Mapping Applications in Python (Basemap and Folium)

22

LEARNING OBJECTIVES

After studying this chapter, you will be able to:

- **LO 1** Know the basics of Basemap toolkit extension for Matplotlib
- **LO 2** Examine the Basemap class which has a large number of attributes (about 35), but only a few are important and will suffice for most purposes
- **LO 3** Understand and use the Folium library for mapping and using its Map() class
- **LO 4** Add a "marker" to a folium map
- **LO 5** Use the Pandas library to import data from a .csv file and plot it on folium map
- **LO 6** Understand basics of GeoJSON format
- **LO 7** Use choropleth maps

22.1 INTRODUCTION

There are a number of libraries/toolkits for mapping in Python. This chapter deals with two mapping applications namely Basemap and Folium. The Basemap toolkit is an extension of the Matplotlib toolkits and can be used to plot 2D data. The Basemap toolkit is to be found under the mpl_toolkits namespace, i.e., at matplotlib.mpl_toolkits.basemap.

Note that Basemap does not do plotting on its own. Internally it uses a C Library PROJ.4 and some other libraries like GEOS and then it depends upon Matplotlib for plotting. Also note that it is possible that the Basemap toolkit may be replaced by a new Python package named Cartopy[1].

[1] See: https://scitools.org.uk/cartopy/docs/latest/

On the other hand, folium is a library for "interactive maps". Folium is actually a "Python port" of a Javascript library called Leaflet. Note that Folium is not the only "port" of Leaflet. There is another library called mapboxgl, which is also a Python port. However, in this chapter only Folium is discussed.

22.2 BASEMAP BASICS

Basemap is used for creating maps using Python. It is an extension of Matplotlib. It has features for data visualization, geographical projections and it also has some data sets to plot countries directly. This section deals with installation and basic usage of the Basemap library.

22.2.1 Basemap Introduction and Installation

To install a package in Anaconda, you may open the Anaconda prompt and type the command:

```
conda install -c anaconda basemap
```

Problems in downloading and use of .whl: Sometimes when you try to download a package, there may be errors due to version conflict. For example, on the author's system, an error was displayed, which is shown in Figure 22.1.

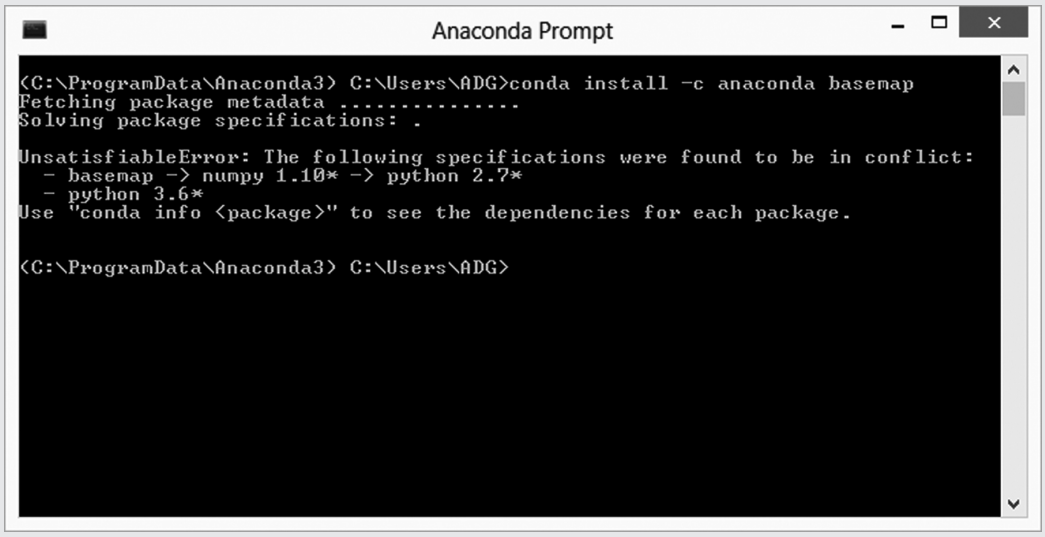


FIGURE 22.1 Screen shot of error when trying to download basemap module.

It appears that there is a version conflict. Apparently the official version of Basemap is in Python version 2.x whereas the author's machine has Python 3.x.

The solution to this is to go to unofficial windows binaries available at:
https://www.lfd.uci.edu/~gohlke/pythonlibs/ .

Note that the whl format is a special zip format for Python packages. The screen shot of the website is shown in Figure 22.2.

FIGURE 22.2 Webpage from where the .whl Python packages can be downloaded

You need to first install the pyproj package and then the .whl package appropriate to your system. First download the appropriate package and then to install the package, open anaconda prompt and change directory using cd command to go the location where you have downloaded the file. Then use pip as follows:

```
pip install [Name_of_package].
```

On the author's system, the following wheel file was downloaded:basemap-1.1.0-cp36-cp36m-win32.whl to location C:\Users\DS\Downloads. Then the cd command was used to change the directory to the one where this file was downloaded. Then pip command was finally used to install the package. The screen shot of steps taken is shown in Figure 22.3.

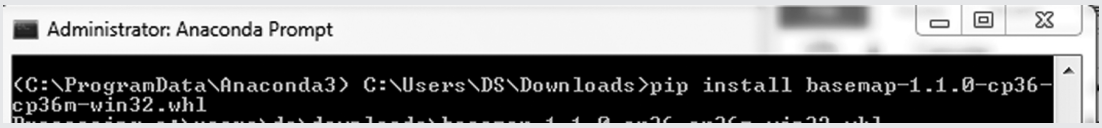


FIGURE 22.3 Use of pip command to install basemap .whl package

22.2.2 The Basemap Library and Basemap() Class of Basemap Library

To use the basemap library, you need to import the Basemap class. The complete signature of the Basemap() class is as follows:

```
mpl_toolkits.basemap.Basemap(llcrnrlon=None, llcrnrlat=None, urcrnrlon=None,
urcrnrlat=None, llcrnrx=None, llcrnry=None, urcrnrx=None, urcrnry=None,
width=None, height=None, projection='cyl', resolution='c', area_thresh=None,
rsphere=6370997.0, ellps=None, lat_ts=None, lat_1=None, lat_2=None, lat_0=None,
lon_0=None, lon_1=None, lon_2=None, o_lon_p=None, o_lat_p=None, k_0=None,
no_rot=False, suppress_ticks=True, satellite_height=35786000, boundinglat=None,
fix_aspect=True, anchor='C', celestial=False, round=False, epsg=None, ax=None)
```

The large number of attribute to the class constructor may look intimidating. However, most of them have default values and hence can be ignored.

However, you need to specify the "bounding box" of the map you want to display. This "bounding box" may be specified in the following two ways:

1. Give the coordinates of the bottom left corner and the top right corner. Note that on earth coordinate systems, the longitude can be thought of as X-axis and the latitude can be thought of as the Y-axis. So as you move from West to East, you are moving in X-direction. Further as you move from equator towards North pole, you are moving in Y-direction. So you have:
   - llcrnrlon: Longitude of the lower left corner. (Note llcrnrlon is amalgamation of four words: lower, left, corner and longitude)
   - llcrnrlat: Latitude of lower left corner. (Note llcrnrlat is amalgamation of four words: lower, left, corner and latitude).
   - urcrnrlon: Longitude of upper right corner. (Note urcrnrlon is amalgamation of four words: upper, right, corner and longitude).
   - urcrnrlat: Latitude of upper right corner. (Note urcrnrlat is amalgamation of four words: upper, right, corner and latitude)
2. Give the coordinates of the centre of the map in terms of longitude and latitude (in degrees) and also give the width and height of the map (both in meters). So you have:
   - lon_0: Longitude of the centre of the map (in degrees)
   - lat_0: Latitude of the centre of the map (in degrees)
   - width: Width of map in projection coordinates (in meters)
   - height: Height of map in projection coordinates (in meters)

So you may specify the "bounding box" of the desired map either by specifying the first set of four numbers or the second set of four numbers given above.

22.2.3 Using the Basemap Library

The following script uses the basemap library to draw some maps:

```
import matplotlib.pyplot as plt
from mpl_toolkits.basemap import Basemap

fig = plt.figure(figsize = (10, 6))

ax1 = fig.add_subplot(121)
ax1.set_title("Sinusoidal")
map = Basemap(projection='sinu',lon_0 = 82,lat_0 = 23,ax = ax1, resolution = 'i')
map.drawmapboundary(fill_color='aqua')
map.fillcontinents(color='coral',lake_color='aqua')
```

```
11 map.drawcoastlines()
12
13 ax2 = fig.add_subplot(122)
14 ax2.set_title("Robinson")
15 map = Basemap(projection='robin', lon_0 = 10, lat_0 = 50, ax = ax2)
16 map.drawmapboundary(fill_color='aqua')
17 map.fillcontinents(color='coral',lake_color='aqua')
18 map.drawcountries()
19 map.drawparallels(circles =[0, 8, 23.5, 37])
20 map.drawmeridians(meridians = [0, 82.5])
21
22 plt.show()
```

Explanation:

Line 2: You need to import Basemap from the basemap module. (Note that the module name begins with small b while the method name has a big or capital B.)

Line 4: As before create a figure object and call it fig.

Line 6: Two sub-plots are created.

Line 8: This is the call to the Basemap() class constructor. You can look up the online documentation at https://matplotlib.org/basemap/api/basemap_api.html to see all the options available. The projection parameter has different types of parameters like azimuthal, geostationary, etc. Here the sinusoidal parameter has been used for which the value of the projection parameter should be 'sinu'. The lon_0 parameter indicates the longitude of the centre of the map. Since IST is at 82.5 degrees East, this map will centre around India. Similarly, lat_0 is the latitude of the centre of the map which has been taken as 23 degree North.

