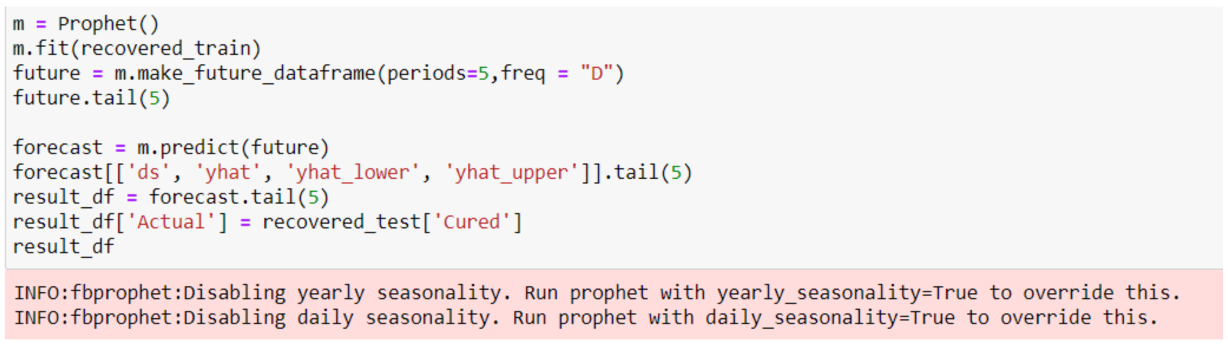


```
recovered_train.columns = ['ds','y']
recovered_train.tail()
```



	ds	y
110	2020-05-19	39174.0
111	2020-05-20	42298.0
112	2020-05-21	45300.0
113	2020-05-22	48534.0
114	2020-05-23	51784.0

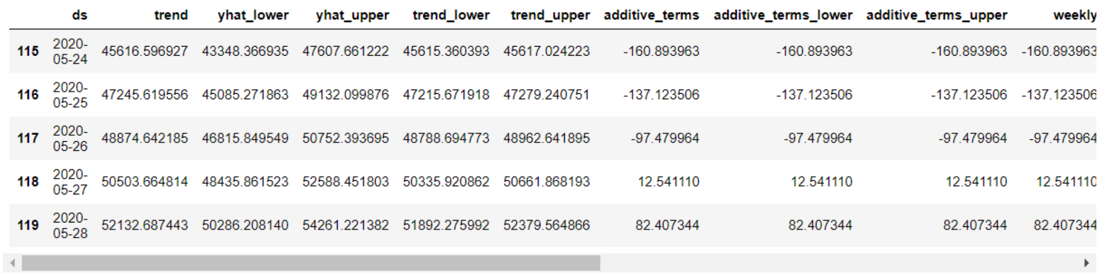
Again, we are specifying date stamp and y (no.) as the column names as they are taken as input by Prophet Model.



