Advanced Spatial Methods II

Peter Ganong and Maggie Shi

February 11, 2026

Skills acquired by the end of this lecture

- Understand how to incorporate spatial methods to make production-quality maps
- Understand how to use spatial methods to create “tools” for spatial calculations

Use Case 3: production-quality maps

Roadmap: production-quality maps

- Spatial methods use case 3: **production-quality maps**
- `matplotlib` to layer maps
- `.boundary`
- `.centroid`
- Iterating through observations: using `.iterrows()`
- Layering different geodata together: `zorder`

Use Case 3: production-quality maps

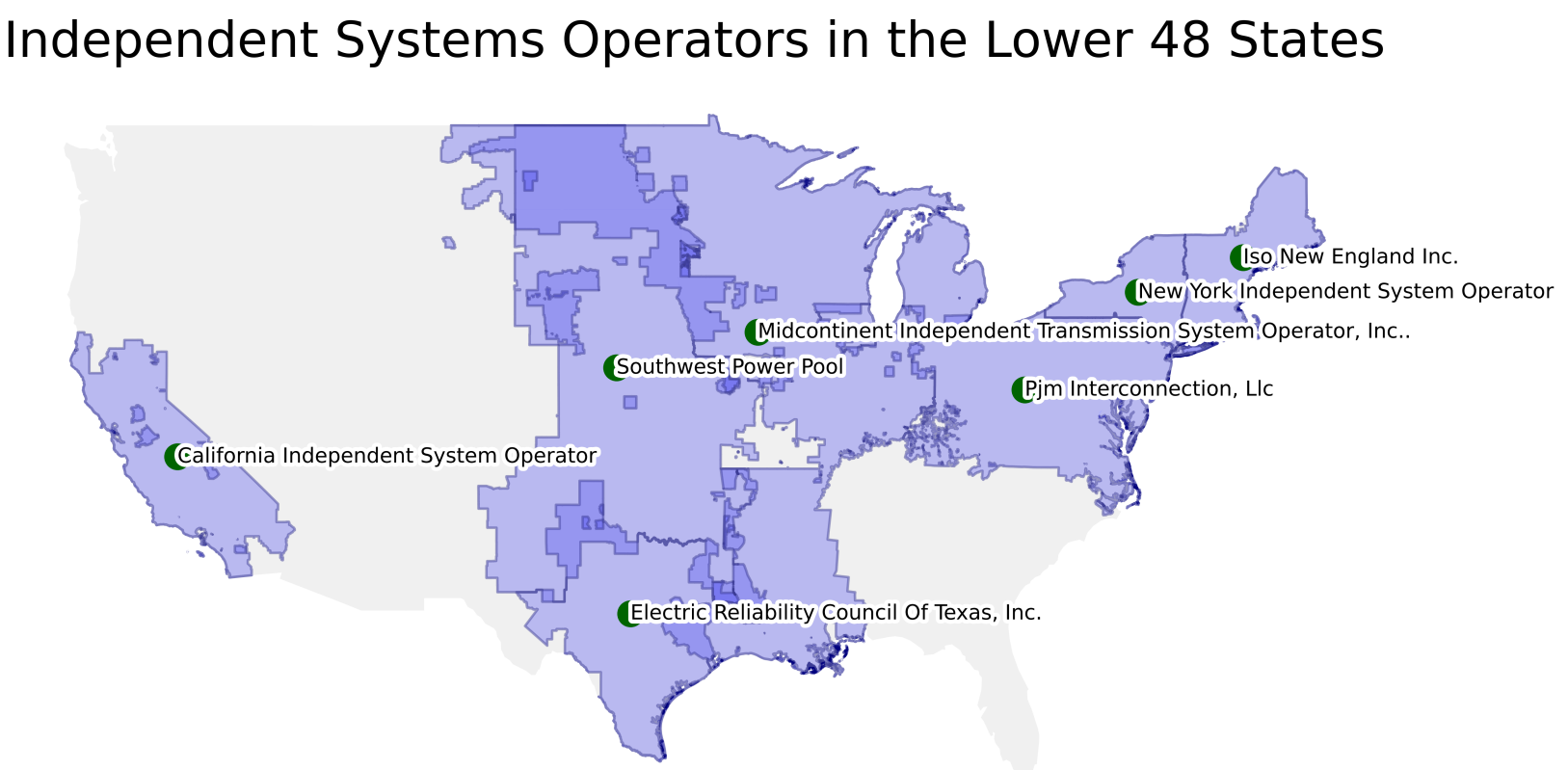
Basic map of all ISOs:

Use Case 3: production-quality maps

- Recall the “exploration” vs. “production” distinction from data visualization lectures
- This plot falls squarely under “exploration” – would not be ready for a stakeholder-facing presentation, published report, etc.