Lines 9-11: These are three methods used to draw map boundary, fill continents and to draw coastline. These methods also have a large number of parameters. The signature of drawmapboundary() method is:

```
drawmapboundary(color='k', linewidth=1.0, fill_color=None, zorder=None, ax=None)
```

The signature of fillcontinents() method is:

```
fillcontinents(color='0.8', lake_color=None, ax=None, zorder=None, alpha=None)
```

The signature of drawcoastlines() is:

```
drawcoastlines(linewidth=1.0, linestyle='solid', color='k', antialiased=1, ax=None,
zorder=None)
```

Line 13-20: This draws a second sub-plot.

Line 19-20: These two lines have two methods namely drawparallels() and drawmeridians(). The signature of drawparallels() is:

```
1 drawparallels(circles, color='k', textcolor='k', linewidth=1.0, zorder=None,
2 dashes=[1, 1], labels=[0, 0, 0, 0], labelstyle=None, fmt='%g', xoffset=None,
3 yoffset=None, ax=None, latmax=None, **text_kwargs)
```

Here the parameter circles must be a valid Python sequence or even a NumPy array. Remember lists, tuples, etc. are all sequences in Python. So in this case, you may use `circles =[0, 8, 23.5, 37]`, which means you will draw parallel at 0, i.e., equator, at 8 degree North, i.e., south most end of India , 23.5, i.e., tropic of cancer and 37, i.e., north most end of India.

And the signature for drawmeridians() is:

```
drawmeridians(meridians, color='k', textcolor='k', linewidth=1.0, zorder=None,
dashes=[1, 1], labels=[0, 0, 0, 0], labelstyle=None, fmt='%g', xoffset=None,
yoffset=None, ax=None, latmax=None, **text_kwargs)
```

Here also the parameter meridians must also be a valid Python sequence giving the sequence of meridians to be drawn in degrees.

The output is shown in Figure 22.4.

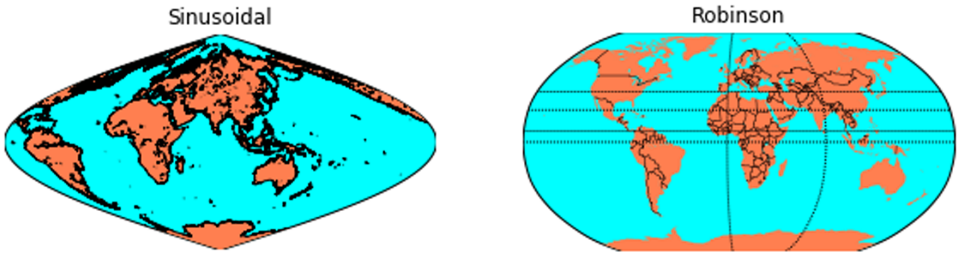


FIGURE 22.4 Output of previous script. Two maps have been drawn. The map on the left has: `projection='sinu',lon_0 = 82,lat_0 = 23`. The map on right has: `projection='robin', lon_0 = 10, lat_0 = 50.`

22.3 FOLIUM[2]

Python is excellent for data manipulation while javascript is excellent for creating web applications and visualization. In javascript there is a library called leaflet[3]. Leaflet is an open-source JavaScript library creating interactive maps. Folium is the Python port of leaflet. Folium is a huge library. In this section only the following points are discussed:

- Installation
- The "Map" class
- Adding a marker
- Combining Folium with pandas data

22.3.1 Installation

You can do a pip install folium on Jupyter notebook. However, if folium is already installed, you will get a message as shown in Figure 22.5.

[2] For folium documentation see: https://media.readthedocs.org/pdf/folium/latest/folium.pdf

[3] See: https://leafletjs.com/

```
In [1]:  1  !pip install folium

Requirement already satisfied: folium in c:\programdata\anaconda3\l
ib\site-packages (0.5.0)
Requirement already satisfied: branca in c:\programdata\anaconda3\l
ib\site-packages (from folium) (0.3.0)
Requirement already satisfied: requests in c:\programdata\anaconda3
\lib\site-packages (from folium) (2.21.0)
Requirement already satisfied: jinja2 in c:\programdata\anaconda3\l
ib\site-packages (from folium) (2.9.6)
Requirement already satisfied: six in c:\programdata\anaconda3\lib
\site-packages (from folium) (1.11.0)
Requirement already satisfied: idna<2.9,>=2.5 in c:\programdata\ana
conda3\lib\site-packages (from requests->folium) (2.6)
Requirement already satisfied: urllib3<1.25,>=1.21.1 in c:\programd
ata\anaconda3\lib\site-packages (from requests->folium) (1.22)
Requirement already satisfied: chardet<3.1.0,>=3.0.2 in c:\programd
ata\anaconda3\lib\site-packages (from requests->folium) (3.0.4)
Requirement already satisfied: certifi>=2017.4.17 in c:\programdata
\anaconda3\lib\site-packages (from requests->folium) (2018.1.18)
Requirement already satisfied: MarkupSafe>=0.23 in c:\programdata\a
naconda3\lib\site-packages (from jinja2->folium) (1.0)
```

FIGURE 22.5 Screen shot of installation of Folium using pip on Windows. The modules needed for installation of folium are marked with black boxes

Note that the output also indicates the other packages like branca, jinja2, etc. which are required for folium to run. Also note that for some of the modules, the minimum and maximum versions are also specified. So if you have an installation problem, do check out if all the dependencies of Folium (along with version requirements) are installed before you install folium.

22.3.2 Map Class

Before you study the Map class (note the capital M indicating that Map is a class and not a function/method). You may write a small script to create an interactive map on Jupyter notebook. The coordinates of Kanyakumari are: 8.0883° N, 77.5385° E (you can google it)

```
1  import folium
2  # Kanyakumari latitude = 8.0883, longitude = 77.5385
3  map_osm = folium.Map(location=[8.0883, 77.5385])
4  map_osm
```

Line 1: It is a simple import.

Line 2: Here you may create a Map object and name it map_osm. Note the use of osm in map_osm. OSM stands for Open Street Map[4]. It is important to understand that the Map class of folium creates a "base map" which is created from a tile set. The default tile set is OSM. Other tile sets which can be "passed to the Map constructor" are discussed later.

[4] See: https://www.openstreetmap.org/

The script and output on Jupyter is shown in Figure 22.6.

FIGURE 22.6 Display of Kanyakumari, latitude = 8.0883 and longitude = 77.5385, on Jupyter

You can get details of the Map class by using '?' on Jupyter as shown in Figure 22.7.

```
In [23]: 1 ?folium.Map
```

FIGURE 22.7 Use the question mark (?) to details of a class or method on Jupyter

The Map class of Folium takes a very large number of parameters. The complete list of parameters of Map class are:

```
folium.Map(location=None, width='100%', height='100%', left='0%', top='0%',
position='relative', tiles='OpenStreetMap', API_key=None, max_zoom=18, min_zoom=0,
max_native_zoom=None, zoom_start=10, world_copy_jump=False, no_wrap=False, attr=None,
min_lat=-90, max_lat=90, min_lon=-180, max_lon=180, max_bounds=False,
detect_retina=False, crs='EPSG3857', control_scale=False, prefer_canvas=False,
no_touch=False, disable_3d=False, subdomains='abc', png_enabled=False,
zoom_control=True)
```

While the complete "signature" of the Map class might look daunting, it actually is very easy to use because most parameters have "default values" and therefore can be ignored.

The one parameter that needs to be provided is location. Some other parameters which one should know are as follows:

Parameter	Description in the docstring/Explanation
location	You can give a tuple or list. Default value is None In the list/ tuple first latitude (Northing) and then longitude (Easting) has to be given.
width	You may give input as (1)pixel int or (2) percentage string (default: '100%'). It gives the width of the map.
height	You may give input as (1) pixel int or (2) percentage string (default: '100%'). It gives the height of the map.

tiles	You have to give a string, i.e., str parameter. The default value is *'OpenStreetMap'* You have three options for the map tileset to use: - Choose from a list of built-in tiles, - pass a custom URL - or pass `None` to create a map without tiles.
API_key:	This parameter has to be type string, i.e., str. Default is None. Some tiles like OSM don't require API_key. But others like Cloudmade and Mapbox need API_key
max_zoom:	This parameter is of type integer, i.e., int. Default value is 18.It gives the maximum zoom for the map.
zoom_start:	This parameter is of type integer, i.e., int. Default value is 10. It gives the initial zoom level for the map.
control_scale:	This parameter is of type bool. Default value is False. If this parameter is True, a control scale is added on the map, but if False then control scale is not added.

Note that for the "tiles" parameter, you can give a value from the following box (taken from the doc string of Map class):

```
You can give any of the following for the "tiles" keyword:
    - "OpenStreetMap"
    - "Mapbox Bright" (Limited levels of zoom for free tiles)
    - "Mapbox Control Room" (Limited levels of zoom for free tiles)
    - "Stamen" (Terrain, Toner, and Watercolor)
    - "Cloudmade" (Must pass API key)
    - "Mapbox" (Must pass API key)
    - "CartoDB" (positron and dark_matter)
You can pass a custom tileset to Folium by passing a Leaflet-style
URL to the tiles parameter: ``http://{s}.yourtiles.com/{z}/{x}/{y}.png``
```

The following script creates a "Stamen Watercolor" map and give it an initial zoom of 13.
The code is as follows:

```
1 import folium
2 myLoc = [18.9220, 72.8347] #Location for Gateway of India
3 map_2 = folium.Map(location= myLoc,
4                    tiles='Stamen Watercolor',
5                    zoom_start=13)
6 map_2
```

Line 2: These are the coordinates of Gateway of India from Google search.