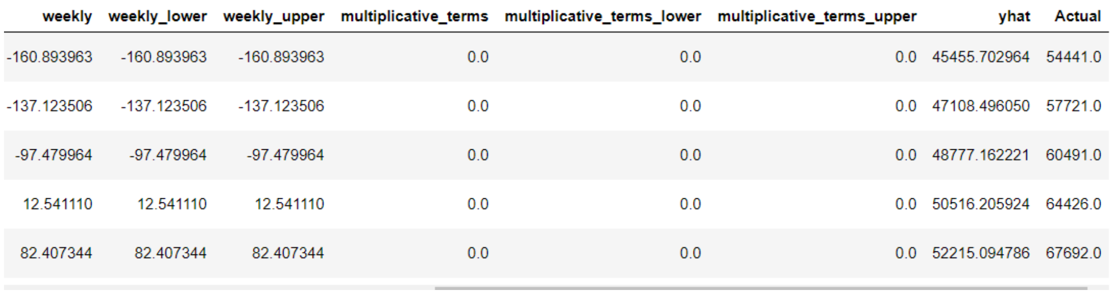
```
m = Prophet()
m.fit(recovered_train)
future = m.make_future_dataframe(periods=5,freq = "D")
future.tail(5)

forecast = m.predict(future)
forecast[['ds', 'yhat', 'yhat_lower', 'yhat_upper']].tail(5)
result_df = forecast.tail(5)
result_df['Actual'] = recovered_test['Cured']
result_df
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

Here we are taking recovered_train as the data frame to fit the model and then making the predictions just like before, the output we obtain is as shown below



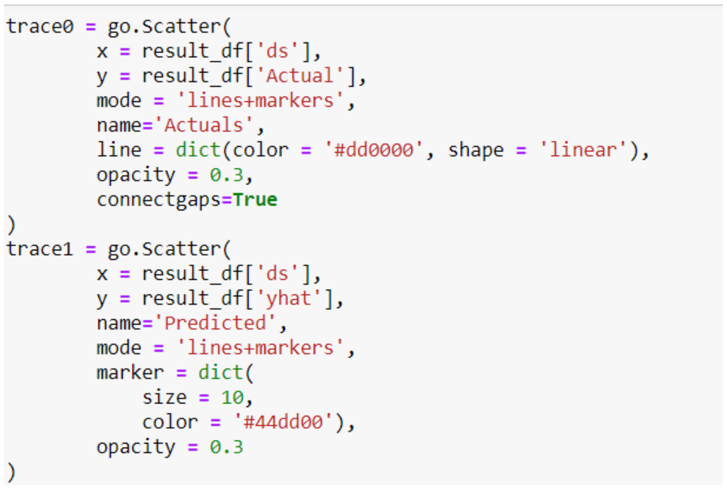
	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	additive_terms	additive_terms_lower	additive_terms_upper	weekly
115	2020-05-24	45616.596927	43348.366935	47607.661222	45615.360393	45617.024223	-160.893963	-160.893963	-160.893963	-160.893963
116	2020-05-25	47245.619556	45085.271863	49132.099876	47215.671918	47279.240751	-137.123506	-137.123506	-137.123506	-137.123506
117	2020-05-26	48874.642185	46815.849549	50752.393695	48788.694773	48962.641895	-97.479964	-97.479964	-97.479964	-97.479964
118	2020-05-27	50503.664814	48435.861523	52588.451803	50335.920862	50661.868193	12.541110	12.541110	12.541110	12.541110
119	2020-05-28	52132.687443	50286.208140	54261.221382	51892.275992	52379.564866	82.407344	82.407344	82.407344	82.407344



weekly	weekly_lower	weekly_upper	multiplicative_terms	multiplicative_terms_lower	multiplicative_terms_upper	yhat	Actual
-160.893963	-160.893963	-160.893963	0.0	0.0	0.0	45455.702964	54441.0
-137.123506	-137.123506	-137.123506	0.0	0.0	0.0	47108.496050	57721.0
-97.479964	-97.479964	-97.479964	0.0	0.0	0.0	48777.162221	60491.0
12.541110	12.541110	12.541110	0.0	0.0	0.0	50516.205924	64426.0
82.407344	82.407344	82.407344	0.0	0.0	0.0	52215.094786	67692.0

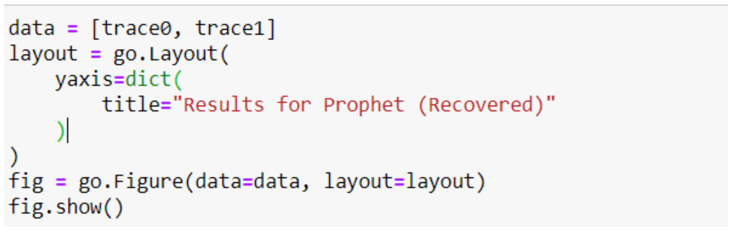
From the above result we observe that the number of recovered cases as predicted by our model are close to the actual values.

<u>Comparing via Visualization</u>



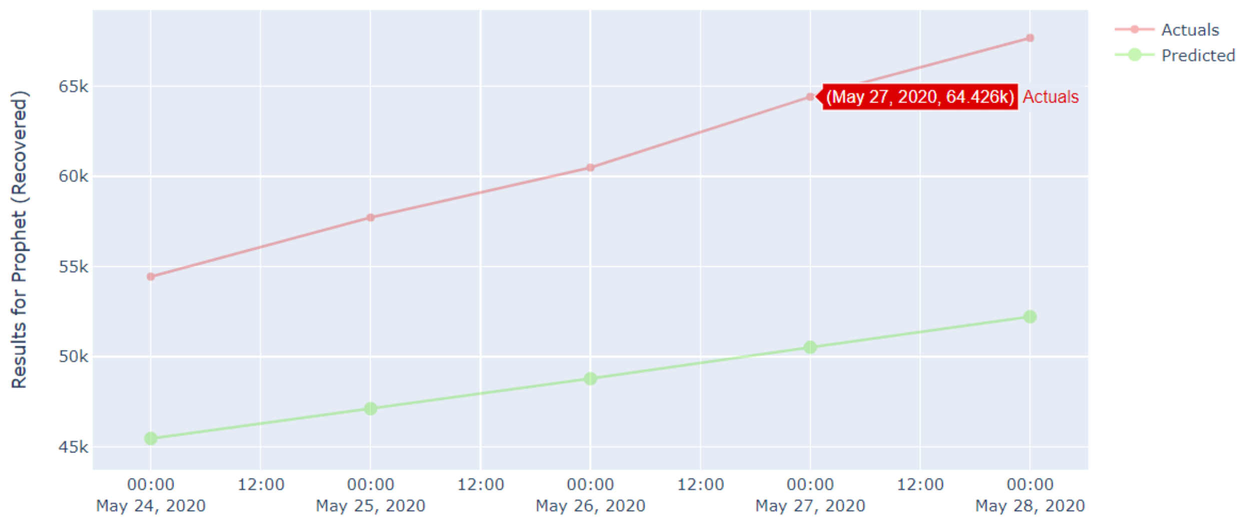
```
trace0 = go.Scatter(
        x = result_df['ds'],
        y = result_df['Actual'],
        mode = 'lines+markers',
        name='Actuals',
        line = dict(color = '#dd0000', shape = 'linear'),
        opacity = 0.3,
        connectgaps=True
)
trace1 = go.Scatter(
        x = result_df['ds'],
        y = result_df['yhat'],
        name='Predicted',
        mode = 'lines+markers',
        marker = dict(
            size = 10,
            color = '#44dd00'),
        opacity = 0.3
)
```

Similarly we are again tracing the predicted and actual curves using go.Scatter class of pyplot library



