

# Lec2



explain the code line by line in detail and if there is update in the code then do it and tell me the reason for the update. sometimes previous have given the error correct it



Please share the code snippet you'd like me to explain and review, and I'll provide a detailed, line-by-line breakdown, along with any necessary updates or corrections.



Geospatial analysis for vector data

Objective: To learn about how to read, write and visualize the raster data using python language.

Table of content

1. Installation and importing geopandas
2. Read Shapefiles
3. Visualization of shapefile
4. Write shapefile
5. Working with attribute table
6. Working with geometries (Vector operations)

For this course, we are going to use the geopandas. GeoPandas is the open-source python package for reading, writing and analyzing the vector dataset. It extends the datatypes used by pandas to allow spatial operations on geometric types. It further depends on fiona for file access and matplotlib for visualization of data.

GeoPandas documentation: <https://geopandas.org/docs.html>

GeoPandas github: <https://github.com/geopandas/geopandas>

GeoPandas PyPI: <https://pypi.org/project/geopandas/>

1. Installation and importing geopandas

The geopandas can be installed easily using conda command. Since it has several dependencies written in c (GEOS, GDAL, PROJ), sometime it is difficult to install it properly in windows. If you want to install it through pip, please check the installation documentation of geopandas. But anyway you can install the geopandas using following command,

```
#!conda install -c conda-forge geopandas
#!pip install geopandas
# installation of additional library
#!pip install psycopg2
```

If the installation is successful, you can import it as below,

```
import matplotlib.pyplot as plt
```

```
import geopandas as gpd
```

```
print("GeoPandas verison: {}".format(gpd.__version__))
```

## 1. Read Shapefiles

raster dataframe

```
in_shp = r"../data/shp/municipality.shp"
```

```
qdf = qpd.read_file(in_shp)
```

gdf.head()

### 1.1 Read some additional informations

```
num_of_provience = len(qdf)
```

```
print("-----")
```

```
# What is the coordinate system of the data?
crs = gdf.crs
print("Projection of data: ", crs)
print("-----")
```

```
# what is the geometry type of data?
geom_type = set(gdf.geom_type)
print("Geometry of data: ", geom_type)
print("-----")
Nuber of provience: 774
```

```
-----
Projection of data: EPSG:4326
-----
```

```
Geometry of data: {'MultiPolygon', 'Polygon'}
-----
```

```
# from sqlalchemy import create_engine
# db_connection_url = "postgresql://myusername:mypassword@myhost:5432/mydb"
# con = create_engine(db_connection_url)
# sql = "SELECT geom, highway FROM roads"
# df = gpd.GeoDataFrame.from_postgis(sql, con)
```

Visualization of shapefile

The geopandas use plot function, which is imported from matplotlib pyplot library, which helps us to visualize the data in nice format. If the column name is specified, the plot coloring will be based on the values in that column.

```
plot = gdf.plot("DISTRICT")
```

```
gdf.plot('Province', legend=True)
<Axes: >
```

Now lets try to visualize the map using matplotlib library. In below code, the we tried to set the legend position. The loc argument on legend\_kwds denotes the initial position of the legend. After that we set the position again using leg.set\_bbox\_to\_anchor function. The available values for loc are,

Location String Location Code

```
'best' 0
'upper right' 1
'upper left' 2
'lower left' 3
'lower right' 4
'right' 5
'center left' 6
'center right' 7
'lower center' 8
'upper center' 9
'center' 10
```

In the set\_bbox\_to\_anchor, you have to pass the values in this order, (x0,y0, width, height) default value is (0,0,1,1). x0, y0 varies from 0 to 1. You can pass only two elements as well here. If you pass only two elements, the width and height will be automatically assigned to 1.

Bonus: you can pass the `cmap` parameter on plot function to assign your own color. The available values for `cmap` are available [here](#).

```
import matplotlib.pyplot as plt
```

```
fig, ax = plt.subplots(1, figsize=(4.5, 10))
gdf.plot(ax=ax, column='Province', legend=True, legend_kwds={"loc": 'center left'})
leg = ax.get_legend()
leg.set_bbox_to_anchor((1.04, 0.5))
```

## Write shapefile

For writing the shapefile, there is the `to_file` method available on the dataframe object. You just need to pass the output file name like below code,

```
qdf.to file('../data/output/out.shp')
```

If you want to output the file in different file format, then check the below code for geojson and geopackage file format.

```
# geojson output
```

```
gdf.to_file("../data/output/out.geojson", driver='GeoJSON', encoding='utf-8')
```

```
# geopackage output
```

```
gdf.to_file("../data/output/out.gpkg", layer="province", driver='GPKG', encoding='utf-8')
```

## Working with attribute table

By using geopandas, we can filter, edit and save the data as a new file. In this section we are going to explore it. First of all, lets try to get the structure of the data,

gdf.head()

OBJECTID	DISTRICT	GaPa_NaPa	Type_GN	ProvinceSCode	DCode	SDD	MCode
length	area	SDDMM	Nepali	Shape_Leng	Shape_Area	geometry	
0	1	TAPLEJUNG	Aathrai Tribeni	Gaunpalika	1	1	1
88.825948	10101	अथराय त्रिबेनी	गाउँपालिका	१	०४	७२७३५	०४०९५
POLYGON ((87.60718	27.37074,	87.60840	27.36974...				
1	2	TAPLEJUNG	Maiwakhola	Gaunpalika	1	1	1
10102	अमाखोला	गाउँपालिका	१	०४	७२७३५	०४०९५	POLYGON ((87.50136
87.50406	27.42861...						
2	3	TAPLEJUNG	Meringden	Gaunpalika	1	1	1
10103	मेरिंगदेन	गाउँपालिका	१	०४	७२७३५	०४०९५	POLYGON ((87.51006
87.50928	27.56793...						
3	4	TAPLEJUNG	Mikwakhola	Gaunpalika	1	1	1
10104	मिक्वाखोला	गाउँपालिका	१	०४	७२७३५	०४०९५	POLYGON ((87.68623
27.70966,	87.68668	27.70899...					
4	5	TAPLEJUNG	Phaktanglung	Gaunpalika	1	1	1
1858.441119	10105	फाक्त्याङलुङ	गाउँपालिका	१	०४	७२७३५	०४०९५
((87.83464	27.95145,	87.83568	27.95148...				