Line 4: 'Stamen Watercolor' given as value to the parameter tiles.

Line 5: An initial zoom of 13 has been given

The output on Jupyter is shown in Figure 22.8.

FIGURE 22.8 Stamen Watercolor map of Mumbai and give it an initial zoom of 13

22.3.3 Adding a Marker to the Map

Now you may add a Marker to your map. You can do this by using:

```
folium.Marker(location, popup, icon).add_to(mapName)
```

Here Marker is a method with following three parameters.

1	location: This parameter must be of type tuple or list. Default value is None. Latitude, i.e., Northing must come first followed by Longitude, i.e., Easting of Marker.
2	popup: This parameter must be of type either string or folium.Popup. Its default value is None. Input text or visualization for object.
3	icon: Icon plugin. the Icon plugin to use to render the marker.

The following script builds upon the previous example and puts a popup on the gateway of India. The script also adds a popup which will give the latitude/longitude of a point clicked on the map. For this folium.LatLngPopup() is used.

The code is as follows:

```
import folium
myLoc = [18.9220, 72.8347] #Location for Gateway of India
map_2 = folium.Map(location= myLoc,
                   tiles='Stamen Watercolor',
                   zoom_start=13)
# Add a marker at myLoc
folium.Marker(location = myLoc, popup = 'GatewayOfIndia').add_to(map_2)
# Add a popup fiving latitude and longitude of point clicked
map_2.add_child(folium.LatLngPopup())
map_2
```

The output on Jupyter is shown in Figure 22.9.

FIGURE 22.9 Adding a 'Marker' at Gateway of India. Marker displays a string 'GatewayOfIndia' when clicked. It also shows generation of a 'Popup' giving latitude and longitude, when a point on the map is clicked.

Explanation:

Line 7: Here you have added a Marker to the map by using Marker() method and add_to(mapName) method.

Line 9: Here you have added a LatLngPopup() to the map by using the add_child() method.

Folium also provides for a ClickForMarker(popup) method which will plot a popup at each point which is clicked on the map.

This is shown by adding a line of code as follows:

```
1  import folium
2  myLoc = [18.9220, 72.8347] #Loc for Gateway of India
3  map_2 = folium.Map(location= myLoc,
4                     tiles='Stamen Watercolor',
5                     zoom_start=13)
6  folium.Marker(location = myLoc, popup = 'GatewayOfIndia').add_to(map_2)
7  map_2.add_child(folium.LatLngPopup())
8  map_2.add_child(folium.ClickForMarker(popup = 'X'))
9  map_2
```

Line 8: This line has been added. It will create a popup at each point where the map is clicked. Also if you click on this newly created popup you will get an 'X'.

22.3.4 Using Folium to Generate Simple Map of Pandas Data

This script shows how to use the pandas library to read some data and plot it on a map.

For this example, data of Indian cities were downloaded from: https://simplemaps.com/data/in-cities in csv format[5]. If you see the data, it has list of 213 cities of India with the following columns:

city	lat	lng	country	iso2	admin	capital	population	population_proper

For this exercise, you may divide cities into following categories by population:

	Population (p)	Color of popup
1	p > 2000000	Black
2	2000000 >= p > 1000000	Blue
3	1000000>= p > 500000	Red
4	500000 <= p	Green

```
1  import folium
2  import pandas as pd
3
4  bhopal = [23.2599, 77.4126]
5  map_ind = folium.Map(bhopal, zoom_start = 5)
6
7  cities_in = pd.read_csv(r'C:\Users\DS\Downloads\in.csv')
8  for index, row in cities_in.iterrows():
9      lat = row['lat']
10     lng = row['lng']
11     city_name = row['city']
12     city_pop = row['population']
13
14     if city_pop > 2000000: # Black for more than 10 lakh population
15         pop_color = 'black'
16     elif city_pop <= 2000000 and city_pop > 1000000: #Blue for 10 to 20 lakh
17         pop_color = 'blue'
18     elif city_pop <= 1000000 and city_pop > 500000:# Red for 5 to 10 lakh
19         pop_color = 'red'
20     else:                    # Green for less than 5 lakh
21         pop_color = 'green'
22     folium.Marker(location = [lat, lng], popup= city_name,
23                   icon =folium.Icon(color = pop_color)).add_to(map_ind)
24
25 map_ind
```

Explanation:

Line 7: read_csv() reads a CSV (comma-separated) file into DataFrame and returns a DataFrame object.

Line 8: DataFrame.iterrows(): This function, iterates over DataFrame rows as (index, row) pairs. The variable index will contain the row number and the row variable will hold the entire row for that index.

[5] The website clearly mentions that you may use the data for personal or commercial purpose. But as a courtesy to the owner, you may like to give proper citation if you use this data in any publication or project.

Line 9-12: You can get the individual cell entry for each row by specifying the column name. Here you need the latitude, longitude, city name and city population.

Line 14-21: This is simply an if-elif-else block to give appropriate color value to the pop_color variable depending upon the population of the city.

Line 22-23: This is the method to add the Marker to the map. Note that the popup parameter contains the name of the city. So if you click any popup on the map, you will get the city name. Also the icon has a color parameter whose value depends on the population of the city.

Output on Jupyter is shown in Figure 22.10.

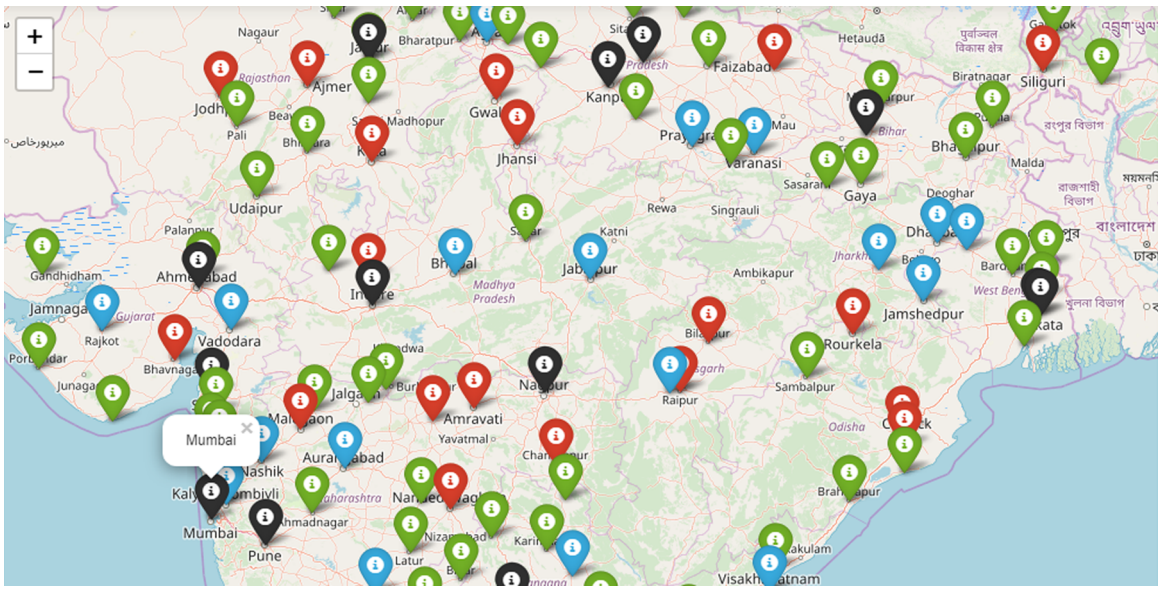


FIGURE 22.10 Plotting of Pandas DataFrame on a Folium map. The cities are marked in markers of different colours depending upon their population. Clicking the marker also gives the name of the city.

22.4 GeoJSON

The next exercise is to overlay a GeoJSON file over a folium map. But before you can do this, you need to understand what a GeoJSON file is. This section covers some basics of GeoJSON and how to use this format for plotting maps.

22.4.1 GeoJSON basics

The json format has already been discussed. GeoJSON format is a kind of JSON for geospatial data interchange. GeoJSON describes several "types" of json objects.

A GeoJSON "object" could be one of the following three types:

- A region of space (i.e., a geometry). Some examples are:
  (1) With 0-dimensional, i.e., (a) Point, and (b) MultiPoint;
  (2) With one-dimensional curve (a) LineString, and (b) MultiLineString;
  (3) With two-dimensional surface (a) Polygon, and (b) MultiPolygon, and finally

(4) GeometryCollection which is "collection of Geometry objects." (So a GeometryCollection can contain other Geometry objects)

- Features in GeoJSON are geometric objects with "additional properties". For example, a simple Point is a "Geometry", but a Point with say "address" becomes a GeoJSON Feature. Similarly, a LineString may be a Geometry, but a LineString with name could be a road, etc.
- FeatureCollection. It is a "list of Features".

Instead of enumerating all the different types of GeoJson objects that can exist, you may like to consider concrete examples of a few common types of GeoJson objects.

For example, a GeoJson object of type Point and type Polygon are shown in Table 22.1.

TABLE 22.1 "Point" Geometry or a "Polygon" Geometry represented in GeoJSON

{
"type": "Point",
"coordinates": [100.0, 0.0]
}
	{
	"type": "Polygon",
	"coordinates": [
	[
	[100.0, 0.0],
	[101.0, 0.0],
	[101.0, 1.0],
	[100.0, 1.0],
	[100.0, 0.0]
	]
	]
	}

Point coordinates are in x, y order so that longitude (i.e. Easting) comes first as x-coordinate and latitude (i.e., Northing) comes next as y-coordinate.	Another common example of GeoJson is a polygon.

So GeoJSON is just a "type" of json. It is not possible to discuss GeoJSON in details here. But it should be noted that GeoJSON is not the only file format for mapping. There are at least two other file formats which are important. They are Shapefile and KML. They are not discussed here.