```
data = [trace0, trace1]
layout = go.Layout(
    yaxis=dict(
        title="Results for Prophet (Recovered)"
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

Using go.Figure we get the following curve



Graph 5.2

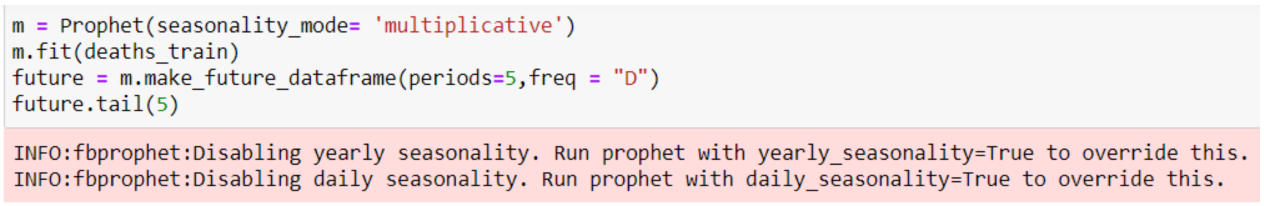
In the graph above we can see that the number of predicted recovered cases are slightly less than that who actually recovered.

Now we are again performing the same operations that we performed to obtain confirmed and recovered cases, on death_train data frame as shown below

```
deaths_train.columns = ['ds','y']
deaths_train.tail()
```

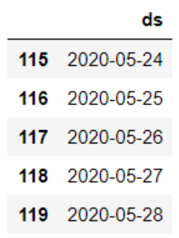
	ds	y
110	2020-05-19	3163.0
111	2020-05-20	3303.0
112	2020-05-21	3435.0
113	2020-05-22	3583.0
114	2020-05-23	3720.0

Again, we are fitting the model but this time we are using a **seasonality_mode,** the seasonality in the forecast is too large at the start of the time series and too small at the end. In this time series, the seasonality is not a constant additive factor as assumed by Prophet, rather it grows with the trend. This is multiplicative seasonality.