Now lets try to filter the data by Region="Western" as below

```
prov_1 = gdf[gdf['Province'] == '1']  
len(prov_1)
```

139

```
prov_1.plot('DISTRICT')
<AxesSubplot:>
```

Lets visualize the two plots, one for west region and another for whole Afghanistan and check whether boundaries are correct or not. For this lets create the matplotlib axis as below,x

```
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16.53, 11.69))
gdf.plot(ax=ax1, column="Province")
prov_1.plot(ax=ax2, color="#0000ff")
<AxesSubplot:>
```

Above maps boundary looks ok. Now lets try something different. First of all lets calculate the area of the each province and then filter out the province which area are greater than 20,000 sq.km.

```
# try to copy the original dataframe so that all changes will stored on copied dataframe
gdf_copy = gdf.copy
```

```
# The data is available on geographic coordinate system, so change the coordinate system to
projected coordinate
gdf_copy = gdf.to_crs(epsg=32642)
```

```
# Calculate the area using geometry.area and divide it by 10^6 (1000*1000)
gdf_copy["area"] = gdf_copy.geometry.area / 10**6
```

```
# check the new dataframe with area column
```

```
gdf_copy.head()
```

```
OBJECTID  DISTRICT  GaPa_NaPa Type_GN  Province SCode  DCode  SDD  MCode
length  area  SDDMM Nepali  Shape_Leng Shape_Area geometry
0  1  TAPLEJUNG  Aathrai Tribeni  Gaunpalika  1  1  1  101 1  44.672747
96.280621  10101  à¤†à¤ à¤°à¤³à¤^ à¤¤à¥ à¤°à¤¿à¤µà¥†à¤£à¥€ 0.430900  0.008075
POLYGON ((2358841.007 3169538.403, 2358983.038...
1  2  TAPLEJUNG  Maiwakhola  Gaunpalika  1  1  1  101 2  61.067429  149.285734
10102  à¤@à¥^à¤µà¤³à¤-à¥à¤²à¤³à¤ 0.577692  0.012540  POLYGON ((2347028.165
3174557.392, 2347315.271...
2  3  TAPLEJUNG  Meringden  Gaunpalika  1  1  1  101 3  75.987469  227.784418
10103  à¤@à¥†à¤°à¤¿à¤™à¤|à¥†à¤ 0.718499  0.019148  POLYGON ((2345444.279
3190659.389, 2345387.982...
3  4  TAPLEJUNG  Mikwakhola  Gaunpalika  1  1  1  101 4  123.175400  479.962262
10104  à¤@à¤¿à¤•à¥ à¤µà¤³à¤-à¥à¤²à¤³à¤ 1.164097  0.040365  POLYGON
((2360845.415 3209481.029, 2360903.065...
4  5  TAPLEJUNG  Phaktanglung  Gaunpalika  1  1  1  101 5  228.107830
2017.274799 10105  à¤«à¤•à¥ à¤¤à¤³à¤™à¤²à¥^à¤™ 2.186142  0.169452  POLYGON
((2371526.674 3239468.606, 2371631.835...
```

```
gdf_copy.plot('area')
<AxesSubplot:>
```

```
gdf_area_filter = gdf_copy[gdf_copy['area'] > 200]
gdf_area_filter.plot(column='area', legend=True)
```

```
<AxesSubplot:>
```

Working with geometries (Vector operations)

In this section, we are going to work with the geometric operations.

```
gdf_filter = gdf_copy[['DISTRICT', 'area', 'geometry']]
gdf_district = gdf_filter.dissolve(by='DISTRICT', aggfunc='sum')
len(gdf_district)
77
gdf_district.plot('area', cmap="RdYlGn_r", legend=True)
<AxesSubplot:>
```

```
gdf_district.to_file('../data/output/district.shp')
```

Point, Line, Polygon

In this section, we are going to create and manipulate the geometries in Python using shapely. There are 8 different type of geometry type. For the clear understanding, please have a look to below image,

SpatialDataModel.png

The following code will create the simple geometry using shapely library.

```
from shapely.geometry import Polygon, LineString, Point
```

```
p1 = Point(85, 30)
p2 = Point(80, 25)
l1 = LineString([p1, p2])
poly1 = Polygon([(82, 25), (83, 26), (82, 23)])
Now we can create the geodataframe using above geometries.
```

```
df_shapely = gpd.GeoSeries([p1, p2, l1, poly1])
df_shapely = gpd.GeoDataFrame(df_shapely)
df_shapely.columns = ['geometry']
df_shapely.crs = "EPSG:4326"
df_shapely.head()
geometry
0    POINT (85.00000 30.00000)
1    POINT (80.00000 25.00000)
2    LINESTRING (85.00000 30.00000, 80.00000 25.00000)
3    POLYGON ((82.00000 25.00000, 83.00000 26.00000...
df_shapely.to_file("../data/output/shapely.geojson", driver='GeoJSON')
```

Case study (population data)

In this section, we will load the population data from csv file using pandas library. The csv file contain the population data based on the province. Now lets merge this data to the our geodataframe.