22.4.2 Overlaying a GeoJSON file over a folium map

You can impose a GeoJSON file on a folium map. For this example, a json file of India available here[6] has been downloaded named as india.json. The code for creating a layer on folium is as follows:

```
1 import folium
2 # Start from Bhopal since it is close to center of India.
3 bhopal = [23.2599, 77.4126]
4 map_ind = folium.Map(bhopal, zoom_start = 5)
5 folium.GeoJson(r'C:\data_geojson\india.json').add_to(map_ind)
6 map_ind
```

The output on Jupyter is shown in Figure 22.11.

[6] See: https://raw.githubusercontent.com/johan/world.geo.json/master/countries/IND.geo.json

FIGURE 22.11 Overlaying a GeoJSON file of outline of India on a Folium map.

22.4.3 Working with geojson Files Using geojson Library

GeoJson file type in Python were briefly discussed. But note that geojson is not only a file type but also a library in Python. You can install geojson on your Jupyter using:

```
1 !pip install geojson
```

In this exercise, you will download a geojson file from the internet and study it. The file that is chosen for this exercise is file us-states.json available here[7]. You should download this file to work along this example and save it as <file_name.json>. On author's machine, the output is shown in Figure 22.12.

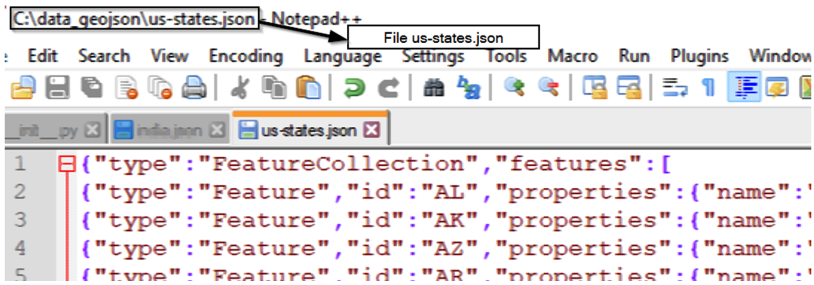


FIGURE 22.12 Screen shot of us-states.json file on Notepad++.

[7] See: https://raw.githubusercontent.com/python-visualization/folium/master/examples/data/us-states.json

Copy the entire contents of the page (control-a, to select all and control-c to copy on windows) and paste in a text editor (I use notepad++). Save the file with extension json (Not geojson, since geojson is a file of type json only)

If you see this file, i.e., us-states.json, the json file looks like a Python data type dictionary whose:

- First key is "type" with value "FeatureCollection".
- Second key is "features" with value which is a list of dictionaries like: {"type":"Feature","id":"AL","properties":{"name":"Alabama"},"geometry":{"type":"Polygon","coordinates":[[[..... . So the second key has a list of dictionaries.

You can always use an online json viewer to get an idea of the tree structure (or rather the inverted tree) and the structure of a json file. One such online service[8] to get the tree view is shown in Figure 22.13.

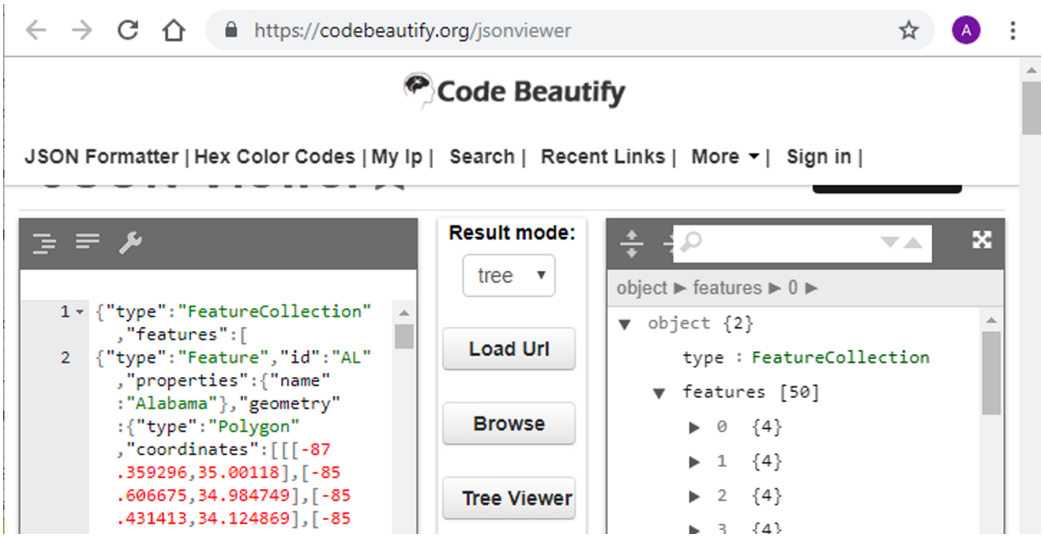


FIGURE 22.13 Screen shot of an online json file structure viewer

So you can open a json file and extract values for keys just like any dictionary in Python.

Now you can write a small script which can extract the two-letter abbreviations for US States. Have a look at the following code:

```
import geojson
import csv
fName = r'C:\data_geojson\us-states.json' # Give your source file name
fCsv = r'C:\data_geojson\us_states_names.csv'# Give your destination file
with open(fName) as f:
    fJson = geojson.load(f)
    with open(fCsv, 'w', newline = '', encoding = 'utf8') as fHandle:
        fWrite = csv.writer(fHandle)
        for eachItm in fJson['features']:
            print(eachItm['id']) # Prints the 2 letter abbreviations of US States
            fWrite.writerow([eachItm['id']]) # Writes them to a .csv file
```

[8] See: https://codebeautify.org/jsonviewer

Example:

Line 6: Here the load() method of geojson library is used to open the file as a geojson object. The json data is loaded in a variable named fJson.

Line 9: Now fJson is a geojson object and fJson['features'] represents each of the inner dictionary entries.

Line 10: Now this inner dictionary has four keys → 'type', 'id', 'properties' and 'geometry'. The two-letter abbreviation for US States has a key of "id". Note that the value for the key → 'geometry' is itself a dictionary.

Line 11: This writes each item row by row to the csv file. Note the extra square brackets in the writerrow() method. If you don't put it, the csv library will put each character in a different column. So if you don't put this extra square bracket, a state name like AK will be stored as A,K, i.e., an extra comma will be inserted between them.

So the above script shows how you can "extract" data from GeoJSON file. The .cs file created here will be used in the next exercise, so keep it saved.

22.5 CHOROPLETH MAPS

A choropleth map is a map which will differentially color different areas of a map based on some indicator. So you need at least two data files to plot a choropleth map. One file must be a .json file containing GeoJSON data. The other file must have some data to "categorize" the "polygons" or the States created by the GeoJSON. This is why a .csv file was created from the original GeoJSON file.

The two files used for this exercise are as follows:

- us-sates.json
- us_states_names.csv

You need to modify the .csv file to include a row for headers. You should also add a column with header values and give it some integer values. You need to add a "Header" because you need to access the values in a column and for that you need a header. Further you need a value to "Categorize" the data. In author's case, an arbitrary value from 1 to 50 has been taken for the 50 states of the US. You can take any data by which you want to categorize. You could divide states into small, medium and big. You could also categorize states based on their population or production of some item, etc. In this sample exercise, the states have simply been given a number from 1 to 50 and this number is used to create different "color" for each state. In author's case, a value from 1 to 51 was given as shown in Figure 22.14.

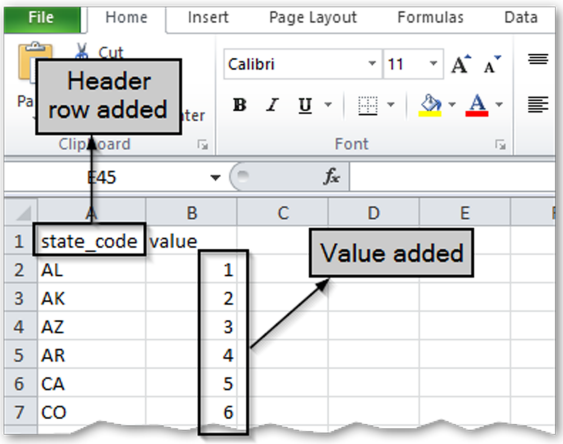


FIGURE 22.14 Modification of .csv file by adding a "Header row" and also by adding a column giving some number as data.

The method used is choropleth(), whose signature is (taken from the docstring of the choropleth() method) as follows:

```
choropleth(geo_data, data=None, columns=None, key_on=None, threshold_scale=None, fill_
color='blue', fill_opacity=0.6, line_color='black', line_weight=1,
line_opacity=1, name=None, legend_name='', topojson=None, reset=False,
smooth_factor=None, highlight=None)
```

Most of the above parameters have default values and can be ignored for most purposes. The description of some relevant parameters, used in next example is shown in Table 22.2.

TABLE 22.2 Explanation on how to use some important parameters of the choropleth() method

1	**geo_data** (*string*) - Here you give the "file path" in form of a string of the GeoJSON file to be plotted.
2	**data** (*Pandas DataFrame*) - Here you give a Pandas dataFrame.
3	**columns** (*list*) - This parameter must specify 2 column names of the DataFrame you provided to the "data" parameter. Must pass first column as the key, and second column as the values. Note that 2 column names of the Pandas dataFrame used in the parameter "data" have to be given here. The first column name must match with the parameter "key_on" and the second column is the data of "values" on which the choropleth map is drawn or the polygons are "colored".
4	**key_on** (*string, default None*) - Variable in the GeoJSON file to bind the data to. Must always start with 'feature' and be in JavaScript objection notation. Ex: 'feature.id' or 'feature.properties.statename'. Note that this "variable" of the GeoJSON file must bind with the first column of the columns parameter.
5	**fill_color** (*string, default 'blue'*) - Area fill color. Can pass a hex code, color name, or if you are binding data, one of the following color brewer palettes: 'BuGn', 'BuPu', 'GnBu', 'OrRd', 'PuBu', 'PuBuGn', 'PuRd', 'RdPu', 'YlGn', 'YlGnBu', 'YlOrBr', and 'YlOrRd'.
6	**fill_opacity** (*float, default 0.6*) - Area fill opacity, range 0-1.
7	**line_opacity** (*float, default 1*) - GeoJSON geopath line opacity, range 0-1.
8	**legend_name** (*string, default empty string*) - Title for data legend.

