Spatial II

Peter Ganong and Maggie Shi

October 28, 2024

Table of contents I

Introduction to data structures in geopandas (6.2)

Geopandas roadmap

In practice, we won't be coding our geodata by hand... Instead we are going to use shapefiles!

```python
import geopandas as gpd
```

Roadmap

- Vocabulary
- File formats
- Read in data
- Preview data

Define vocabulary

Vocabulary

- A `GeoDataFrame` is basically like a `pandas.DataFrame` that contains dedicated columns for storing geometries.
  - We will start with examples with a single column and later teach you how to use more than one column
- That column is called a `GeoSeries`. This can be any of data types (point, line, polygon) from the prior section. All of the methods you saw in the last section can also be used on a `GeoSeries`

File format I: Shapefile

- consists of at least three files `.shp` has feature geometrics, `.shx` has a positional index, `.dbf` has attribute information
- Usually also have `.prj` which describes the Coordinate Reference System (CRS)
- When you read in `map.shp` it automatically reads the rest of them as well to give you proper GeoDataFrame composed of geometry, attributes and projection.

Coordinate Reference Systems

- Coordinate Reference System (CRS) is a combination of:
    - "Datum": origin of latitude and longitude
    - "Project": representation of curved surface onto flat map
- Most common CRS: WGS84 (used for GPS)
- All coordinates are consistent *within* a CRS, but not always *across* CRS's
- Different CRS's suit different needs
    - optimized for local vs. global accuracy
    - different approaches to approx. shape of the earth
    - distance is measured in different units: degrees, miles, meters
- Each system is associated with a unique *EPSG code*. Searchable on https://epsg.io
    - (Aside: EPSG stands for European Petroleum Survey Group)
    - These codes are used to convert one CRS into another

Reading a Shapefile `.shp`

```python
#in same dir:  `.shx` and `.dbf`
filepath = "data/shp/austin_pop_2019.shp"
data = gpd.read_file(filepath)
```

File format II: GeoPackage

- single file `.gpkg`
- Supports both raster and vector data
- Efficiently decodable by software, particularly in mobile devices

GeoPackage is more modern, but you will encounter shapefiles everywhere you look so good to be familiar with it.

Reading a GeoPackage gpkg

```
filepath = "data/austin_pop_2019.gpkg"
data = gpd.read_file(filepath)
type(data)
```

```
geopandas.geodataframe.GeoDataFrame
```

Previewing a GeoDataFrame

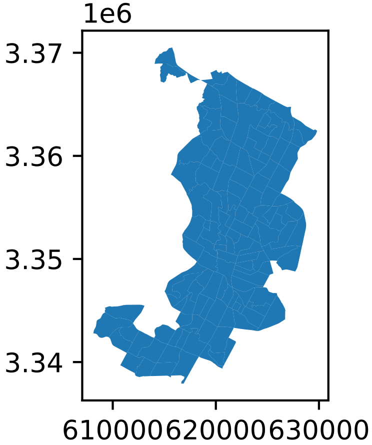
```
data.head()
```

	pop2019	tract	geometry
0	6070.0	002422	POLYGON ((615643.487 3338728.496, 615645.4
1	2203.0	001751	POLYGON ((618576.586 3359381.053, 618614.3
2	7419.0	002411	POLYGON ((619200.163 3341784.654, 619270.8
3	4229.0	000401	POLYGON ((621623.757 3350508.165, 621656.2
4	4589.0	002313	POLYGON ((621630.247 3345130.744, 621717.9

Previewing a GeoSeries

```
data.plot()
```

```
<Axes: >
```



Discussion question: Why isn't it enough to just to `head()`?

Geopandas summary

- `GeoDataFrame` and `GeoSeries` are the counterparts of `pandas.DataFrame` and `pandas.Series`
- `.shp` and `.gpkg` are two ways of storing geo data
- Always plot your map before you do anything else

Geometries in geopandas (6.2)

geometries: roadmap

- methods applied to `GeoSeries`
- my first choropleth

GeoSeries

```python
type(data["geometry"])
```

```
geopandas.geoseries.GeoSeries
```

head()

```python
data["geometry"].head()
```

```
0    POLYGON ((615643.487 3338728.496, 615645.477 3...
1    POLYGON ((618576.586 3359381.053, 618614.330 3...
2    POLYGON ((619200.163 3341784.654, 619270.849 3...
3    POLYGON ((621623.757 3350508.165, 621656.294 3...
4    POLYGON ((621630.247 3345130.744, 621717.926 3...
Name: geometry, dtype: geometry
```

calculate area (in km^2)

```
data["geometry"].area
```

```
0      4.029772e+06
1      1.532030e+06
2      3.960344e+06
3      2.181762e+06
4      2.431208e+06
           ...
125    2.321182e+06
126    4.388407e+06
127    1.702764e+06
128    3.540893e+06
129    2.054702e+06
Length: 130, dtype: float64
```