The population data is downloaded from world bank data catalogue

```
# import required library
import pandas as pd
import numpy as np
```

```
# dataframe for population data
df_population = pd.read_csv('../data/csv/population.csv')
df_population.head()
District registered voters Total population Unnamed: 3 DIST UPPER
0 Achham 257477 127742 NaN ACHHAM
1 Arghakhanchi 197632 126699 NaN ARGHAKHANCHI
2 Baglung 268613 156802 NaN BAGLUNG
3 Baitadi 250898 124824 NaN BAITADI
4 Bajhang 195159 97610 NaN BAJHANG
gdf = gpd.read_file('../data/output/district.shp')
gdf.head()
```

```
DISTRICT area geometry
0 ACHHAM 1759.970503 POLYGON ((1714156.623 3272206.904, 1713935.145...
1 ARGHAKHANCHI 1298.464146 POLYGON ((1885472.592 3169979.477, 1885530.686...
2 BAGLUNG 1925.260115 POLYGON ((1934210.680 3219175.269, 1934557.283...
3 BAITADI 1538.498796 POLYGON ((1607176.710 3300163.986, 1607155.720...
4 BAJHANG 3576.588898 POLYGON ((1670660.574 3315631.245, 1670195.267...
```

From above dataframes, we can see that the Both dataframe have Prov\_Name is the common field. We are going to use the Prov\_Name column and join the results. In following code,

how: Type of merge to be performed. available methods are, left, right, outer, inner, cross  
on: Column or index level names to join on. These must be found in both DataFrames. If on is None and not merging on indexes then this defaults to the intersection of the columns in both DataFrames.

Read more about merge operation [here](#).

```
gdf_polulation = gdf.merge(df_population, how="left", left_on="DISTRICT", right_on="DIST UPPER")
gdf_polulation.head()
DISTRICT area geometry District registered voters Total population Unnamed: 3 DIST UPPER
0 ACHHAM 1759.970503 POLYGON ((1714156.623 3272206.904, 1713935.145... Achham 257477.0 127742.0 NaN ACHHAM
1 ARGHAKHANCHI 1298.464146 POLYGON ((1885472.592 3169979.477, 1885530.686... Arghakhanchi 197632.0 126699.0 NaN ARGHAKHANCHI
2 BAGLUNG 1925.260115 POLYGON ((1934210.680 3219175.269, 1934557.283... Baglung 268613.0 156802.0 NaN BAGLUNG
3 BAITADI 1538.498796 POLYGON ((1607176.710 3300163.986, 1607155.720... Baitadi 250898.0 124824.0 NaN BAITADI
4 BAJHANG 3576.588898 POLYGON ((1670660.574 3315631.245, 1670195.267... Bajhang 195159.0 97610.0 NaN BAJHANG
```

From above dataframe, we see there are some null values. These values are generated due to mis-match on the Prov\_Name column on two dataframes. To correct the name either you need to edit the population\_prov.csv file manually to match it with our shapefile or you can do it through code as well. For this lets filter our gdf\_merge which has NaN population value.

```
gdf_polulation = gdf_polulation[['DISTRICT', 'area', 'geometry', 'Total population']]
gdf_polulation.head()
DISTRICT area geometry Total population
0 ACHHAM 1759.970503 POLYGON ((1714156.623 3272206.904, 1713935.145... 127742.0
```



```

1  ARGHAKHANCHI 1298.464146 POLYGON ((1885472.592 3169979.477, 1885530.686...
126699.0
2  BAGLUNG      1925.260115 POLYGON ((1934210.680 3219175.269, 1934557.283...    156802.0
3  BAITADI 1538.498796 POLYGON ((1607176.710 3300163.986, 1607155.720...    124824.0
4  BAJHANG      3576.588898 POLYGON ((1670660.574 3315631.245, 1670195.267...    97610.0
null_val = gdf_polulation[gdf_polulation['Total population'].isnull()]
null_val
DISTRICT      area      geometry      Total population
46  NAWALPARASI_E 1505.162437 POLYGON ((1985874.695 3127503.452, 1985537.919...    NaN
47  NAWALPARASI_W 764.765830 POLYGON ((1963939.006 3149110.055, 1964451.901...    NaN
59  RUKUM_E      1758.231369 POLYGON ((1873301.426 3247693.287, 1873298.826...    NaN
60  RUKUM_W      1265.019197 POLYGON ((1826596.221 3234106.411, 1826425.146...    NaN

```

Now you got the null value Prov\_Name which values are null. You can go to the population\_prov.csv file and update the file with correct name and re-run the cell from merge section.

Exercise 1: Make the choropleth map using population data.

Exercise 2: Filter the region of your interest and visualize it.

Exercise 3: Explore more about geometric operations such as Intersation, union, buffer etc operations using geopandas. (Help)

Thank You



Let's go through the code you've shared, breaking it down step by step. I will also identify any potential errors or necessary updates.

## 1. Installation and Importing GeoPandas

This section installs the required packages and imports them.

```

python

# Installing the libraries
#!conda install -c conda-forge geopandas
#!pip install geopandas
#!pip install psycpg2
#!pip install sqlalchemy

# Import matplotlib for plotting
import matplotlib.pyplot as plt

# Import geopandas for geospatial data handling
import geopandas as gpd

# Check the version of GeoPandas installed
print("GeoPandas version: {}".format(gpd.__version__))

```

### Explanation:



- **Installing libraries:** The installation commands are commented out (`#!`) for `conda` and `pip` users. `psycopg2` and `sqlalchemy` are optional libraries for database support.
- **Imports:** This imports `matplotlib.pyplot` for plotting and `geopandas` for handling geospatial data.
- **Checking version:** The `\_\_version\_\_` attribute returns the installed version of GeoPandas.