Table 22.2 gives explanation on how to use some of the important parameters of the choropleth() method.

Some important things to remember are:

- You need two datasets to plot a choropleth map.
- The first data set has to be a GeoJSON data in .json format.
- The second data set has to be a DataFrame object (of Pandas module).
- There has to be a "common field" between the two data sets.
- Based on the "common field" of the two data sets, the second column of the Pandas DataSet is used to "color" the "polygons" created by the GeoJSON data set.

The following script creates a choropleth map using two data sets. The first is a GeoJSON file named us-states.json. The second is a .csv file. Note that the two letter codes of US States was "extracted" from the GeoJSON file and kept in a .csv file. Then a "header" and a "second column" was added to the .csv file.

The script to generate choropleth map is as follows:

```
# Import libraries
import pandas as pd
import folium
# Path to geojson file with us states
fJson = r'C:\data_geojson\us-states.json' # Give your source .json file path
#Load the .csv file with data for each state in a pandas dataframe
fCsv = r'C:\data_geojson\us_states_names.csv'# Give your source .csv file path
# Create a DataFrame object from the .csv file
stateData = pd.read_csv(fCsv)
# Initialize the map somewhere in middle of USA
m = folium.Map(location=[37, -102], zoom_start=3)
# Use choropleth method:
m.choropleth(geo_data=fJson,         # fJson is the path to the .json file
             name='choropleth',
             data= stateData,        # stateData is a DataFrame from .csv file
             columns=['state_code', 'value'], # 2 column headers of DataFrame
             key_on='feature.id',    # 'feature.id' of .json file must match
                                     # 'state_code' column of DataFrame
             fill_color='OrRd',      # 'OrRd' is for Orange (Or) to Red (Rd)
             fill_opacity=0.7,
             line_opacity=0.5,
             legend_name='RandomValues')  # Give some name to the legend
folium.LayerControl().add_to(m)
#Display final map
m
```

The output on Jupyter is shown in Figure 22.15.

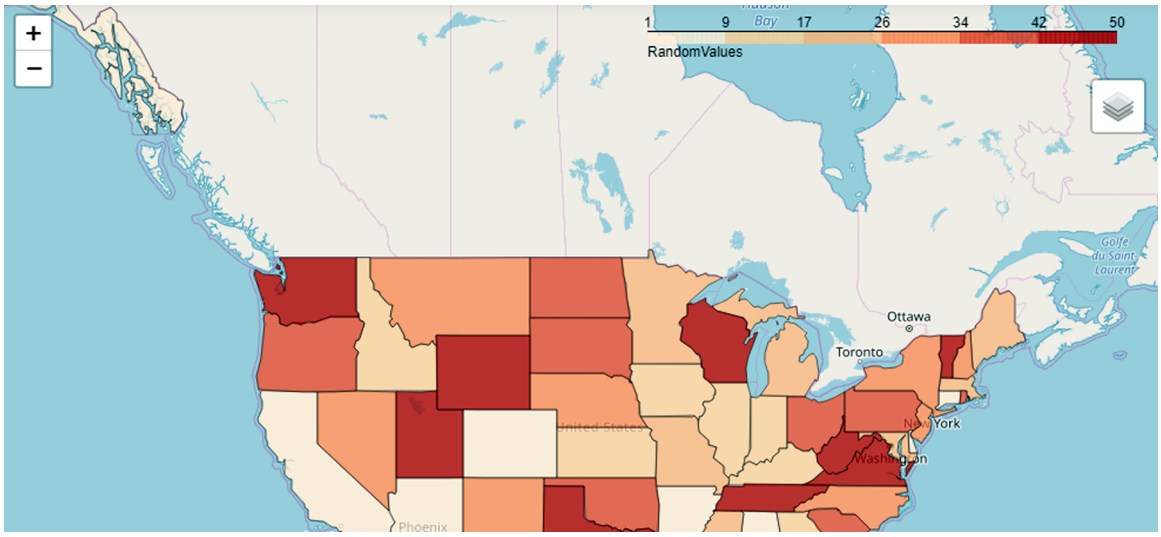


FIGURE 22.15 Screen shot of choropleth map of US States. The states are colored on scale from 1 to 50.

Conceptual Questions

1. What is the ".whl" format?
2. When creating an object of the Basemap class, you may give the "bounding box" in two different formats. What are they?
3. What are "leaflet" and "folium"?
4. When using the Map class of the Folium module, you have to give three parameters namely "location", "width" and "height" to describe the "bounding box" of the map. Explain the use of these three parameters. Further the "location" parameter is a list of two numbers. What are these numbers? Why are they also called Easting and Northing?
5. The Map class of Folium has a parameter "tiles". What is it used for? Which tiles does the Map class use as default?
6. Folium has a Marker class. What does it do? Further the constructor to the Marker class takes two important parameters namely: "location, " and "popup". Explain their use.
7. The Map class constructor of Folium has a parameter "API_key". Which tile sets need it? Does OSM need an API_key?
8. "The Map class of folium creates a "base map" which is created from a tile set. The default tile set is OSM." What does this mean?
9. "GeoJSON is a JSON based geospatial data interchange format". Explain
10. One type of GeoJSON object is what is called "Geometry". What is a "Geometry" object?
11. Which are zero-dimensional geometry objects?
12. Which are one-dimensional and two-dimensional geometry objects?
13. Another type of geometry object is a "GeometryCollection". How is a GeometryCollection object different from other Geometry objects?
14. In GeoJSON, what is the difference between a Geometry object and a Feature object?
15. What are "choropleth maps"?
16. The choropleth() method of Folium has four important parameters namely: "geo_data", "data", "columns" and "key_on". Explain the type of data each one of them expects.

Beyond Text

1. Cartopy library

It was mentioned in the chapter that the Basemap library may be ultimately replaced by a new project called Cartopy. This section discusses the Cartopy library.

Note on Installation of Cartopy through conda and its channels.

Most Python libraries can be installed either by the pip command or by using the "conda" command on the command prompt. However many "mapping" libraries may create "version issues" during installation. In fact the Basemap library did create such issues and that is why it was installed via the .whl binaries.

This section deals with using conda for installation. The following points are noteworthy:

- Conda is distributed with Anaconda, but it is actually a package manager.

- You can use conda for (1) installation, (2) updating, and (3) removal of packages.
- One important difference between conda and pip is that pip is for installing only Python packages, while conda can install packages written in other languages also.
- Further note that conda is a product of a company called Continuum Inc.
- Further note that the way conda works is that it provides a number of packages to users so that users can download these packages free of cost. For providing these packages, conda uses "channels". A "channel" is a place or location which conda looks for a package.
- Another thing to understand is that there can be a "default channel" or some "other channel". A "default channel" is the one created by the company Continuum. On the other hand "other channels" are generally those created by open source community.
- The point to note is that a particular package that you are looking for may be available at the "default channel" or at the so called "other channel" or sometimes at both places. One of the most popular "other channel" is called **conda-forge** and it is totally driven by the open source community.
- So the question arises is that why have "other channels"? The answer is that while conda is a "company driven project", conda-forge is a community driven project. Moreover, some of the projects on conda-forge may be more "up to date" than on conda default.

The "default channel" does not have any name. So if you want to use the "default" channel, the command is:

```
$conda install package_name
```

However if you want to use the a channel say channel_name, then you have to give command with a –c option as well as the channel_name

```
$conda install -c channel_name package_name
```

So if you want to use the channel conda-forge, then the instead of channel_name, use conda-forge and the command will be:

```
$conda install -c conda-forge package_name
```

It is not possible to do a detailed discussion on Cartopy library. This section gives a basic introduction with some example code to put you on the path.

The basic module of Cartopy for projections is cartopy.crs which is conventionally imported as ccrs. Cartopy has a number of projections, some of which are as follows:

Cartopy projection list: PlateCarree, LambertConformal, LambertCylindrical, Mercator, Miller, Mollweide, Orthographic, Robinson, Stereographic, TransverseMercator, InterruptedGoodeHomolosine, RotatedPole, OSGB, EuroPP, Geostationary, Gnomonic, NorthPolarStereo, OSNI, SouthPolarStereo

The following example code shows how Cartopy is used.

The steps are as follows:

- Import cartopy.crs as ccrs.
- For this example, the AzimuthalEquidistant class is used. So create an instance of this class (each projection has a class of its own. If you want to use a different projection, then accordingly use appropriate class).