add column to data frame

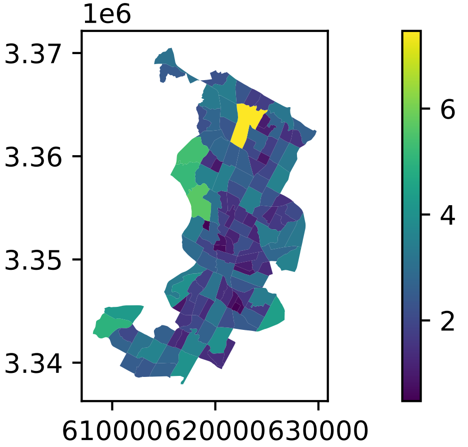
```python
#data.area is just a shorthand for data.geometry.area
data["area_km2"] = data.area / 1000000
data.head()
```

	pop2019	tract	geometry
0	6070.0	002422	POLYGON ((615643.487 3338728.496, 615645.4
1	2203.0	001751	POLYGON ((618576.586 3359381.053, 618614.3
2	7419.0	002411	POLYGON ((619200.163 3341784.654, 619270.8
3	4229.0	000401	POLYGON ((621623.757 3350508.165, 621656.2
4	4589.0	002313	POLYGON ((621630.247 3345130.744, 621717.9

my first choropleth

```
data.plot(column="area_km2", legend=True)
```

```
<Axes: >
```



Why is this a nearly useless set of colors?

geometries: summary

- can do all the same operations on a `GeoSeries` that you would do on any other polygon, like `Area`
- `data.plot(column="var")` draws a choropleth map with shading corresponding to the highlighted variable

Common geometric operations (6.3)

common geometric operations: roadmap

- load and explore data
- methods
  - centroid
  - bounding box
  - buffer
  - dissolve
  - spatial join
- do-pair-share

Austin, continued

(The textbook uses a slightly different file here, unclear why.)

```
filepath = "data/austin_pop_density_2019.gpkg"
data = gpd.read_file(filepath)
```

explore the data I

```
data.head()
```

	pop2019	tract	area_km2	pop_density_km2	geometry
0	6070.0	002422	4.029772	1506.288778	MULTIPOLYG
1	2203.0	001751	1.532030	1437.961394	MULTIPOLYG
2	7419.0	002411	3.960344	1873.322161	MULTIPOLYG
3	4229.0	000401	2.181762	1938.341859	MULTIPOLYG
4	4589.0	002313	2.431208	1887.538658	MULTIPOLYG

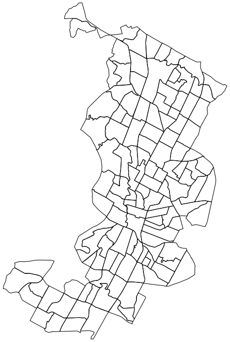
explore the data II

```
type(data["geometry"].values[0])
```

```
shapely.geometry.multipolygon.MultiPolygon
```

explore the data III

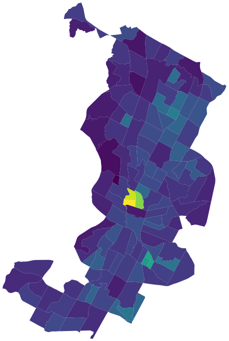
```
import matplotlib.pyplot as plt
data.plot(facecolor="none", linewidth=0.2)
plt.axis("off")
plt.show()
```



- Import `matplotlib.pyplot` to access additional plotting options (e.g., x and y labels, title)

explore the data IV

```
data.plot(column="pop_density_km2")
plt.axis("off")
plt.show()
```



- `facecolor` (or `fc` or `color`) defines a uniform color across all geometries
- whereas `columns` generates colors based on the underlying

methods: centroid I

What it is: arithmetic mean position of all the points in a polygon

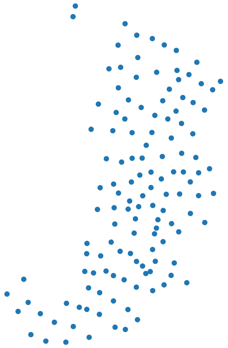
Sample use case: measuring distance between center of each multipolygon

```
data["geometry"].centroid.head()
```

```
0    POINT (616990.190 3339736.002)
1    POINT (619378.303 3359650.002)
2    POINT (620418.753 3342194.171)
3    POINT (622613.506 3351414.386)
4    POINT (622605.359 3343869.554)
dtype: geometry
```

methods: centroid II

```python
data.centroid.plot(markersize=1)
plt.axis("off")
plt.show()
```



methods: bounding box definition

What it is: the tightest possible rectangle around a shape, capturing all of its points within this rectangle.

Sample use case: filtering a larger spatial dataset to subset of interest

methods: bounding box for each polygon I

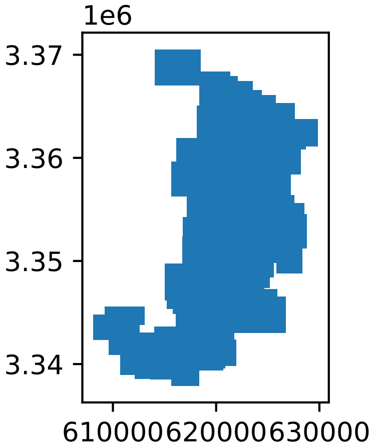
```
data.envelope.head()
```

```
0    POLYGON ((615643.488 3337909.895, 618358.033 3...
1    POLYGON ((618529.497 3358797.000, 620192.632 3...
2    POLYGON ((619198.456 3340875.421, 621733.880 3...
3    POLYGON ((621599.087 3350329.320, 623714.365 3...
4    POLYGON ((621630.247 3343015.679, 624133.189 3...
dtype: geometry
```

methods: bounding box for each polygon II

```
data.envelope.plot()
```

```
<Axes: >
```



methods: bounding box for whole data I

```
data.total_bounds
```

```
array([ 608125.39429998, 3337909.89499998,  629828.38850021
        3370513.68260002])
```

methods: bounding box for whole data II

Flashback to section 6.1

```
from shapely import Point, Polygon
point1 = Point(data.total_bounds[0], data.total_bounds[1])
point2 = Point(data.total_bounds[2], data.total_bounds[1])
point3 = Point(data.total_bounds[2], data.total_bounds[3])
point4 = Point(data.total_bounds[0], data.total_bounds[3])
poly = Polygon([point1, point2, point3, point4])
#poly
```

- *Note*: the order in which you put these points together matters, and you'll get all sorts of interesting shapes with different orders!

methods: buffer I

What it is: shape representing all points that are less than a certain distance from the original shape

Sample use case: how many stores or parks near a neighborhood

methods: buffer II

```
data.buffer(1000).plot(edgecolor="white") #1000 meters
plt.axis("off")
plt.show()
```

methods: dissolve I

What it is: combining geometries into coarser spatial units based on some attributes.

Sample use case: construct the geometries that you want to serve with public transit

```
# Create a new column and add a constant value
data["dense"] = 0

# Filter rows with above average pop density and update the
data.loc[data["pop_density_km2"] > data["pop_density_km2"]
data.dense.value_counts()
```

```
dense
0    86
1    44
Name: count, dtype: int64
```

methods: dissolve II

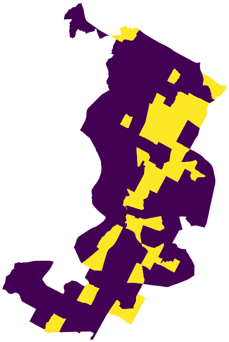
```
dissolved = data[["pop2019", "area_km2", "dense", "geometry
    by="dense", aggfunc="sum"
)
#aggregation step set index to "dense", reset to default
dissolved = dissolved.reset_index()
dissolved
```

	dense	geometry	p
0	0	MULTIPOLYGON (((614108.230 3339640.551, 614288...	3
1	1	MULTIPOLYGON (((612263.531 3338931.800, 612265...	2

- Aggregating alters the way the data is indexed and makes the grouping variable the index
- We need to reset it in order to plot, since some plotting libraries expect data to be indexed in a specific way

methods: dissolve III

```
dissolved.plot(column="dense")
plt.axis("off")
plt.show()
```



Discussion Question: is this a good map? What can we do to improve it?

methods: spatial join

Spatial join: find the closest neighbor.

(Contrived) example: Join Austin tracts to themselves. How many tracts should we expect to get?

```
data_for_join = data[["tract", "geometry"]]
print("N tracts " + str(len(data_for_join)))
join_to_self = gpd.sjoin_nearest(data_for_join, data_for_jo
```

```
N tracts 130
```

methods: spatial join II

```
print("N tracts w closest neighbor " + str(len(join_to_sel
join_to_self.head(4)
```

```
N tracts w closest neighbor 848
```

	tract_left	geometry
0	002422	MULTIPOLYGON (((615643.488 3338728.496, 615645...
0	002422	MULTIPOLYGON (((615643.488 3338728.496, 615645...
0	002422	MULTIPOLYGON (((615643.488 3338728.496, 615645...
0	002422	MULTIPOLYGON (((615643.488 3338728.496, 615645...

common geometric operations: summary

- methods
  - centroid computes arithmetic mean of points in the polygon
  - bounding box expands polygon in a rectangle
  - buffer expands polygon in every direction
  - dissolve combines several polygons
  - spatial join finds nearest neighbor
- do-pair-share

do pair share

Goal: Create and plot a 500m buffer zone around the dense areas in Austin.

Steps

1. From the `dissolved` `GeoDataFrame`, get the polygon for the dense areas
2. Create a new geometry object called `geo`, which is the dense areas with a 500m buffer
3. `geo.plot()`

After you are done, here are some cosmetic suggestions:

- Start with a grey plot of all of the Austin boundaries: `austin = data.plot(color="grey")`
- Make your buffer transparent
- Putting it all together `geo.plot(ax = austin, alpha=0.5)`
    - This plots the `geo` object with 50% transparency, on top of axes based on the `austin` object