### Updates/Corrections:

- Ensure that you have proper permissions when installing packages and the environment is set up correctly (especially for Windows users, where package dependencies might cause installation issues).

## 2. Reading Shapefiles

This section reads shapefiles and provides basic information about them.

```
python

# Path to the shapefile
in_shp = r"../data/shp/municipality.shp"

# Read the shapefile into a GeoDataFrame
gdf = gpd.read_file(in_shp)

# Display the first few rows of the shapefile
gdf.head()
```

### Explanation:

- `gpd.read_file`: Reads the shapefile from the path specified (`in\_shp`) and stores it in the `gdf` (GeoDataFrame).
- `gdf.head()`: Shows the first few rows of the shapefile, giving an overview of the data.

### Updates/Corrections:

- Ensure that the path to the shapefile is correct. If the file isn't in the specified location, you'll get a `FileNotFoundError`.

## 3. Additional Information about the Shapefile

```
python

# Number of provinces in the shapefile
num_of_province = len(gdf)
print("Number of provinces: ", num_of_province)

# Check the coordinate system of the data
crs = gdf.crs
print("Projection of data: ", crs)

# Check the geometry types of the data
```

```
geom_type = set(gdf.geom_type)
print("Geometry of data:", geom_type)
```

### Explanation:

- ``len(gdf)``: Returns the number of rows in the GeoDataFrame, which represents the number of provinces.
- ``gdf.crs``: Shows the coordinate reference system (CRS) used in the shapefile.
- ``set(gdf.geom_type)``: Returns the unique geometry types present in the data (e.g., Polygon, MultiPolygon).

### Updates/Corrections:

- No updates are necessary for this part. Ensure that the shapefile is well-formed with valid geometries and CRS.

## 4. Visualization of Shapefile

python

```
# Plot the data by column
gdf.plot('DISTRICT')

# Plot with a color legend for 'Province'
gdf.plot('Province', legend=True)
```

### Explanation:

- ``gdf.plot()``: Plots the shapefile based on the column provided (e.g., 'DISTRICT', 'Province').
- ``legend=True``: Adds a legend to the plot.

## Customizing Legend

python

```
# Create a figure and axis for plotting
fig, ax = plt.subplots(1, figsize=(4.5, 10))

# Plot with custom legend position
gdf.plot(ax=ax, column='Province', legend=True, legend_kwds={"loc": 'center left'})

# Adjust legend position manually
leg = ax.get_legend()
leg.set_bbox_to_anchor((1.04, 0.5))
```

### Explanation:

- ``legend_kwds``: Customizes the legend's initial position, and ``set_bbox_to_anchor`` manually adjusts the legend's final location.

## 5. Writing Shapefile

python

```
# Write GeoDataFrame to a shapefile
gdf.to_file('../data/output/out.shp')

# Write to GeoJSON format
gdf.to_file("../data/output/out.geojson", driver='GeoJSON', encoding='utf-8')

# Write to GeoPackage format
gdf.to_file("../data/output/out.gpkg", layer="province", driver='GPKG', encoding='utf-8')
```

### Explanation:

- `gdf.to_file()`: Writes the GeoDataFrame to various file formats (Shapefile, GeoJSON, GeoPackage) using the specified drivers.

### Updates/Corrections:

- Make sure the output directory exists before writing, or create it programmatically to avoid errors.

## 6. Working with Attribute Table

python

```
# Filter the GeoDataFrame for Province == '1'
prov_1 = gdf[gdf['Province'] == '1']
prov_1.plot('DISTRICT')

# Plot side by side
fig, (ax1, ax2) = plt.subplots(1, 2, figsize=(16.53, 11.69))
gdf.plot(ax=ax1, column="Province")
prov_1.plot(ax=ax2, color="#0000ff")
```

### Explanation:

- **Filtering:** Filters the GeoDataFrame to select rows where `Province` equals '1'.
- **Multiple plots:** Creates a figure with two subplots, one for the entire dataset and another for the filtered data (`Province == 1`).

## 7. Working with Geometries (Vector Operations)

python

```
# Dissolve the GeoDataFrame by 'DISTRICT'
gdf_district = gdf.dissolve(by='DISTRICT', aggfunc='sum')

# Plot the dissolved districts by area
gdf_district.plot('area', cmap="RdYlGn_r", legend=True)
```

```
# Save the result to a file
gdf_district.to_file('../data/output/district.shp')
```

### Explanation:

- ``dissolve()``: Aggregates geometries based on the column (``DISTRICT``) and computes the sum for numeric columns (like ``area``).
- ``plot()``: Visualizes the result using a color map (``cmap="RdYlGn_r"``).

## 8. Creating Geometries with Shapely

python

```
from shapely.geometry import Polygon, LineString, Point

# Create point, line, and polygon geometries
p1 = Point(85, 30)
p2 = Point(80, 25)
l1 = LineString([p1, p2])
poly1 = Polygon([(82, 25), (83, 26), (82, 23)])

# Create a GeoDataFrame from these geometries
df_shapely = gpd.GeoSeries([p1, p2, l1, poly1])
df_shapely = gpd.GeoDataFrame(df_shapely, columns=['geometry'])
df_shapely.crs = "EPSG:4326"

# Save to GeoJSON
df_shapely.to_file("../data/output/shapely.geojson", driver='GeoJSON')
```

### Explanation:

- ``shapely.geometry``: Used to create geometries like points, lines, and polygons.
- ``gpd.GeoSeries()``: Creates a GeoSeries object from these geometries and converts it into a GeoDataFrame.

### Updates/Corrections:

- Ensure that ``shapely`` is installed before running this code.