- The signature of the AzimuthalEquidistant class is as follows:

```
# On Jupyter
import cartopy.crs as ccrs
?ccrs.AzimuthalEquidistant
```

```
# OUTPUT (Has been modified and truncated)
Init signature: ccrs.AzimuthalEquidistant(central_longitude=0.0,
central_latitude=0.0, false_easting=0.0, false_northing=0.0, globe=None)
Docstring: An Azimuthal Equidistant projection
This projection provides accurate angles about and distances through the
central position. Other angles, distances, or areas may be distorted.
Init docstring:
Parameters
   (1) central_longitude: optional. The true longitude of the central meridian in
       degrees. Defaults to 0.
   (2) central_latitude: optional. The true latitude of the planar origin in
       degrees. Defaults to 0.
   (3) false_easting: optional. X offset from the planar origin in metres.
       Defaults to 0.
   (4) false_northing: optional. Y offset from the planar origin in metres.
       Defaults to 0.
   (5) globe: optional. An instance of :class:`cartopy.crs.Globe`. If omitted, a
       default globe is created.
```

The following script shows how Cartopy may be used (on Jupyter). Note that the five methods used are (1) ax.stock_img() (2) ax.coastline() (3) ax.gridlines() (4) ax.set_extent() and (5) ax.set_title(). These are all methods of the GeoAxes class which is explained later on.

```
%matplotlib inline
# crs is conventionally imported as ccrs
import cartopy.crs as ccrs
import matplotlib.pyplot as plt
# 82.5 E for Allahabad, 23.5 N for Tropic of cancer
# Create an instance of AzimuthalEquidistant class
my_proj = ccrs.AzimuthalEquidistant(central_longitude = 82.5,
                                    central_latitude = 23.5)
ax = plt.axes(projection= my_proj)
ax.stock_img()
ax.coastlines()
ax.gridlines()
# bbox covers India.It is [West, East, South, North]
bbox = [60, 100, 5, 40]
ax.set_extent(bbox)
ax.set_title('AzimuthalEquidistant')
# The type of ax is GeoAxesSubplot
print(type(ax))
```

The output on Jupyter is shown in Figure 22.16.

```
<class 'cartopy.mpl.geoaxes.GeoAxesSubplot'>
```

AzimuthalEquidistant

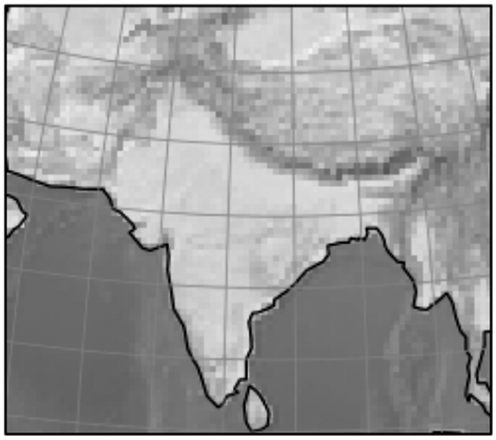


FIGURE 22.16 Output on Jupyter of script using the Cartopy mapping application.

A few relevant points are as follows:

- Note that when you pass a class of the crs module (like you passed the AzimuthalEquidistant class), then the object created is an object of class GeoAxes. This class GeoAxes "sub-classes" matplotlib.axes.Axes class. So you can say that Cartopy classes "extend" or "sub-class" or "inherit from" the Axes class of matplotlib. It is instructive to see the source code of this class[9]. Screen shot of relevant portion of file geoaxes.py (available on github) is shown in Figure 22.17.

```
277  class GeoAxes(matplotlib.axes.Axes):
278      """
279      A subclass of :class:`matplotlib.axes.Axes` which represents a
280      map :class:`~cartopy.crs.Projection`.
281
282      This class replaces the Matplotlib :class:`~matplotlib.axes.Axes` class
283      when created with the *projection* keyword. For example::
284
```

The class GeoAxes subclasses the class Axes

FIGURE 22.17 Relevant portion of file geoaxes.py. You can see that GeoAxes class inherits from the class Axes which is at matplotlib.axes.Axes

As pointed out earlier, the above script uses the following methods of the GeoAxes class: (1) ax.stock_img() (2) ax.coastline() (3) ax.gridlines() (4) ax.set_extent() and (5) ax.set_title(). You can see the signatures of these methods by either of the following two methods:

- See the source code of cartopy/mpl/geoaxes.py (source code is available on your local machine as well on github).
- You can see the signature on Jupyter also.

[9] Source code of GeoAxes class is available at: https://github.com/SciTools/cartopy/blob/master/lib/cartopy/mpl/geoaxes.py

As an example, the signature of the method coastlines() of class GeoAxes is as follows:

```
# On Jupyter
from cartopy.mpl.geoaxes import GeoAxes
?GeoAxes.coastlines
```

```
# OUTPUT (Has been modified and truncated)
Signature: GeoAxes.coastlines(self, resolution='110m', color='black', **kwargs)
Docstring: Add coastal **outlines** to the current axes from the Natural Earth
"coastline" shapefile collection.

Parameters
    (1) Resolution:- A named resolution to use from the Natural Earth dataset.
        Currently can be one of "110m", "50m", and "10m".
```

Similarly you can examine the docstrings of other methods also. Do pay attention to the import statement. In the import statement, the GeoAxes class was imported from the geoaxes module which is in file geoaxes.py

2. Basics of Spatial Data Model and Shapely library

Note: This discussion focuses more on getting an "insight" into the Spatial Data Model and its usage rather than a formal course.

When you plot geospatial data, you need what are called "geometric objects". There are basically 3 types of "geometric objects", namely (1) points, (2) curves, and (3) surfaces.

Further every "geometric object" divides any space into 3 parts: (1) interior, (2) boundary, and (3) exterior. This means that any "geometric object" will divide a space into 3 sets of points namely: (1) a set of points in the interior of the geometric object, (2) a set of points on the boundary of the geometric object, and (3) a set of points exterior to the geometric object.

Some noteworthy aspects are:

- the "boundary" divides the interior from the exterior.
- it is possible that for a given shape a particular set of points may be an "empty set". For example a "point" geometric object will have 1 point in its interior but no point on its "boundary" and infinite points to its exterior.
- a "curved line" is presumed to be made up of multiple straight lines joined together end to end, or we can say that a "curve" may be "approximated" by a number of straight lines joined end to end.
- for a "curve" the two "end-points" of the curve are considered to be their boundary.
- for a "curve" all the points in the curve excluding the end-points are considered to be "interior" to the curve and all other points are considered to be exterior to the curve. This might appear a bit different but it is important to understand that for a curve, the interior is set of all such points in the curve except the two terminating points.
- a surface may be represented by a "closed set" of coordinates. Such a geometry is called a "linear ring".
- by convention, a linear ring is represented in counter clock wise direction as seen from the "top".
- If a surface has a "hole" in it, then it is represented by a Polygon. So a Polygon has 2 parameters, the first to specify the "outer linear ring" and the other to specify the "inner hole". (Note this

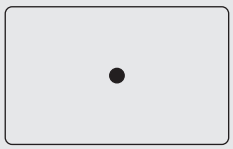
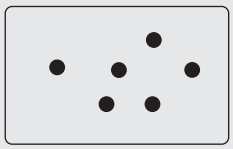
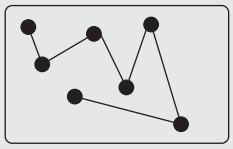
discussion is based on how geometries are represented in Shapely library. The definitions/ discussion may not be true for other libraries.).

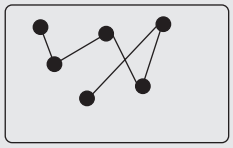
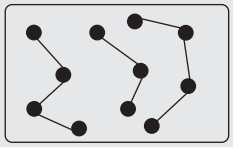
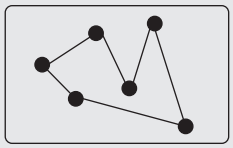
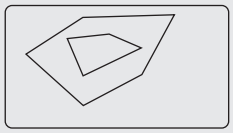
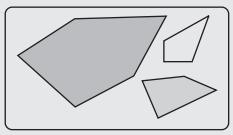
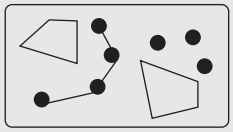
- So the difference between a "linear ring" and a polygon is that a linear ring is a surface without any "hole", while a polygon may have a "hole" in it. Note that the way the Shapely library is implemented, the linear ring and hole may touch at one point only and must not intersect. Should the "linear ring" and the "hole" intersect, an error will be thrown.

The following table summarizes the characteristics of the 3 basic geometries:

Geometry	Interior	Boundary	Exterior	Dimension
Point	Has exactly 1 point as interior	Has exactly 0 point on its boundary	All other points excluding the point form the exterior	Has a topological dimension of 0
Curve	All the points making up the curve excluding the two end points are the interior of a curve	Has exactly 2 points as its boundary. These 2 points are the 2 terminating points	All other points not included in the curve form the exterior of the curve	Has a topological dimension of 1
Surface	All the points lying 'inside' the surface, that is all points of the surface excluding the points forming the edges of the surface	All the points forming the edges of the surface. Note that the boundary of a surface will be a curve.	All other points not lying on the surface.	Has a topological dimension of 2

The above discussion relates to the "geometries". Coming back to the "implementation" of these geometries in a library, the Shapely library implements the geometries in form of different types of geometric objects. These objects may be "individual" geometries or a "collection" of geometries. The following table summarizes the geometries and the "Python classes" used to represent them.

S.No	Geometry	Shapely Class	Example
1	A single point	Point	
2	A **collection** of points	MultiPoint	
3	A single line which does not intersect itself.	LineString (It is also called a **simple** LineString)	

4	A single line (Which intersects itself)	LineString. (It is also called a **complex** LineString)	
5	A collection of lines	MultiLineString	
6	A Linear ring (It is a closed figure)	LinearRing	
7	A Surface with a hole in it. The outer ring is just like a linear ring. The inner "hole" is optional i.e. it may or may not be present. If the inner "hole" is absent, then the polygon is just like a linear ring.	Polygon	
8	A collection of surfaces	MultiPolygon	
9	A collection of various geometries	GeometryCollection	

Some other noteworthy points regarding geometries in Shapely are as follows:

- A Point object has 0 area and 0 length.
- A LineString object is allowed to cross itself. If it crosses itself, it is a complex LineString, else it is a simple LineString object.
- A LineString object has 0 area but non-zero length.
- A LineRing may be "explicitly closed". This will happen if the last point is the same as the first point. However there may be a case where the last point is not same as initial point. In such cases, the first point is "implicitly copied" as the last point, thereby making it into a closed object.
- A LineRing has 0 area but non-zero length. The fact that a LineRing has 0 area may appear confusing, but the area of a LineRing is the area occupied by the ring and not the area "enclosed" by the ring.