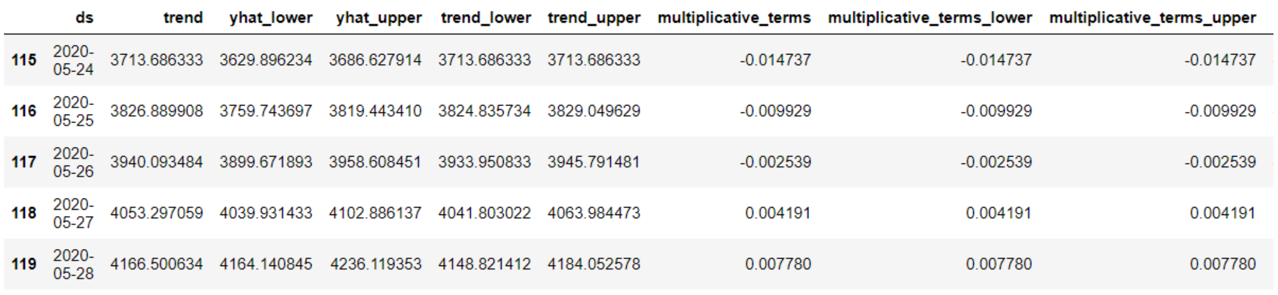
```
m = Prophet(seasonality_mode= 'multiplicative')
m.fit(deaths_train)
future = m.make_future_dataframe(periods=5,freq = "D")
future.tail(5)
```

```
INFO:fbprophet:Disabling yearly seasonality. Run prophet with yearly_seasonality=True to override this.
INFO:fbprophet:Disabling daily seasonality. Run prophet with daily_seasonality=True to override this.
```

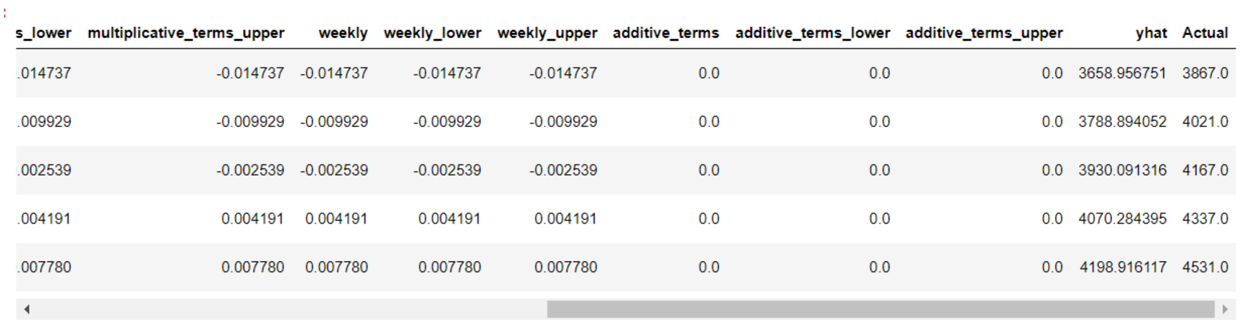


	ds
115	2020-05-24
116	2020-05-25
117	2020-05-26
118	2020-05-27
119	2020-05-28

Now we are again predicting the result using the predict object of the model class



	ds	trend	yhat_lower	yhat_upper	trend_lower	trend_upper	multiplicative_terms	multiplicative_terms_lower	multiplicative_terms_upper
115	2020-05-24	3713.686333	3629.896234	3686.627914	3713.686333	3713.686333	-0.014737	-0.014737	-0.014737
116	2020-05-25	3826.889908	3759.743697	3819.443410	3824.835734	3829.049629	-0.009929	-0.009929	-0.009929
117	2020-05-26	3940.093484	3899.671893	3958.608451	3933.950833	3945.791481	-0.002539	-0.002539	-0.002539
118	2020-05-27	4053.297059	4039.931433	4102.886137	4041.803022	4063.984473	0.004191	0.004191	0.004191
119	2020-05-28	4166.500634	4164.140845	4236.119353	4148.821412	4184.052578	0.007780	0.007780	0.007780



s_lower	multiplicative_terms_upper	weekly	weekly_lower	weekly_upper	additive_terms	additive_terms_lower	additive_terms_upper	yhat	Actual
.014737	-0.014737	-0.014737	-0.014737	-0.014737	0.0	0.0	0.0	3658.956751	3867.0
.009929	-0.009929	-0.009929	-0.009929	-0.009929	0.0	0.0	0.0	3788.894052	4021.0
.002539	-0.002539	-0.002539	-0.002539	-0.002539	0.0	0.0	0.0	3930.091316	4167.0
.004191	0.004191	0.004191	0.004191	0.004191	0.0	0.0	0.0	4070.284395	4337.0
.007780	0.007780	0.007780	0.007780	0.007780	0.0	0.0	0.0	4198.916117	4531.0