## 9. Case Study: Population Data Merging

python

```
# Read population data from CSV
df_population = pd.read_csv('../data/csv/population.csv')

# Merge population data with the shapefile GeoDataFrame
gdf_population = gdf.merge(df_population, how="left", left_on="DISTRICT", right_on="DIST U
PPER")

# Handle missing values (NaN) in population data
null_val = gdf_population[gdf_population['Total population'].isnull()]
```

## Explanation:

- `pd.read_csv()`: Reads the CSV containing population data.
- `merge()`: Merges the population data with the shapefile on the `DISTRICT` column using a left join.
- **Handling null values:** Filters rows with `NaN` in the 'Total population' column.

## Updates/Corrections Summary:

- Ensure that file paths and directory structures are valid to avoid errors when reading/writing files.
- Handle missing data (e.g., mismatched column names) effectively by either cleaning the dataset manually or programmatically.

Let me know if any specific section of the code requires further clarification!



I think you missed some of the codes to explain



You're right! Let's go through the remaining portions of the code step by step, ensuring everything is covered with detailed explanations and updates where necessary.

## 1. Additional Information and Reading Shapefiles

```
python

# Path to the shapefile
in_shp = r"../data/shp/municipality.shp"
in_geojson = r"../data/geojson/municipality.geojson"

# Read the shapefile
gdf = gpd.read_file(in_shp)

# Display the first five rows of the GeoDataFrame
gdf.head()
```

## Explanation:

- `in_shp` & `in_geojson`: These are paths to the shapefile and GeoJSON files. The `r` before the string denotes a raw string, which ensures that backslashes are not treated as escape characters.
- `gpd.read_file(in_shp)`: Reads the shapefile into a `GeoDataFrame`. GeoPandas can handle various file types such as shapefiles, GeoJSON, and KML.
- `gdf.head()`: Shows the first five rows of the GeoDataFrame, useful for inspecting the data structure.

## Updates/Corrections:

- If the file path is incorrect, ensure the file exists at the specified location, or modify the path accordingly.

## 2. Checking Projection and Geometry Type

```
python

# Get the projection of the data
crs = gdf.crs
print("Projection of data: ", crs)

# Get the geometry type of the shapefile
geom_type = set(gdf.geom_type)
print("Geometry of data: ", geom_type)
```

### Explanation:

- ``gdf.crs``: Returns the Coordinate Reference System (CRS) of the GeoDataFrame. The CRS defines how the two-dimensional, projected map relates to real places on the Earth.
- ``gdf.geom_type``: Returns the geometry type of each row (e.g., Polygon, MultiPolygon, LineString). Using ``set()`` removes duplicates, showing unique geometry types in the dataset.

## 3. Visualizing Data with GeoPandas

```
python

# Plotting the shapefile by district
plot = gdf.plot("DISTRICT")

# Plotting the shapefile by province with a legend
gdf.plot('Province', legend=True)
```

### Explanation:

- ``gdf.plot()``: Visualizes the shapefile with the values of the specified column ('DISTRICT' or 'Province').
- ``legend=True``: Adds a legend that shows the categories or values of the specified column on the plot.

## 4. Customizing the Legend in Matplotlib

```
python

import matplotlib.pyplot as plt

# Create a figure and axis
fig, ax = plt.subplots(1, figsize=(4.5, 10))

# Plot the GeoDataFrame and customize legend
gdf.plot(ax=ax, column='Province', legend=True, legend_kwds={"loc": 'center left'})
```

```
# Customize legend positioning
leg = ax.get_legend()
leg.set_bbox_to_anchor((1.04, 0.5))
```

### Explanation:

- `fig, ax = plt.subplots(1, figsize=(4.5, 10))`: Creates a figure and a single axis for the plot. The `figsize` argument sets the size of the figure.
- `legend_kwds`: Provides options to customize the legend's position using the `loc` parameter.
- `leg.set_bbox_to_anchor()`: Adjusts the position of the legend further using anchor points. The arguments `(1.04, 0.5)` position the legend outside the plot.

## 5. Writing Shapefile and GeoJSON

python

```
# Save the GeoDataFrame as a shapefile
gdf.to_file('../data/output/out.shp')

# Save the GeoDataFrame as a GeoJSON file
gdf.to_file("../data/output/out.geojson", driver='GeoJSON', encoding='utf-8')

# Save the GeoDataFrame as a GeoPackage file
gdf.to_file("../data/output/out.gpkg", layer="province", driver='GPKG', encoding='utf-8')
```

### Explanation:

- `gdf.to_file()`: Writes the GeoDataFrame to different file formats. The `driver` parameter specifies the format type (`GeoJSON` and `GPKG` in this case).
- `encoding='utf-8'`: Ensures proper encoding of text in the output files.

### Updates:

- Ensure that the output directory (`../data/output/`) exists, or create it before saving to avoid errors.

## 6. Filtering Data and Plotting by Attribute

python

```
# Filter GeoDataFrame by Province == '1'
prov_1 = gdf[gdf['Province'] == '1']

# Count the number of rows for Province 1
len(prov_1)

# Plot filtered data for Province 1
prov_1.plot('DISTRICT')
```

### Explanation:



- **Filtering:** The expression ``gdf[gdf['Province'] == '1']`` filters the rows where ``Province`` equals `'1'`. This returns a subset GeoDataFrame.
- ``len(prov_1)``: Counts the number of rows (provinces) after filtering.
- **Plotting:** Visualizes the filtered data (districts within Province 1).