- A Polygon can have a "hole" in it. But the "hole" may "touch" the outer enclosing LineRing at most 1 point. So the "exterior" and "interior" rings of a Polygon object cannot touch along a line. Nor can they "intersect" each other.
- The "hole" in a Polygon object is optional. So you can have a Polygon object without a "hole" in it also.
- A Polygon object has non-zero area and non-zero length.

The following example code explains how Point, LineString, LinearRing and Polygon objects are created and used in Shapely:

```
from shapely.geometry import Point, LineString, LinearRing, Polygon
# Point object
p1 = Point(0.0, 0.0)
print('p1->', p1)
print('area of p1->', p1.area)  # Point object has 0 area
print('length p1->', p1.length)  # Point object has 0 length
# LineString objects
line1 = LineString([(0, 0), (3, 4)])
print('line1->', line1)
print('length line1->', line1.length)
print('area line1->', line1.area)
# Explicitly closed LinearRing of 4 points. First and last Point are same
ring1 = LinearRing([(0, 0), (2, 3), (4, 5), (0, 0)])
print('ring1->', ring1)
print('length ring1->', ring1.length)
print('area ring1->', ring1.area)
# Implicitly closed LinearRing. Last point not same as first
ring2 = LinearRing([(0, 0), (2, 3), (4, 5), (0, 0)])
print('ring2->', ring2)
# Polygon. It will also implicitly close if last point not same as first
poly1 = Polygon([(0, 0), (1, 1), (2, 0)])
print('poly1->', poly1)
print('length poly1->', poly1.length)
print('area poly1->', poly1.area)
```

```
# OUTPUT - - -
p1-> POINT (0 0)
area of p1-> 0.0
length p1-> 0.0
line1-> LINESTRING (0 0, 3 4)
length line1-> 5.0
area line1-> 0.0
ring1-> LINEARRING (0 0, 2 3, 4 5, 0 0)
length ring1-> 12.837102637643028
area ring1-> 0.0
ring2-> LINEARRING (0 0, 2 3, 4 5, 0 0)
poly1-> POLYGON ((0 0, 1 1, 2 0, 0 0))
length poly1-> 4.82842712474619
area poly1-> 1.0
```

3. Predicates of Shapely

The shapely module has 2 types of predicates namely (1) Unary and (2) Binary
The following table describes them:

	Property	Unary	Binary
1	Number of objects involved	Only 1 object is involved	2 objects are involved
2	Method of implementation	They are implemented as "attributes" of the single object. So if object_name is a Shapely geometric object and attribute_name is its some attribute, then the unary predicate is implemented as object_name.attribute_name	Binary predicates are implemented as methods. So if obj1 and obj2 are 2 Shapely geometric objects and some_method() is some method, then this binary predicate is implemented as obj1.some_method(obj2)
3	Example	Whether a Shapely geometric object has "area" or not is an example of its unary predicate	Whether two LineString objects intersect or not is an example of binary predicate of these 2 LineString objects

4. The DE-9IM Relationships of binary predicates

DE-9IM stands for Dimensionally Extended nine-Intersection Model (DE-9IM). It deals with binary (not unary) predicates.

The salient features of this model are:

- This model deals with the "relationships" of 2 geometric objects.
- Each of the 2 objects has 3 "regions" namely: (1) interior (can be represented by I), (2) boundary (can be represented by B), and (3) exterior (can be represented by E).
- The model deals with the "intersection" of each of the 3 regions of the 2 geometric objects.
- So when you "intersect" 3 regions each of 2 objects, then there are 9 possibilities.
- It is because of these 9 possibilities, that this model is called DE-9IM model.
- These 9 possibilities are represented in form of a 3×3 matrix.
- The intersection of 2 geometric objects may be True or False. A value of True indicates that there are some "common points " in the intersection and a false indicates that there are no common points in the intersection meaning that the two geometries are mutually exclusive.
- Further, if the intersection is True, then an intersection of 2 geometric objects may produce a point, a line or a surface. If it produces a point, then it is of dimension 0. If it produces a line, then it is of dimension 1 and if it produces a surface, then it is of dimension 2.
- So we can say that an "interaction" of 2 geometric objects will produce either a False (When no intersection happens) or a True which will have a dimension of either 0, 1 or 2.

Suppose you have 2 objects say X and Y. Then the DE-9IM matrix can be represented as follows:

		Object Y		
		Interior	Border	Exterior
Object X	Interior	Dim(I(X) ∩ I(Y))	Dim(I(X) ∩ B(Y))	Dim(I(X) ∩ E(Y)
	Border	Dim(B(X) ∩ I(Y))	Dim(B(X) ∩ B(Y))	Dim(B(X) ∩ E(Y))
	Exterior	Dim(E(X) ∩ I(Y))	Dim(E(X) ∩ B(Y))	Dim(E(X) ∩ E(Y))

This can be best understood by an example. Suppose X is a point X → (0, 0) and Y is also a point Y→ (1, 1). For the 2 points The matrix is as follows

		Object Y		
		Interior	Border	Exterior
Object X	Interior	Dim(I(X) ∩ I(Y)) → F	Dim(I(X) ∩ B(Y)) → F	Dim(I(X) ∩ E(Y) → 0
	Border	Dim(B(X) ∩ I(Y)) → F	Dim(B(X) ∩ B(Y)) → F	Dim(B(X) ∩ E(Y)) → F
	Exterior	Dim(E(X) ∩ I(Y)) → 0	Dim(E(X) ∩ B(Y)) → F	Dim(E(X) ∩ E(Y)) → 2

The above table/matrix has mostly False (F) values. However there are 3 entries which are not False. They are:

- Intersection of Interior of X with Exterior of Y, which will have point X in it (Because the Exterior of Y contains point X). So it is of Dimension 0.
- Intersection of Exterior of X with Interior of Y, which will have point Y in it (Because the Exterior of X contains point Y in it). So it is also of Dimension 0.
- The intersection of Exterior of X with Exterior of Y. This will produce a surface so, it will be of Dimension 2.

The DE-9IM model is implemented in Shapely by a method relate() which works as follows:

```
1  obj1.relate(obj2)
```

The following can be said about the method relate():

- In the above definition obj1 and obj2 can be any valid Shapely geometric objects like Point, LineString, LineRing, Polygon or their collection objects.
- The method returns a "string representation" of the "DE-9IM intersection matrix for the two geometries". This string representation is in form: <items_row1><items_row2><items_row3>

This will be clear from an example. The following script uses the relate() method to find the DE-9IM matrix of 2 points (0, 0) and (1, 1):

```
1  from shapely.geometry import Point
2  p1 = Point(0, 0)
3  p2 = Point(1, 1)
4  m1 = p1.relate(p2)
5  print('type of m1->', type(m1))  # m1 is a string
6  print('m1->', m1)  # You get FF0FFF0F2
```

```
7  # OUTPUT - - -
8  <class 'str'>
9  m1-> FF0FFF0F2
```

5. GeoPandas

GeoPandas project makes it very easy to work with Geo spatial data in Python.

GeoPandas depends in turn upon a number of other modules for various purposes. Briefly they are:

- For creating geometric shapes, it depends upon the module shapely.
- For storing the geometric data and other types of data, it "extends" the capabilities of 2 data structures of pandas namely: (1) Series, and (2) DataFrame. GeoPandas extends the Series

data structure of Pandas and creates GeoSeries data structure. Similarly GeoPandas extends the DataFrame data structure of Pandas and creates GeoDataFrame data structure from it.

- For "file access", it depends upon Fiona
- For plotting, it depends upon: (1) Descartes, and (2) matplotlib

To be able to use GeoPandas, one needs to understand how the data structures of GeoPandas are organized. GeoPandas consists of 2 basic data structures namely: (1) GeoDataFrame, and (2) GeoSeries

GeoSeries

The following facts about the data structure GeoSeries are relevant:

- GeoSeries sub-classes Series data structure of Pandas. So it "inherits" from Series.
- A GeoSeries "contains" a shapely object in form of 2-dimensional points. The number of points would depend upon the "type" of geo spatial object contained in the GeoSeries. For example if the GeoSeries contains a single shapely Point object, then it would have a single instance of x and y coordinates. However if the GeoSeries contains one straight line, then it would have 2 instances of x and y coordinates.

GeoDataFrame

The following facts about GeoDataFrame are relevant:

- GeoDataFrame sub-classes the dataFrame class of Pandas.
- A GeoDataFrame object contains "columns" and "rows" like any other Pandas DataFrame. However it has an additional requirement, that is, it must always have a column which by default is named "geometry".
- Further this "geometry" column of the GeoDataFrame object must always contain a GeoSeries object.

The best way to understand the concepts is to actually "create" a GeoSeries and a GeoDataFrame object from "scratch".