Here too the predicted output is quite comparable to the actual result

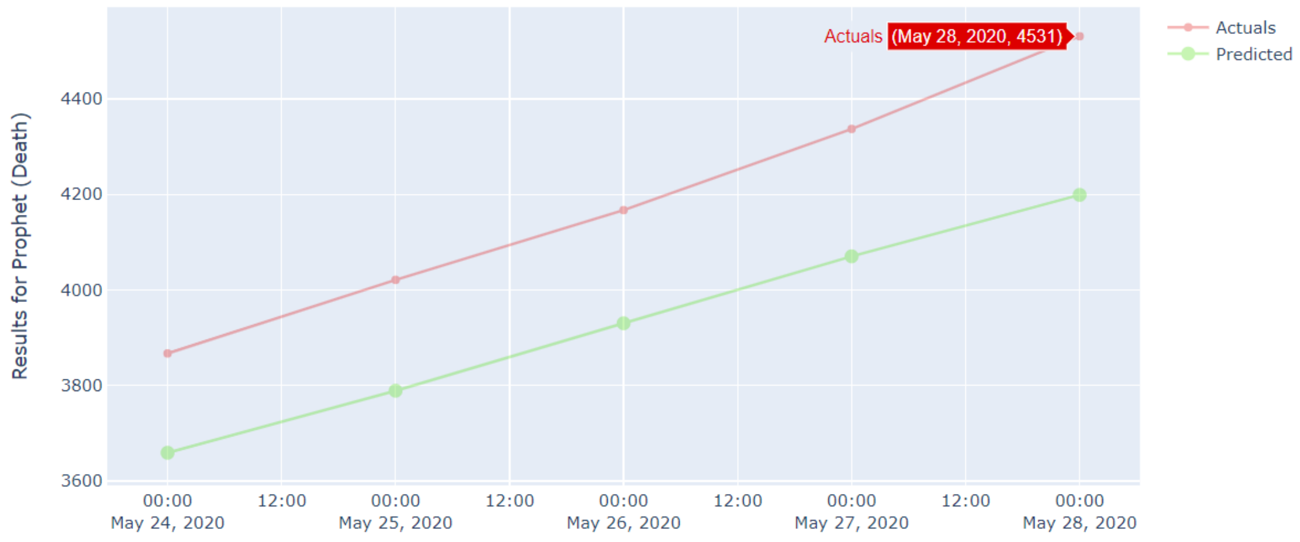
Comparing via Visualization

```
trace0 = go.Scatter(
        x = result_df['ds'],
        y = result_df['Actual'],
        mode = 'lines+markers',
        name='Actuals',
        line = dict(color = '#dd0000', shape = 'linear'),
        opacity = 0.3,
        connectgaps=True
)
trace1 = go.Scatter(
        x = result_df['ds'],
        y = result_df['yhat'],
        name='Predicted',
        mode = 'lines+markers',
        marker = dict(
            size = 10,
            color = '#44dd00'),
        opacity = 0.3
)
```

Similarly, as we did twice before we are making graphs for the predicted and actual deaths due to COVID from 24th to 28th May.

```
data = [trace0, trace1]
layout = go.Layout(
    yaxis=dict(
        title="Results for Prophet (Death)"
    )
)
fig = go.Figure(data=data, layout=layout)
fig.show()
```

From the above go.Figure object we obtain the following graph



Graph 5.3

CONCLUSION

In our project, we imported two files

1. covid_19_india
2. Indian Coordinates

We then prepared our data for analysis with the help of data cleaning by removing or modifying data that was incorrect, incomplete, irrelevant, duplicated, or improperly formatted.

Then we merged two data frames, in this we got all the information about a particular entity common to both data sets, further each state's cases are analyzed using a pie and later overall country's cases are analyzed.

i. In states, Maharashtra, Gujrat, MP, West Bengal, Meghalaya had death rate nearly 5 percent
ii. In states, Tamil Nadu, MP, UP, Rajasthan, Haryana, Punjab, Arunachala Pradesh, Kerala, Andhra Pradesh more than 50 percent people had recovered.
iii. No death cases were recorded in Mizoram, Sikkim, Nagaland, Meghalaya
iv. The total active cases in India by 29th May were recorded to be 53.1% and the recovered were around 44%, the death rate was 2.9%

Next, we visualized the number of cases in India geographically with which showed us the worst affected areas of India by COVID, after this we saw the trend of COVID from Jan to the end of May, the maximum number of cases appeared on 26th May, by the end of May the number of confirmed cases in India raised up to 160k.

Later, with the help of fbprophet library we estimated the rough figures of number of cases of COVID, these predictions are plotted on a graph through which we draw following conclusions:

i. Considering the data set used being small, the predictions were quite correct.
ii. In all three parameters, the number of actual cases outnumbered the predicted ones.
iii. The slope of actual cases curve is more than predicted stating the rise in coefficient is higher in actual case scenario.