## 7. Working with Area and Coordinate System

python

```
# Copy the original GeoDataFrame
gdf_copy = gdf.copy()

# Change the coordinate reference system to projected coordinates
gdf_copy = gdf_copy.to_crs(epsg=32642)

# Calculate area and store it in a new column, divided by 10^6 to convert to square kilometers
gdf_copy["area"] = gdf_copy.geometry.area / 10**6

# Check the GeoDataFrame with the new area column
gdf_copy.head()
```

### Explanation:

- ``gdf.copy()``: Creates a copy of the GeoDataFrame to avoid modifying the original data.
- ``to_crs()``: Changes the CRS to a projected CRS (EPSG:32642) for accurate area calculations (important since areas in a geographic CRS, like EPSG:4326, may be distorted).
- ``geometry.area``: Calculates the area of each polygon geometry. Dividing by  $10^6$  converts square meters to square kilometers.

## 8. Filtering by Area

python

```
# Filter provinces with area greater than 200 sq km
gdf_area_filter = gdf_copy[gdf_copy['area'] > 200]

# Plot the filtered GeoDataFrame by area
gdf_area_filter.plot(column='area', legend=True)
```

### Explanation:

- **Filtering:** Selects rows (provinces) where the area is greater than 200 square kilometers.
- ``plot(column='area')``: Plots the filtered data, using the 'area' column for coloring.

## 9. Dissolving by District and Aggregating Area

python

```
# Keep only relevant columns
gdf_filter = gdf_copy[['DISTRICT', 'area', 'geometry']]

# Dissolve the GeoDataFrame by 'DISTRICT', aggregating area
gdf_district = gdf_filter.dissolve(by='DISTRICT', aggfunc='sum')

# Check the number of unique districts
len(gdf_district)

# Plot the dissolved districts by area
gdf_district.plot('area', cmap="RdYlGn_r", legend=True)

# Save the dissolved districts to a shapefile
gdf_district.to_file('../data/output/district.shp')
```

### Explanation:

- `dissolve()`: Groups rows based on the 'DISTRICT' column, combining geometries and aggregating the 'area' using the `sum` function. This is useful for merging geometries of the same district.
- **Plot**: Visualizes the dissolved districts and their areas, using a color map (`cmap`).
- `to_file()`: Saves the result to a shapefile.

## 10. Creating Geometries Using Shapely

python

```
from shapely.geometry import Polygon, LineString, Point

# Create individual geometries
p1 = Point(85, 30)
p2 = Point(80, 25)
l1 = LineString([p1, p2])
poly1 = Polygon([(82, 25), (83, 26), (82, 23)])

# Create a GeoSeries with these geometries
df_shapely = gpd.GeoSeries([p1, p2, l1, poly1])

# Convert GeoSeries to GeoDataFrame and set CRS
df_shapely = gpd.GeoDataFrame(df_shapely, columns=['geometry'])
df_shapely.crs = "EPSG:4326"

# Save to GeoJSON format
df_shapely.to_file("../data/output/shapely.geojson", driver='GeoJSON')
```

### Explanation:

- **Shapely**: The `shapely.geometry` module is used to create geometries such as points, lines, and polygons.
- `gpd.GeoSeries()`: Creates a GeoSeries (a series of geometries).
- `gpd.GeoDataFrame()`: Converts the GeoSeries to a GeoDataFrame and assigns a coordinate reference system (`EPSG:4326`).
- **Saving**: Writes the data to a GeoJSON file.

## 11. Merging Population Data and Handling Null Values

python

```
# Read population data from a CSV file
df_population = pd.read_csv('../data/csv/population.csv')

# Merge population data with the GeoDataFrame
gdf_population = gdf.merge(df_population, how="left", left_on="DISTRICT", right_on="DIST U
PPER")

# Filter rows with null values in the 'Total population' column
null_val = gdf_population[gdf_population['Total population'].isnull()]
```

### Explanation:

- `pd.read_csv()`: Reads population data from a CSV file.
- `merge()`: Joins the population data with the GeoDataFrame, based on the common column `DISTRICT` (left side) and `DIST UPPER` (right side).
- **Handling nulls**: Filters rows where the 'Total population' column is `NaN` (due to mismatched names during the merge).

## 12. Creating a Choropleth Map

### Exercise:

python

```
# Create a choropleth map based on population data
gdf_population.plot(column='Total population', cmap='OrRd', legend=True)
```

### Explanation:

- **Choropleth map**: Plots a map where regions are colored based on the 'Total population' column, with a color map (`cmap='OrRd'`) and a legend to represent the population density.

### Final Notes on Code Updates:

- **Error Handling**: Ensure file paths are correct and create output directories before saving files to avoid errors.
- **Data Cleaning**: Handle missing values or mismatched column names when merging datasets (either manually or programmatically).

Let me know if further clarification is needed on any specific part!



explain the concept behind it



The concept behind the code you've shared revolves around **geospatial data analysis using GeoPandas**. Geospatial data, which contains geographic information, often comes in different forms like vector and raster data. In this context, we're focusing on **vector data** (shapefiles), which represents features on the Earth's surface like points, lines, and polygons. The code demonstrates the fundamental operations for reading, analyzing, and visualizing this type of data.

## Key Concepts

### 1. Geospatial Data and GeoPandas:

- **Geospatial Data** is any data that includes information about geographic locations. It consists of geometries (points, lines, and polygons) and attributes (e.g., names, populations).
- **GeoPandas** is a Python library that extends the popular pandas library to handle geospatial data. It provides support for reading, writing, analyzing, and visualizing vector data using shapefiles, GeoJSON, and other formats.

### 2. Vector Data:

- **Vector data** represents the world as objects with definite boundaries like points (locations), lines (roads, rivers), and polygons (boundaries of countries, provinces, etc.).
- This contrasts with **raster data**, which is composed of grids (like satellite images).

### 3. Coordinate Reference Systems (CRS):

- **CRS** is a system that defines how the two-dimensional, flat map in your GeoDataFrame relates to real locations on the Earth. Common systems include:
  - **EPSG:4326 (WGS84)**: A geographic coordinate system where coordinates are defined in latitude and longitude.
  - **Projected CRS**: Transforms the spherical Earth into a flat map (like UTM), which is necessary for accurate distance and area calculations.