Consider the table below which gives the latitude and longitude of some Indian cities:

City	Latitude	Longitude
Mumbai	19.0728302	72.8826065
Delhi	28.6519508	77.2314911
Bengaluru	12.97194	77.593689
Kolkata	22.5626297	88.3630371
Chennai	13.0878401	80.2784729

The exercise is to plot these 5 cities. The exercise is done in a number of steps. So each step is given first and then the code for that particular step is given. This is shown below (Exercise is on Jupyter notebook):

1. Create a Python dictionary of above 5 cities with 3 keys namely: (1) city_name, (2) latitude, and (3) longitude. (Also do all the imports at the beginning)

```
%matplotlib inline
import pandas as pd
import geopandas
from geopandas import GeoSeries, GeoDataFrame
from shapely.geometry import Point
import matplotlib.pyplot as plt
# city_dict is a Python dictionary
city_dict = {'city_name': ['Mumbai', 'Delhi', 'Benguluru', 'Kolkata', 'Chennai'],
             'latitude': [19.0728302, 28.6519508, 12.97194, 22.5626297,
13.0878401],
             'longitude': [72.8826065, 77.2314911, 77.593689, 88.3630371,
80.2784729]}
```

2. Convert the Python dictionary into a Pandas DataFrame. (The line numbers of output are not numbered because the numbers are kept for the main script)

```
# Create a Pandas DataFrame from a Python dictionary
df = pd.DataFrame(city_dict)
print('df->', df)
```

```
# OUTPUT - - -
df->    city_name   latitude  longitude
0     Mumbai  19.072830  72.882606
1      Delhi  28.651951  77.231491
2  Benguluru  12.971940  77.593689
3    Kolkata  22.562630  88.363037
4    Chennai  13.087840  80.278473
```

3. From the DataFrame, extract the longitude and latitude columns and zip them together to form x-y pairs. These pairs are given to the Point class of shapely module to create Point objects. So a list of 5 Point objects (One for each city) will be created.

```
# Create a list of Point objects. Point is an object of Shapely module
# Longitude is x. Latitude is y
my_points = [Point(xy) for xy in zip(df['longitude'], df['latitude'])]
print('type(my_points)->', type(my_points))
print('my_points->', my_points)
# my_geometry will have 5 Point objects.
print('number of Point objects in the list->', len(my_points))
```

```
# OUTPUT - - -
type(my_points)-> <class 'list'>
my_points-> [<shapely.geometry.point.Point object at 0x0C399610>,
<shapely.geometry.point.Point object at 0x0C3999B0>,
<shapely.geometry.point.Point object at 0x0C399970>,
<shapely.geometry.point.Point object at 0x0C399F90>,
<shapely.geometry.point.Point object at 0x0C399B30>]
number of Point objects in the list-> 5
```

4. Create a GeoSeries object from the list of Point objects by giving the list of Point objects as parameter to the constructor of GeoSeries class.

```
# Create a GeoSeries object. The geometry column is my_points
my_gs = GeoSeries(my_geometry)
print('my_gs->',my_gs)
# my_gs is an object of GeoSeries class
print('type(my_gs)->', type(my_gs))
```

```
# OUTPUT- - -
my_gs-> 0    POINT (72.88260649999999 19.0728302)
1           POINT (77.2314911 28.6519508)
2              POINT (77.593689 12.97194)
3           POINT (88.3630371 22.5626297)
4           POINT (80.2784729 13.0878401)
dtype: object
type(my_gs)-> <class 'geopandas.geoseries.GeoSeries'>
```

5. Create a GeoDataFrame object. The constructor of the GeoDataFrame object will have 2 parameters namely: (1) the DataFrame object df, and (2) the list of Point objects. Finally plot the points using plot() method of GeoPandas. (Note that crs stands for Coordinate Reference System. This is a detailed topic and not discussed here)

```
# You can "embed" the GeoSeries in the GeoDataFrame
my_gdf = GeoDataFrame(df, geometry = my_geometry)
#my_gdf = GeoDataFrame(df, geometry = my_gs)
print('my_gdf->', my_gdf)
print('my_gdf crs->', my_gdf.crs)
print('my_gdf geometry type->', my_gdf.geom_type)
my_gdf.plot()
```

```
# OUTPUT- - -
my_gdf->    city_name   latitude  longitude                          geometry
0     Mumbai  19.072830  72.882606  POINT (72.88260649999999 19.0728302)
1      Delhi  28.651951  77.231491         POINT (77.2314911 28.6519508)
2  Benguluru  12.971940  77.593689            POINT (77.593689 12.97194)
3    Kolkata  22.562630  88.363037         POINT (88.3630371 22.5626297)
4    Chennai  13.087840  80.278473         POINT (80.2784729 13.0878401)
my_gdf crs-> None
my_gdf geometry type-> 0    Point
1    Point
2    Point
3    Point
4    Point
dtype: object
```

```
Out[34]: <matplotlib.axes._subplots.AxesSubplot at 0xc1b0b30>
```

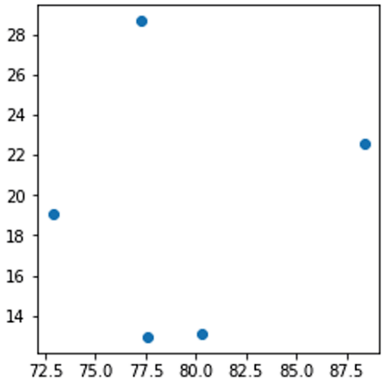


FIGURE Plot of 5 cities on Jupyter notebook

The above exercise gives an insight into the GeoPandas library.

6. *Plotting GeoDataFrame objects on Folium*

It is possible to plot GeoDataFrame objects on a Folium map. To do this however following 2 things are required:

- The GeoDataFrame object must have a crs. As is explained in the discussion in the box below, Folium uses crs EPSG:3857, so the GeoDataFrame object should also have the same crs.
- The GeoDataFrame object needs to be converted into a GeoJSON object

Co-ordinate systems: Earth is spherical or rather somewhat elliptical. The method by which you project the surface of earth onto a map is the "projection" of the surface.

You can think of a projection as follows:

- Suppose you had a globe in the form of a sphere.
- Further suppose you want to project certain cities of the globe on a map.
- One way by which you could do this is to put ink-dots at all such cities and the "wrap" the globe in a graph paper (Which has horizontal and vertical lines drawn on it).
- Now when you "flatten out" the graph paper, then they will have "ink spots" corresponding to the cities.
- So you have a kind of "projection" of the cities on a graph paper.

A standard used worldwide in cartography and also in GPS devices is what is known as the World Geodetic System (WGS). One of the latest versions of this standard is WSG 84 (also known as WGS 1984, EPSG:4326). It was established in 1984. WGS 84 is the reference coordinate system used by the Global Positioning System.

The common coordinate reference systems (crs) used are:

- EPSG:4326 (WGS84). Used mostly for GIS. It uses "simple Equirectangular projection".
- EPSG:3857. Most common for online maps and default for Folium. It uses Spherical Mercator projection
- EPSG:3395. It is rarely used. It uses Elliptical Mercator projection

Folium uses EPSG:3857 by default. You can confirm this by seeing the signature of the Map class of Folium (Also shown in section 22.3.2 of this chapter).

The following script is quite self-explanatory and does the following:

- It first creates a GeoDataFrame object of 5 Indian cities. This step is simply copied from the previous example.
- Since the GeoDataFrame object was created "manually", its crs attribute is None. So you need to set its crs attribute to EPSG:3857, which is the default for Folium.
- Then convert the GeoDataFrame object into a GeoJSON object.
- Then download the geojson file for India from internet and use it to create a Folium map of India.
- Add the 5 city data to the Folium map. This data was converted from GeoDataFrame to GeoJSON in an earlier step.

The script:

```
import requests
import folium
import geojson
%matplotlib inline
import pandas as pd
import geopandas
from geopandas import GeoSeries, GeoDataFrame
from shapely.geometry import Point
import matplotlib.pyplot as plt
# Create Python Dictionary of 5 cities with lat-long
city_dict = {'city_name': ['Mumbai', 'Delhi', 'Benguluru', 'Kolkata',
'Chennai'],
             'latitude': [19.0728302, 28.6519508, 12.97194, 22.5626297,
13.0878401],
             'longitude': [72.8826065, 77.2314911, 77.593689, 88.3630371,
80.2784729]}
# Create Pandas DataFrame from dictionary
df = pd.DataFrame(city_dict)
# Create a list of Point objects. Point is an object of shapely.geometry
# Longitude is x. Latitude is y
my_points = [Point(xy) for xy in zip(df['longitude'], df['latitude'])]
# Create a GeoSeries object. The geometry column is my_points
my_gs = GeoSeries(my_points)
# You can "embed" the GeoSeries in the GeoDataFrame
my_gdf = GeoDataFrame(df, geometry = my_gs)
# The crs of this GeoDataFrame is None
print('my_gdf crs->', my_gdf.crs)
# To plot GeoDataFrame on a Folium map. Ypu need to do 2 thongs as follows
# (1) the crs attribute must be epsg:3827
#
my_gdf.crs = {'init' :'epsg:3827'}
# (2) The GeoDataFrame must be converted to GeoJSON
my_json_cities = my_gdf.to_json()

# The following url gives json of map of india
# Download it using requests and convert it to json format.
my_url =
```

```
'https://raw.githubusercontent.com/johan/world.geo.json/master/countries/IND.geo.
json'
r = requests.get(my_url)
my_india = r.json()
# Select bhopal for centering the map since bhopal is somewhat at center of
India
bhopal = [23.2599, 77.4126]
map_ind = folium.Map(bhopal, zoom_start = 4)
# You can confirm that the default crs for the Folium map is EPSG3857
print('map_india crs->', map_ind.crs)
#folium.GeoJson(r'india.json').add_to(map_ind)
folium.GeoJson(my_india).add_to(map_ind)
# Now add the 5 city points
city_points = folium.features.GeoJson(my_json_cities)
map_ind.add_children(city_points)
map_ind
```

```
# OUTPUT - - -
my_gdf crs-> None
map_india crs-> EPSG3857
```

Further, the screen shot of Folium map created on Jupyter is given in figure below:

Out[12]: