# Spatial II

Peter Ganong and Maggie Shi

October 30, 2024

# Table of contents I


Introduction to data structures in geopandas (6.2)

Geometries in geopandas (6.2)

Common geometric operations (6.3)

Spatial Join and Nearest Neighbor Analysis (6.7, 6.8)

# 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!

```
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

# Reading a Shapefile .shp

```
#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

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

```
geopandas.geodataframe.GeoDataFrame
```

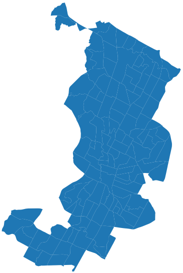
# Previewing a GeoDataFrame

```
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().set_axis_off()
```



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()

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

```
0    POLYGON ((615643.487 3338728.496, 615645.477 3...
1    POLYGON ((618576.586 3359381.053, 618614.33 33...
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

```python
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
```

# calculate area

```
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
```

**In-class exercise**: what unit is this? Open the `austin_pop_2019.prj` file in Quarto, then copy into ChatGPT and query it.

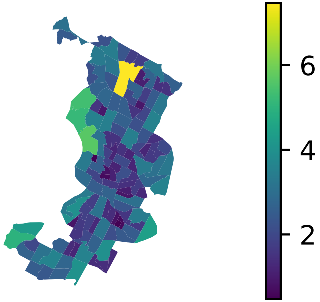
# add column to data frame

```
data["area_km2"] = data.area / 1000000
data[['tract', 'area_km2', 'geometry']].head()
```

| | tract | area_km2 | geometry |
|---|---|---|---|
| 0 | 002422 | 4.029772 | POLYGON ((615643.487 3338728.496, 615645.477 3 |
| 1 | 001751 | 1.532030 | POLYGON ((618576.586 3359381.053, 618614.33 33 |
| 2 | 002411 | 3.960344 | POLYGON ((619200.163 3341784.654, 619270.849 3 |
| 3 | 000401 | 2.181762 | POLYGON ((621623.757 3350508.165, 621656.294 3 |
| 4 | 002313 | 2.431208 | POLYGON ((621630.247 3345130.744, 621717.926 3 |

# my first choropleth

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




**Discussion question** – are the colors in this graph useful?

# 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
- do-pair-share

# Austin, continued

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

```
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( |

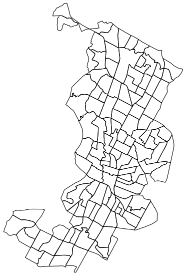
# explore the data II

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

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

# explore the data III

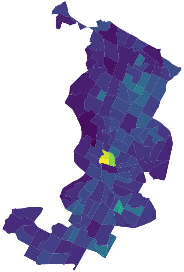
```
import matplotlib.pyplot as plt
data.plot(facecolor="none", linewidth=0.2).set_axis_off()
```



- Import `matplotlib.pyplot` to access additional plotting options
- `facecolor` (or `fc` or `color`) defines a uniform color across all geometries

# explore the data IV

```
data.plot(column="pop_density_km2").set_axis_off()
```



- in contrast to `facecolor`, `columns` generates colors based on the underlying values

# methods: centroid I

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

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

```
0     POINT (616990.19 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
```

Note the data type: we're making a new geometry object

# methods: centroid II

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

# example use cases: centroid

- measuring distance between center of each multipolygon
- simplifying data representation for dense or complex polygons
- label placement in mapping

# aside: change active geometry

- a `GeoDataFrame` can store multiple `Geoseries`
- and we can switch which one is the "main" one – the one that geopandas will by default do spatial operations on

```
data["centroid"] = data.centroid
data.set_geometry("centroid")
data[['tract', 'centroid', 'geometry']].head()
```

| | tract | centroid | geometry |
|---|---|---|---|
| 0 | 002422 | POINT (616990.19 3339736.002) | MULTIPOLYGON (((615643 |
| 1 | 001751 | POINT (619378.303 3359650.002) | MULTIPOLYGON (((618576 |
| 2 | 002411 | POINT (620418.753 3342194.171) | MULTIPOLYGON (((619200 |
| 3 | 000401 | POINT (622613.506 3351414.386) | MULTIPOLYGON (((621623 |
| 4 | 002313 | POINT (622605.359 3343869.554) | MULTIPOLYGON (((621630 |

# methods: bounding box definition

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

# methods: bounding box for each polygon I

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

```
0    POLYGON ((615643.488 3337909.895, 618358.033 3...
1    POLYGON ((618529.497 3358797, 620192.632 33587...
2    POLYGON ((619198.456 3340875.421, 621733.88 33...
3    POLYGON ((621599.087 3350329.32, 623714.365 33...
4    POLYGON ((621630.247 3343015.679, 624133.189 3...
dtype: geometry
```

# methods: bounding box for each polygon II

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

# example use cases: bounding box

- use this when you don't have a better way to filter the data
- when would you not have a better way?
  - when pulling data to make a map. maps are rectangular because screens are rectangular
  - when you want to do something fast computationally. because it just compares points in XY space to X_min, X_max, Y_min, Y_max, it is much faster than using a spatial join (discussed below)

# methods: bounding box for whole data

We can also retrieve the corner coordinates of the bounding box for a `GeoDataFrame`

```
data.total_bounds
```

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

# use cases: bounding box for whole data

- identifying total coverage area of a map
- making cropped or zoomed-in maps – start with the overall bounding box, then “zoom” in by reducing coordinates
- make loading large spatial datasets easier – load only a subset of the data based on the bounding box

# methods: buffer I

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

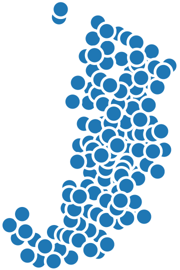
## methods: buffer II

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

Since our CRS is in meters, the buffer is defined to be 1000 meters around the border

# methods: buffer III

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



# example use cases: buffer

- how many stores or parks near a neighborhood
- identify safety/hazard zones around buildings
- add bike lines or parking spots along roads
- working with geometries that have complicated borders (e.g. coasts)
- selecting nearby geometries – example below with spatial join

# methods: dissolve I

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

# example use cases: dissolve

- aggregating from smaller spatial unit to larger: counties to states, census tracts to school districts
- reducing complexity of large, dense spatial datasets
- **Example with Austin data**: we will construct the geometries that you might want to serve with public transit by identifying *dense vs. non-dense* tracts

# dissolve example I

```python
data["dense"] = 0

data.loc[data["pop_density_km2"] >
↪   data["pop_density_km2"].quantile(0.75), "dense"]
↪   = 1
data.dense.value_counts()
```

```
dense
0    97
1    33
Name: count, dtype: int64
```

# dissolve example II

```
dissolved = data[["pop2019", "area_km2", "dense",
↪  "geometry"]].dissolve(
    by="dense", aggfunc="sum"
)
dissolved = dissolved.reset_index()
dissolved
```

| | dense | geometry | p |
|---|---|---|---|
| 0 | 0 | MULTIPOLYGON (((618185.858 3340270.827, 618127... | 4 |
| 1 | 1 | MULTIPOLYGON (((619541.045 3341062.71, 619547.... | 1 |

- 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").set_axis_off()
plt.axis("off")
plt.show()
```

Discussion Question: What can we do to improve this map?

# 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
- 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 a single polygon for the dense areas
   - Hint: you can use standard `pandas` commands to subset

2. Create a new geometry object called `geo`, which is the dense areas with a 500m buffer around them

3. Plot your new geometry object `geo.plot()`

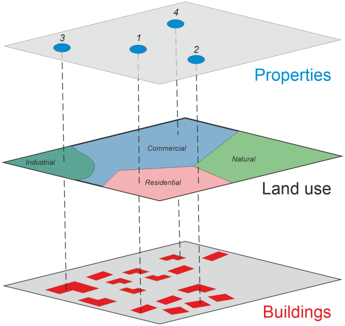
# Spatial Join and Nearest Neighbor Analysis (6.7, 6.8)

# spatial Join and Nearest neighbor analysis: roadmap

- spatial join: example with finding intersection between two layers
- nearest neighbor analysis: identify neighbors of each census tract

# Spatial Join

**What it is**: retrieving attributes from one layer and transferring them into another layer based on their spatial relationship




**Layer attributes**

| Id | Address | N. of rooms | Floor area (m2) | Price |
|---|---|---|---|---|
| 1 | Street 1 | 3 | 70 | €600k |
| 2 | Street 2 | 2 | 42 | €450k |
| 3 | Street 3 | 4 | 300 | €350k |
| 4 | Street 4 | 1 | 600 | €200k |

+

| Landuse | Land cover | Area (km2) |
|---|---|---|
| Residential | Urban fabric | 3.2 |
| Residential | Urban fabric | 3.2 |
| Industrial | Urban fabric | 2.1 |
| Commercial | Urban fabric | 5.3 |

+

| Construction year | N. of floors | Elevator |
|---|---|---|
| 1932 | 5 | Yes |
| 1960 | 4 | No |
| 1999 | 1 | No |
| 2007 | 2 | Yes |

# Spatial Join: use cases

Example use cases:

- adding attributes from one geographic unit to another, smaller geographic unit: assigning county-level statistics to zip codes
- using one layer to crop another
- identifying where two layers *do not* overlap

# spatial join I

**Example with Austin**: we want to identify all census tracts that are adjacent to the dense tracts

Step 1: get polygon of dense zones

```
dense_zones = dissolved[dissolved['dense'] == 1]
dense_zones.plot().set_axis_off()
```

# spatial join II

Step 2: create 10-meter buffer around dense areas

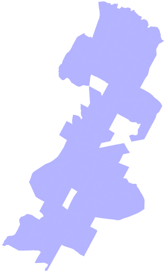
```
dense_buffer = dense_zones.copy()
dense_buffer['geometry'] =
↪  dense_buffer.geometry.buffer(10)

dense_buffer.plot(color="red", alpha=0.5).set_axis_off()
```

# spatial join III

Step 3: find intersection between data and dense_buffer

```
near_dense = gpd.sjoin(data, dense_buffer, how="inner",
↪ predicate="intersects")
near_dense.plot(color="blue", alpha=0.3).set_axis_off()
```

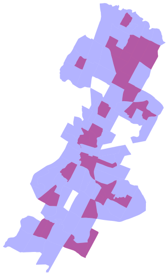


# spatial join IV

Step 4: plot `dense_zones` and `near_dense`

```
fig, ax = plt.subplots()
dense_zones.plot(ax=ax,color="red", alpha=0.5,
↪  label="Dense").set_axis_off()
near_dense.plot(ax=ax,color="blue", alpha=0.3, label="Near
↪  Dense").set_axis_off()

plt.show()
```



# spatial join V

Spatial join options: * `how`: `inner` (default), `left`, `right` * `predicate`: `intersects` (default), `contains`, `covered_by`, `covers`, `crosses`, `overlaps`, `touches`, `within` * Discussion question: what is another type of join we could have used to identify tracts next to dense areas?

# methods: nearest neighbor analysis I

**What it is**: for every geography unit, find the other unit(s) that are closest in distance.

Example use cases:

- find closest voting center to an address
- public transportation planning: how far is the closest metro stop?
- real estate: how much did nearby houses sell for?

# methods: nearest neighbor analysis II

```
data_for_join = data[["tract", "geometry"]]
print("N tracts " + str(len(data_for_join)))
```

```
N tracts 130
```

**Example with Austin**: Join every Austin tract to its closest neighbor or neighbors.

# methods: nearest neighbor analysis III

```
join_to_self = gpd.sjoin_nearest(
    data_for_join,  #left df
    data_for_join, #right df
    how='inner',
    distance_col="distance"
)
```

- you will always specify a left dataframe and a right dataframe
- if both dataframes have the same variable, `sjoin_nearest` will add suffixes indicating if it comes from the left or the right dataframe
- in our example, both left and right dataframes have the exact same columns

# methods: nearest neighbor analysis IV

```
print("N tracts w closest neighbor " +
    str(len(join_to_self)))
join_to_self[['tract_left', 'tract_right',
 ↪  'distance']].head(4)
```

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

| | tract_left | tract_right | distance |
|---|---|---|---|
| 0 | 002422 | 002423 | 0.0 |
| 0 | 002422 | 002422 | 0.0 |
| 0 | 002422 | 002424 | 0.0 |
| 0 | 002422 | 002402 | 0.0 |

Note that `002422` is considered its own neighbor!

# methods: nearest neighbor analysis V

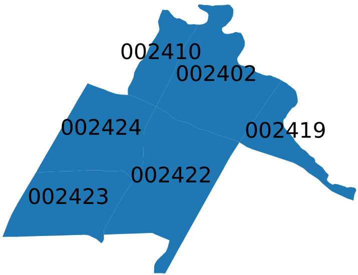
```
neighbor_ids = join_to_self[join_to_self['tract_left'] ==
↪  '002422']['tract_right']
print(neighbor_ids)
```

```
0    002423
0    002422
0    002424
0    002402
0    002410
0    002419
Name: tract_right, dtype: object
```

# methods: nearest neighbor analysis VI

```
tracts_of_interest =
↪  data[(data['tract'].isin(neighbor_ids))]
tracts_of_interest.plot().set_axis_off()

for idx, row in tracts_of_interest.iterrows():
    centroid = row.geometry.centroid
    plt.annotate(text=row['tract'], xy=(centroid.x,
↪  centroid.y),
                 ha='center', fontsize=8)
plt.show()
```



# summary: spatial join and nearest neighbor analysis

- Spatial joins and nearest neighbor analysis allow us to "join" different `Geodataframes` based on space and proximity

- Like regular joins, the order in which you specify left vs. right dataframe matters