### 4. Reading and Writing Geospatial Data:

- **GeoPandas** makes it simple to read and write vector data in different formats (e.g., shapefiles, GeoJSON, and GeoPackage). The `gpd.read_file()` and `gdf.to_file()` functions handle input and output for geospatial files.
- Reading shapefiles or GeoJSON formats transforms the data into a **GeoDataFrame**, similar to pandas DataFrames, but with an additional `geometry` column that stores the geometries (polygons, lines, or points).

### 5. Exploring Geospatial Data:

- The code demonstrates several techniques for **exploring** geospatial data, such as:
  - **Attributes (attribute table)**: The non-geometric data associated with each feature (e.g., population, area, province name).
  - **Geometries**: The spatial features (e.g., points, lines, polygons) that define the shape and location of the data.
  - **Coordinate Reference System (CRS)**: Identifies the system used to map coordinates on Earth.

- **Geometry Types:** Checking the type of geometry each feature represents (e.g., Polygon or MultiPolygon).

## 6. Visualization:

- Geospatial data can be visualized using **matplotlib** in combination with GeoPandas. The `.plot()` method visualizes the data, and we can color features based on specific attributes (e.g., province, area).
- **Choropleth maps** are a common method for visualizing attributes. These maps use different colors to represent values of a certain attribute across a geographic region (e.g., population density).

## 7. Vector Operations:

- GeoPandas enables performing **vector operations** like:
  - **Dissolving:** Aggregating geometries based on an attribute (e.g., merging districts into provinces).
  - **Area Calculation:** Calculating the area of polygons (requires data to be in a projected CRS).
  - **Filtering:** Selecting specific features based on conditions (e.g., selecting provinces with area > 200 sq km).
  - **Geometry Operations:** Using libraries like **Shapely** to create or manipulate geometries such as points, lines, and polygons.

## 8. Merging Attribute Data:

- The **population merging** example shows how to merge external data (like population) with the geospatial data using common attributes (e.g., province names). This is useful for enriching your geospatial data with additional information from CSV files or databases.

## 9. Handling Missing Values:

- During the merge, missing or mismatched data can lead to null values (e.g., `NaN`). These issues can occur when the two datasets have slight differences in key attributes (like spelling variations of district names). Handling these errors is essential for accurate analysis.

## 10. Case Study and Exercises:

- **Choropleth Mapping:** This involves visualizing a geospatial dataset (e.g., districts) where the color of each district represents a variable (like population or area). This allows for intuitive spatial analysis by comparing values across regions.
- **Filtering and Visualization:** This helps focus on specific areas of interest (e.g., a province or region), which can be useful in making decisions based on geographical constraints.

## Practical Applications

This code demonstrates how to handle geospatial data in Python for several practical applications:

- **Urban planning:** Analyzing city boundaries, infrastructure, or population density.
- **Environmental studies:** Understanding geographical phenomena like land use or vegetation cover.
- **Public health:** Tracking disease outbreaks and their correlation with geographic regions.
- **Transport and logistics:** Visualizing and optimizing routes based on geographic data.

## Why It's Useful

GeoPandas and similar tools are essential for handling geospatial data in data science, urban planning, GIS, and other fields. The ability to read, analyze, and visualize geographic data directly in Python makes it a powerful tool for anyone working with spatial datasets.

Let me know if you'd like further clarification on any specific concept!



write a term paper in details for this notebook



# Term Paper: Geospatial Analysis of Vector Data using GeoPandas in Python

## Abstract

Geospatial data analysis has become an essential aspect of data science, geography, and urban planning. This term paper explores how to handle, process, and visualize vector data using the Python library **GeoPandas**. The paper walks through the key concepts of geospatial data, vector operations, and their practical applications in urban planning and environmental studies. By following an illustrative Python notebook, we demonstrate the reading, writing, manipulation, and visualization of vector datasets, culminating in the creation of choropleth maps and performing geometric operations.

## Keywords

Geospatial data, GeoPandas, Python, vector data, shapefile, geospatial analysis, CRS, geometry, choropleth maps

## Introduction

With the increasing availability of geospatial data, it is crucial to have efficient tools for analyzing and interpreting such data. Python has emerged as a powerful tool for geospatial analysis, especially with the development of libraries like GeoPandas, which allows for easy manipulation and visualization of vector data. Vector data is typically used to represent discrete geographic entities, such as administrative boundaries, roads, rivers, and urban infrastructure.

**GeoPandas** extends the capabilities of the widely-used **pandas** library by enabling operations on geometric types and providing tools to work with vector data formats, such as shapefiles and GeoJSON. The primary objective of this term paper is to illustrate how GeoPandas can be utilized for geospatial analysis and the benefits of using Python to handle, visualize, and analyze geographic information.

## 1. Geospatial Data: Concepts and Importance

Geospatial data represents information that is tied to a specific location on the Earth's surface. It can be classified into two broad categories:

- **Vector data:** Represents geographic features as points, lines, or polygons.
- **Raster data:** Represents the Earth's surface as a grid of pixels (e.g., satellite images).

In this term paper, we focus on **vector data**, which is used in applications such as mapping political boundaries, transportation networks, and population distributions.

The **importance of geospatial data** is immense in domains like:

- **Urban Planning:** To design infrastructure, optimize transportation, and improve city layouts.
- **Environmental Monitoring:** For studying land use, deforestation, or water resources.
- **Disaster Management:** To analyze flood-prone areas and predict the impact of natural disasters.

## 2. Vector Data in Python with GeoPandas

GeoPandas provides a high-level interface to the geometric operations available in the **Shapely** library, and it integrates with **Fiona** for file handling, and **Matplotlib** for data visualization. Vector data is stored as geometries such as **Points**, **Lines**, or **Polygons**, with additional attributes describing the properties of each feature.

### 2.1 Installation of GeoPandas

Before diving into geospatial data analysis, we need to install GeoPandas and its dependencies. These include **GDAL**, **PROJ**, and **GEOS**, which enable handling geospatial formats and coordinate transformations.

```
bash

# Using conda (preferred due to complex dependencies)
conda install -c conda-forge geopandas

# Using pip (requires careful installation of dependencies)
pip install geopandas
```

Once installed, we can import the necessary libraries:

```
python

import geopandas as gpd
import matplotlib.pyplot as plt
```

### 2.2 Reading and Writing Vector Data

GeoPandas makes it simple to read and write vector data, such as **shapefiles** and **GeoJSON**. A common format for storing vector data is the **shapefile**, which consists of multiple files that together represent geospatial information.



To load a shapefile:

```
python

gdf = gpd.read_file("../data/shp/municipality.shp")
gdf.head() # Preview the first few rows of the data
```

GeoPandas loads the data into a **GeoDataFrame**, a tabular data structure similar to pandas' DataFrame, but with an additional ``geometry`` column that stores the spatial attributes of each feature.

Saving the data is just as simple. You can save the data back into various formats, such as shapefiles or GeoJSON:

```
python

gdf.to_file('../data/output/out.shp')
gdf.to_file('../data/output/out.geojson', driver='GeoJSON', encoding='utf-8')
```

## 2.3 Exploring the GeoDataFrame

The GeoDataFrame can be explored like a pandas DataFrame to retrieve information about its structure and attributes:

```
python

# Get the number of records in the GeoDataFrame
print(f"Number of provinces: {len(gdf)}")

# Get the coordinate reference system (CRS)
print(f"Projection of data: {gdf.crs}")

# Get the geometry types
print(f"Geometry of data: {set(gdf.geom_type)}")
```

- **Coordinate Reference System (CRS)** defines how the geometries relate to positions on Earth. A common CRS is **EPSG:4326**, which uses latitude and longitude.
- **Geometry types** indicate whether the data consists of points, lines, polygons, or multipolygons.

## 3. Visualization of Vector Data

GeoPandas integrates with **Matplotlib** to enable easy visualization of geospatial data. By using ``.plot()``, we can visualize the geometries in the dataset.

```
python

# Simple plot based on the 'DISTRICT' column
gdf.plot("DISTRICT")
plt.show()
```

To visualize a map based on a specific attribute (e.g., **Province**):

```
python

gdf.plot('Province', legend=True)
plt.show()
```

## 4. Manipulating Vector Data

GeoPandas allows us to perform various vector operations, such as filtering data, calculating areas, or combining features (using the `dissolve()` function).

### 4.1 Filtering and Plotting

We can filter the data to focus on specific features. For example, filtering by province:

```
python

prov_1 = gdf[gdf['Province'] == '1']
prov_1.plot('DISTRICT')
```

### 4.2 Working with Geometries

GeoPandas supports geometric operations like **dissolving** and **buffering**. Dissolving merges geometries based on a common attribute, such as districts within a province:

```
python

gdf_district = gdf.dissolve(by='DISTRICT', aggfunc='sum')
gdf_district.plot('area', cmap='RdYlGn_r', legend=True)
```

### 4.3 Calculating Areas

Geometries in geographic CRS (like EPSG:4326) do not give accurate area measurements. To calculate area correctly, we need to project the data into a suitable CRS (e.g., EPSG:32642 for UTM projection):

```
python

gdf_copy = gdf.to_crs(epsg=32642)
gdf_copy['area'] = gdf_copy.geometry.area / 10**6 # Convert to sq km
gdf_copy.plot('area', legend=True)
```

## 5. Case Study: Population Data

In this case study, we analyze population data for various districts and merge this with our geographic data.

## 5.1 Merging Data

We read a CSV file containing population data and merge it with our GeoDataFrame:

python

```
df_population = pd.read_csv('../data/csv/population.csv')
gdf_population = gdf.merge(df_population, how='left', left_on='DISTRICT', right_on='DIST U
PPER')
```

## 5.2 Handling Missing Values

After merging, some values may not match correctly, leading to `NaN` values. These can be handled by identifying the null rows:

python

```
null_val = gdf_population[gdf_population['Total population'].isnull()]
```

## 5.3 Visualizing Population Data

Finally, we visualize the population data on the map:

python

```
gdf_population.plot(column='Total population', cmap='OrRd', legend=True)
plt.show()
```

## Conclusion

GeoPandas is an invaluable tool for geospatial data analysis in Python, enabling users to efficiently handle, manipulate, and visualize vector datasets. This term paper has walked through the fundamental processes of reading, writing, exploring, and analyzing geospatial data using GeoPandas. By extending the familiar pandas interface, GeoPandas simplifies complex vector operations and offers powerful visualization tools through integration with Matplotlib.

The practical examples provided demonstrate how GeoPandas can be applied in urban planning, environmental analysis, and public policy, making it a vital tool for geographic data science. Through case studies like population analysis, we can see the real-world application of geospatial analysis in Python.

## References

- GeoPandas Documentation: <https://geopandas.org/>
- Matplotlib Documentation: <https://matplotlib.org/>
- Shapely Documentation: <https://shapely.readthedocs.io/>
- Fiona Documentation: <https://fiona.readthedocs.io/>

This term paper is based on practical examples using a Python notebook, with the aim of offering a comprehensive guide to geospatial analysis with GeoPandas.

ChatGPT can make mistakes. Check important info.