Use Case 3: production-quality maps

We can use spatial methods to get something much nicer-looking. Here's the end result we're aiming for:



Improving our map: boundaries

- `boundary` method returns `GeoSeries` with the boundary of every geometry in `iso_gdf`
- Converts `MultiPolygon` into `MultiLineString`
- Plotting boundaries allows for better layering

```
1  iso_boundaries = iso_gdf.boundary
2  print("Boundary geometry type:", iso_boundaries.geom_type.value_counts())
```

```
Boundary geometry type: MultiLineString    7
Name: count, dtype: int64
```

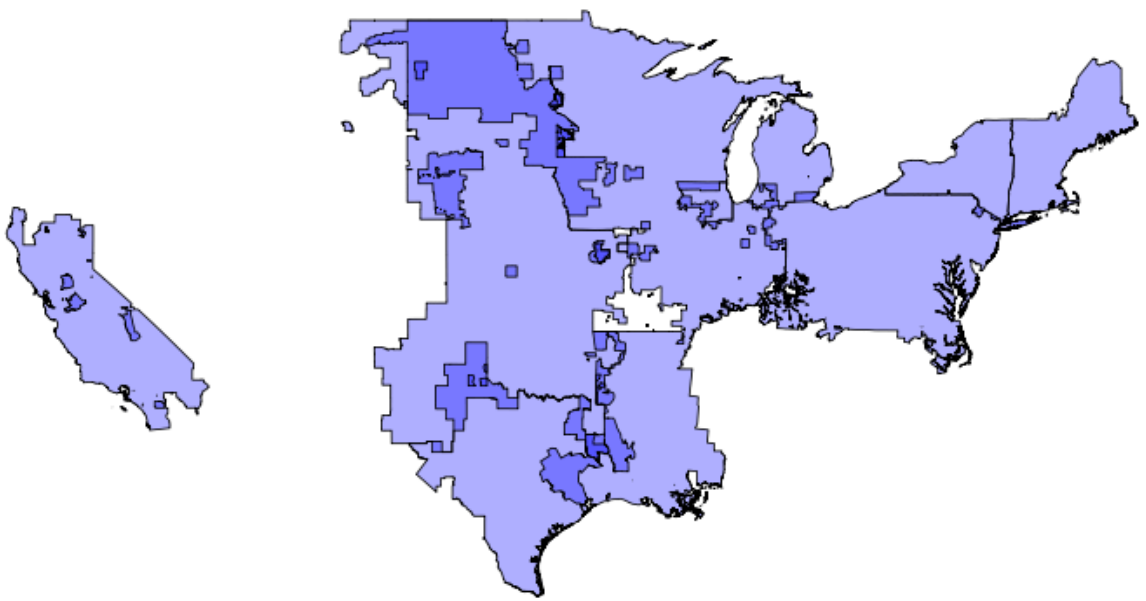
Improving our map: boundaries

```
fig, ax = plt.subplots()
iso_boundaries.plot(ax=ax, color="black", linewidth=0.6)
iso_gdf.plot(ax=ax, facecolor = 'blue', alpha = 0.3).set_axis_off()
```

- Define `fig, ax` so we can plot layers on top of each other
- Use `matplotlib` aesthetic options like `color`, `alpha = 0.3`, `linewidth = 0.6` to customize look

Visualizing at each step

```
1  plt.show()
```



Discussion question: now that we've modified the transparency and added boundaries, what observations can you now make about the ISOs?

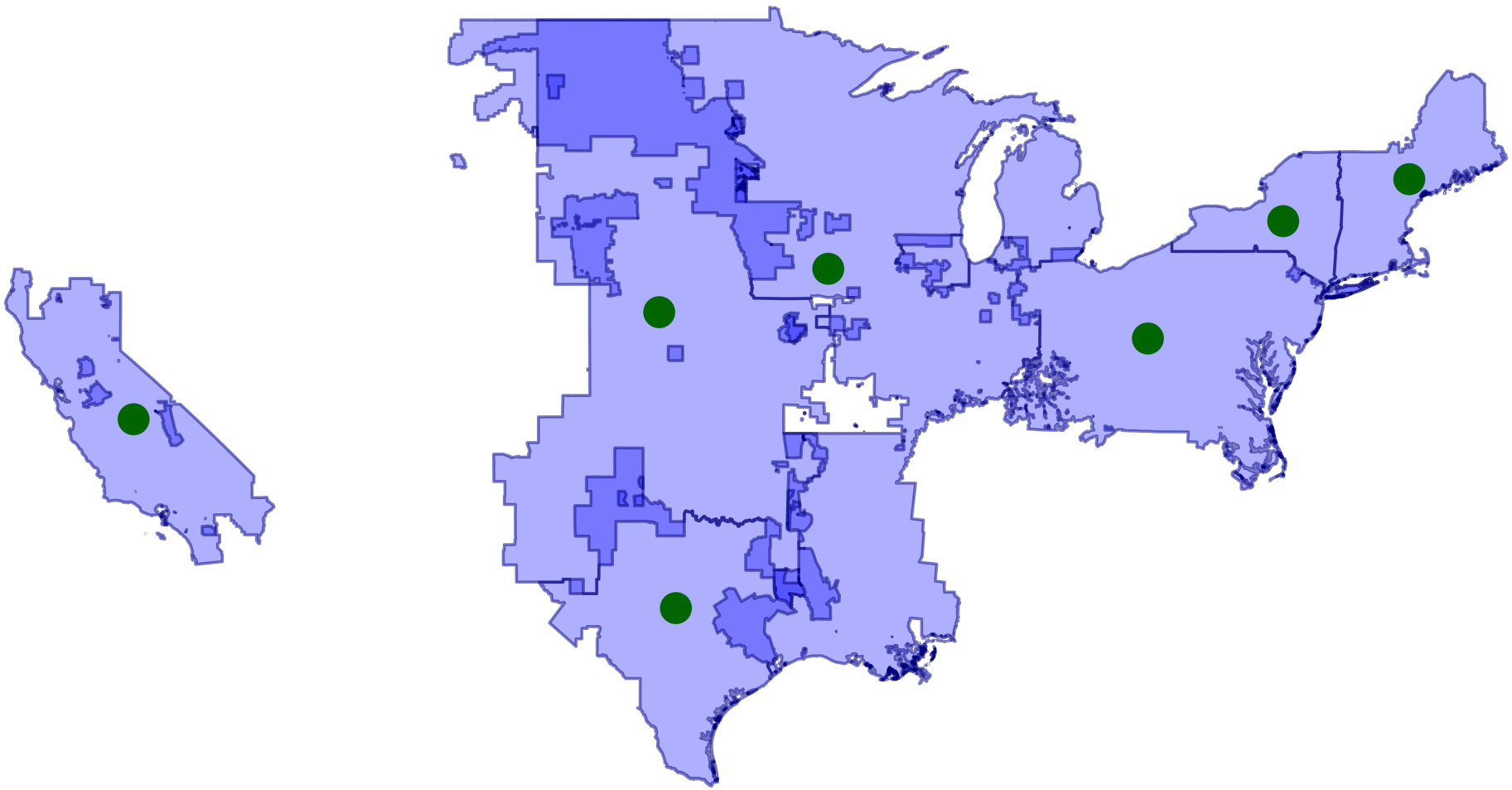
Improving our map: add centroids

- We want to add text labels and position them so they are on top of each ISO.
- Could hand code and pick these coordinates by hand, but this is cumbersome
- Alternatively: use spatial methods to place them at the *centroid* of each ISO

```
# previous layers of plot...
# create centroids and plot in green
iso_gdf.centroid.plot(ax=ax, marker="o", color="darkgreen", markersize=30,
plt.show()
```

- `zorder = 3`: places centroids on *top* of other two layers

Visualizing at each step



Improving our map: add labels

```
# previous layers of plot...
# format and place labels
for _, row in iso_gdf.iterrows():
    ax.text(x=row.centroid.geometry.x, y=row.centroid.geometry.y, ha="left"
        s=row["NAME"],
        fontsize=5, zorder=3,
        path_effects=[
            pe.Stroke(linewidth=2, foreground="white"),  # white halo
            pe.Normal(),]
    )
```

- `.iterrows()`: returns each row's index + data one-by-one
  - *Note*: example of imperative plotting!
- `for _, row`: ignores the index, just gets data

Improving our map: add labels

```
# previous layers of plot...
# format and place labels
for _, row in iso_gdf.iterrows():
    ax.text(x=row.centroid.geometry.x, y=row.centroid.geometry.y, ha="left"
        s=row["NAME"],
        fontsize=5, zorder=3,
        path_effects=[
            pe.Stroke(linewidth=2, foreground="white"),  # white halo
            pe.Normal(),]
    )
```

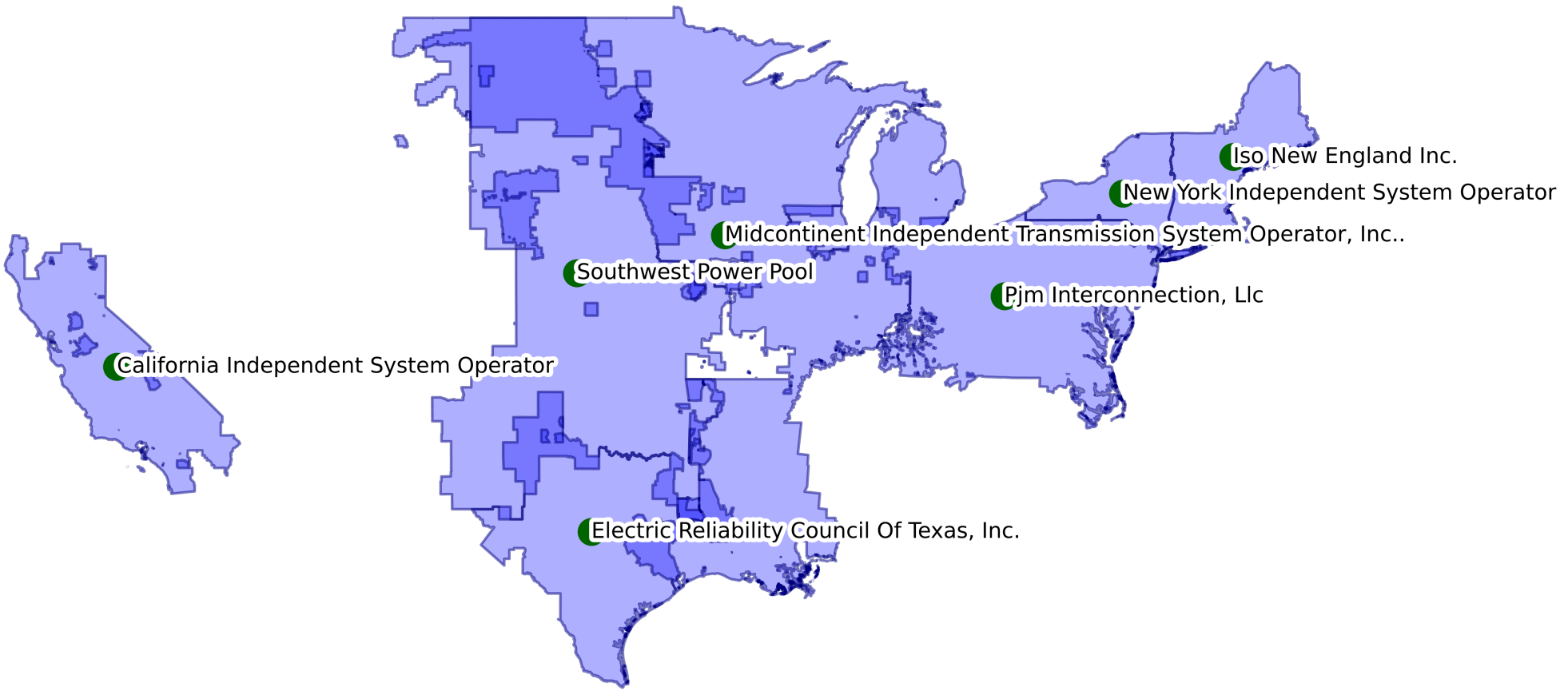
- `row.centroid.geometry.x` and `row.centroid.geometry.y`: extract the x and y coords of centroid of that row
- `ha, va`: horizontal and vertical alignment relative to `x,y`

Improving our map: add labels

```
# define (fig, ax) as before -- see source code
# format and place labels
for _, row in iso_gdf.iterrows():
    ax.text(x=row.centroid.x, y=row.centroid.y, ha="left", va="center",
        s=row["NAME"],
        fontsize=5, zorder=3,
        path_effects=[
            pe.Stroke(linewidth=2, foreground="white"),  # white halo
            pe.Normal(),]
    )
```

- `s`: string to use for label
- `zorder = 3`: places labels on top of two other layers
- `path_effects`: adds white stroke on top of labels

Improving our map: add labels



Improving our map: add context

- The ISO map only covers part of the U.S. – but this may not be immediately apparent
- To provide context, let's add the map of the remaining 48 states

```
# importing our lower 48 states
us_states_gdf = gpd.read_file('data/derived-data/lower_48_states.geojson')
us_states_gdf.crs = iso_gdf.crs
us_states_gdf.plot().set_axis_off()
```

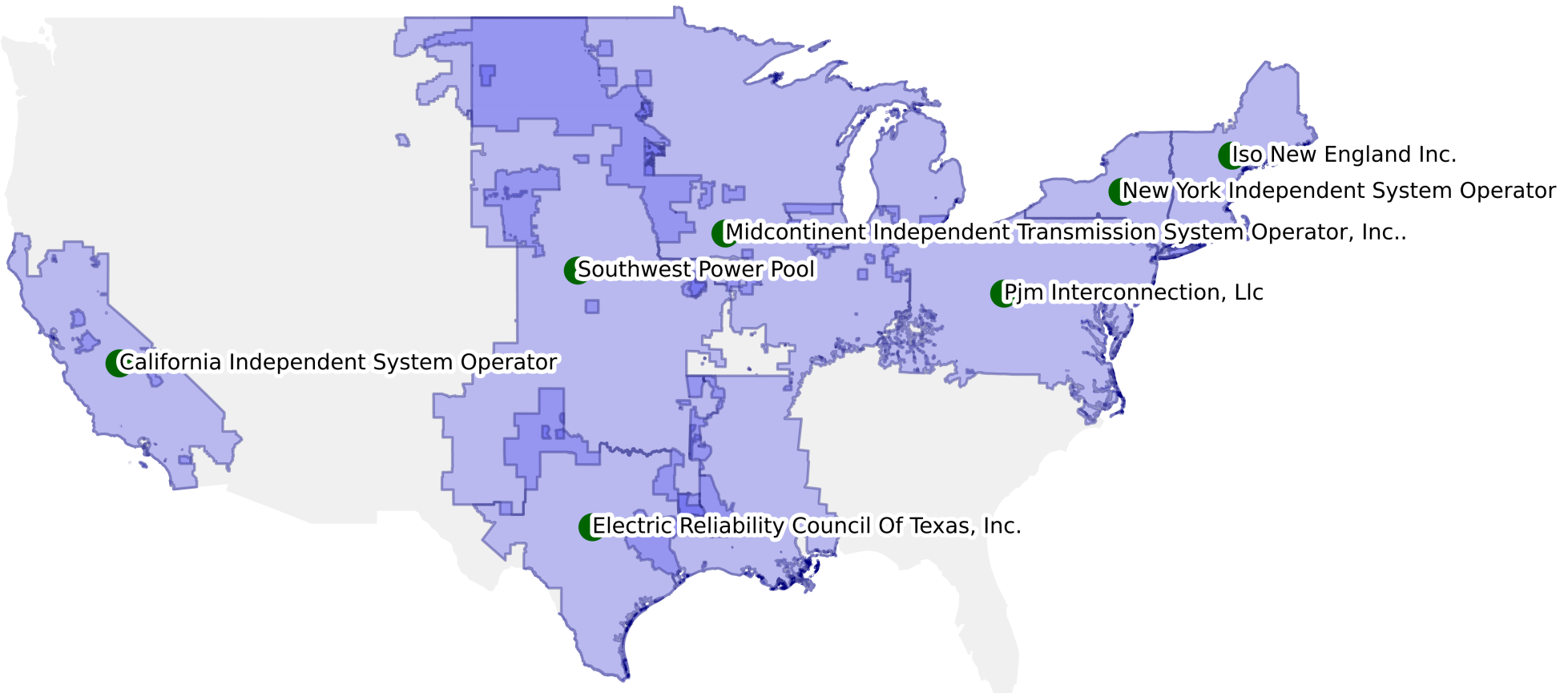
Improving our map: add context

```
# define (fig, ax) as before -- see source code

# plot remaining 48 states below ISO plot
us_states_gdf.plot(ax= ax, color= 'lightgray', edgecolor='none', zorder= 2,
plt.title("Independent Systems Operators in the Lower 48 States")
plt.show()
```

- `ax = ax`: add to same plot as previous layers
- `zorder = 2`: ensures 48 states are plotted *below* ISOs
- `color = 'lightgray'`: color as gray so as to not distract from main focus: ISOs

Improving our map: the final product

Independent Systems Operators in the Lower 48 States

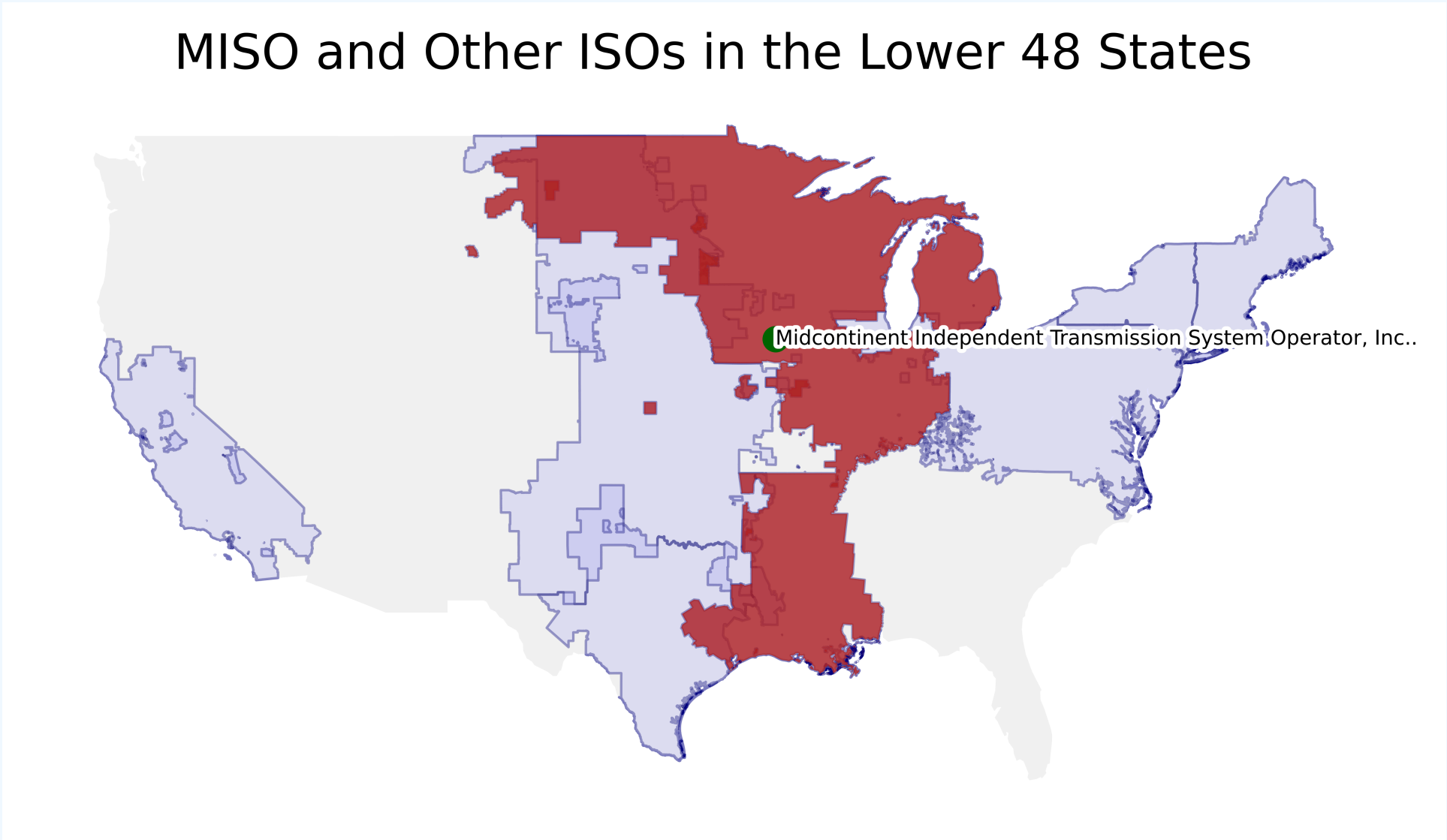


Do-pair-share

Modify the map in `spatial2_dps.qmd` so that:

- The Midwest ISO (MISO) is highlighted in `firebrick`
- The only centroid + label is for MISO
- But still keep the context of other ISOs and rest of U.S.

Do-pair-share: end goal



Use case 3: production quality maps – summary

- Maps can add useful context and help audience better visualize information
- Can use spatial methods to create custom, production-quality maps
  - `centroid` to place labels
  - `boundary` to highlight features
- `matplotlib` tools:
  - `zorder` to specify order of layers
  - `iterrows()` to add text labels one by one

Use Case 4: spatial calculations

Use case 4: spatial calculations – roadmap

- **Motivating question**: for the largest cities in the U.S., what is the total production capacity of nearby power plants?
- Load and inspect cities data
- Working with CRS
  - Harmonizing for plotting
  - Looking up units
- Using spatial methods to create “tools” to answer question: `buffer`

Load data

- Load in cities dataset from `.shp`
- The original format of the spatial data doesn't matter – `geopandas` loads it all in as GeoDataFrame

```
1 us_cities_gdf = gpd.read_file('data/derived-data/cities_greater_500k.shp')
2
3 print("Geometry type:", us_cities_gdf.geom_type.value_counts())
4 print("CRS of us_cities_gdf:", us_cities_gdf.crs)
```

```
Geometry type: Point    33
Name: count, dtype: int64
CRS of us_cities_gdf: EPSG:3347
```

Load data

```
1 print(us_cities_gdf.head())
```

```
  city_fips cbsa_fips           city st_fips   pop_2010  \
0   0455000     38060        Phoenix      04  1445632.0
1   0477000     46060         Tucson      04   520116.0
2   0644000     31080    Los Angeles      06  3792621.0
3   0666000     41740      San Diego      06  1307402.0
4   0667000     41860  San Francisco      06   805235.0

                        geometry
0  POINT (4188179.251 -30022.362)
1  POINT (4254222.09 -231751.035)
2  POINT (3618422.756 259366.151)
3   POINT (3676500.155 65351.387)
4  POINT (3454644.566 819013.232)
```

Powerplant dataset from last class

```
# power plants
us_powerplant_df = pd.read_csv('data/derived-data/us_gppd.csv')
us_powerplant_gdf = gpd.GeoDataFrame(
    geometry=gpd.points_from_xy(
        us_powerplant_df.longitude,
        us_powerplant_df.latitude,
        crs="EPSG:4326"),
    data=us_powerplant_df)
```

CRS comparison

```
1 print("CRS of us_cities_gdf:", us_cities_gdf.crs)
2 print("CRS of us_states_gdf:", us_states_gdf.crs)
3 print("CRS of us_powerplant_gdf:", us_powerplant_gdf.crs)
```

```
CRS of us_cities_gdf: EPSG:3347
CRS of us_states_gdf: EPSG:4326
CRS of us_powerplant_gdf: EPSG:4326
```

- Note that the CRS of `us_cities_gdf` does *not* match `us_states_gdf` or `us_powerplant_gdf`!
- Is this an issue? Let's plot and see…

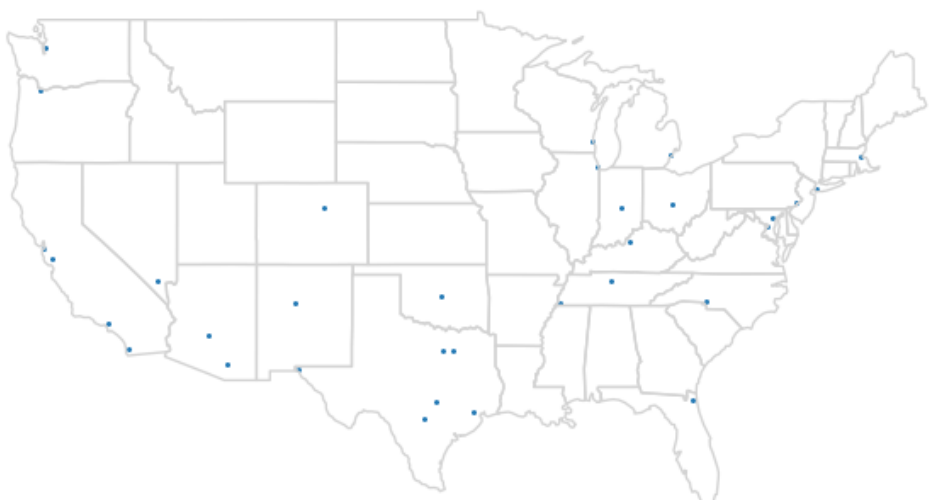
Plotting *without* harmonizing CRS

```
fig, ax = plt.subplots(figsize=(8, 6))
us_states_gdf.boundary.plot(ax=ax, color='lightgray', linewidth=1)
us_cities_gdf.plot(ax=ax, markersize=1.5)
ax.set_axis_off()
plt.show()
```

Plotting *after* harmonizing CRS

```
1  us_cities_gdf = us_cities_gdf.to_crs(us_states_gdf.crs)
2  print("CRS of (new) us_powerplant_gdf:", us_powerplant_gdf.crs)
3
4  fig, ax = plt.subplots(figsize=(8, 6))
5  us_states_gdf.boundary.plot(ax=ax, color='lightgray', linewidth=1)
6  us_cities_gdf.plot(ax=ax, markersize=1.5)
7  ax.set_axis_off()
8  plt.show()
```

```
CRS of (new) us_powerplant_gdf: EPSG:4326
```



Plan to answer our question

- **Motivating question**: for the largest cities in the U.S., what is the total production capacity of nearby power plants?
- Steps
  1. Find all power plants within a certain distance of each city
  2. Sum up production capacity across power plants near each city
  3. Plot

1. Find all power plants within a certain distance

Can we just use `sjoin` between `us_powerplant_gdf` and `us_cities` to find the nearby power plants?

```
powerplants_near_cities_gdf = gpd.sjoin(us_powerplant_gdf,
    us_cities_gdf,
    how='inner', predicate='intersects')
print("Shape:", powerplants_near_cities_gdf.geometry.head())
```

```
GeoSeries([], Name: geometry, dtype: geometry)
```

No – we get an empty `GeoDataFrame`!

Issues with using `sjoin` here

- `us_powerplant_gdf` and `us_cities_gdf` are both `Point` geometries – generally won't overlap unless they have the exact same coordinates
- Furthermore: no notion of how close we mean by "nearby"
- **Solution**: use spatial methods to create a "tool" to join on
  - Turn `us_cities_gdf` into a `Polygon` geometry using `.buffer()`
  - Then apply `sjoin` with `us_cities_gdf`

Constructing a 100km buffer

- Let's say we want to construct **100 km buffer** around each city
- First need to check what unit the CRS calculates distance in:

```
1 print(us_cities_gdf.crs.axis_info)
```

```
[Axis(name=Geodetic latitude, abbrev=Lat, direction=north,
unit_auth_code=EPSG, unit_code=9122, unit_name=degree), Axis(name=Geodetic
longitude, abbrev=Lon, direction=east, unit_auth_code=EPSG, unit_code=9122,
unit_name=degree)]
```

`unit_name = degree` – *not* meters!

Constructing a 100km buffer

So we need to re-project `us_cities_gdf` to a CRS with unit meters. We'll use **EPSG 5070 (NAD 83)**

Attributes

Unit: metre

Geodetic CRS: NAD83

Datum: North American Datum 1983

Ellipsoid: GRS 1980

Scope: Data analysis and small scale data presentation for contiguous lower 48 states.

Remarks: Replaces NAD27 / Conus Albers (CRS code 5069). For applications with an accuracy of better than 1m, replaced by NAD83(HARN) / Conus Albers (CRS code 5071).

Source: epsg.io/5070

Constructing a 100km buffer

```
us_cities_100km_gdf = us_cities_gdf.to_crs(epsg=5070)
us_cities_100km_gdf["geometry"] = us_cities_100km_gdf.buffer(100000)
us_cities_100km_gdf = us_cities_100km_gdf.to_crs(epsg=4326)
```

1. Re-project `us_cities_gdf` into **EPSG 5070**
    - At this point, `us_cities_100km_gdf` geometry type is still `Point`
2. Override `geometry` with 100km buffer
    - This modifies the geometry to be `Polygon`
3. Re-project `us_cities_100km_gdf` back into **EPSG 4326** to be consistent with other layers

Inspecting `us_cities_100km_gdf`

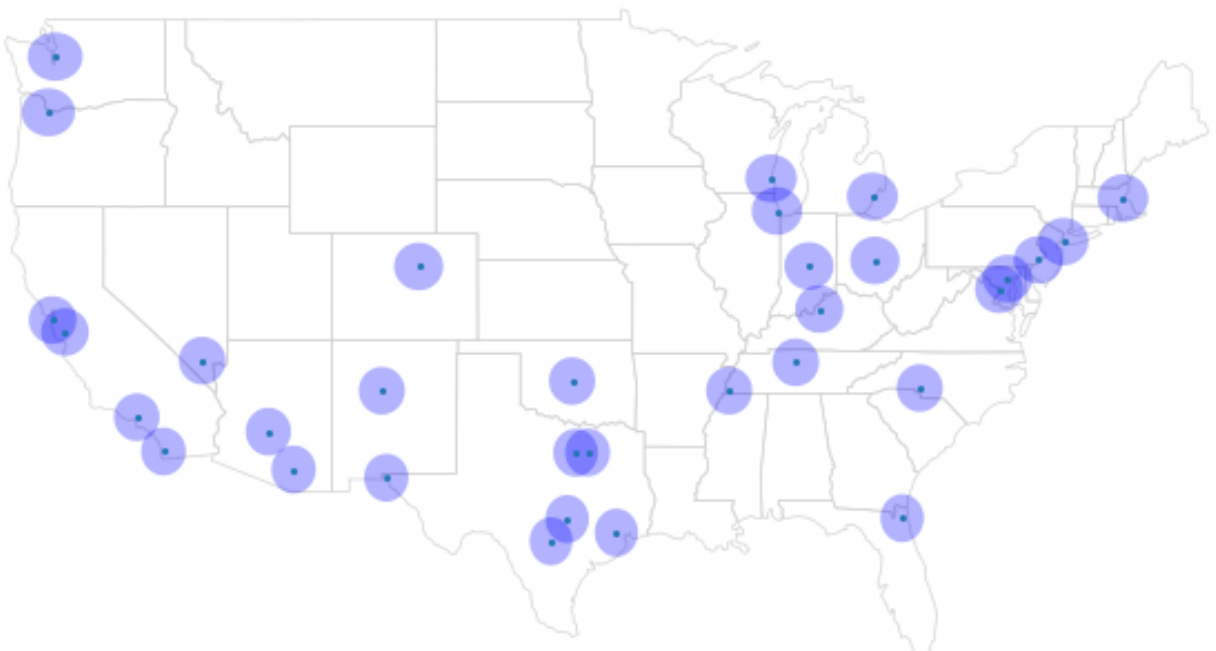
```
1 print(us_cities_100km_gdf.head())
```

```
  city_fips cbsa_fips           city st_fips   pop_2010  \
0   0455000     38060        Phoenix      04  1445632.0
1   0477000     46060         Tucson      04   520116.0
2   0644000     31080    Los Angeles      06  3792621.0
3   0666000     41740      San Diego      06  1307402.0
4   0667000     41860  San Francisco      06   805235.0

                                            geometry
0  POLYGON ((-111.01793 33.71736, -111.00635 33.6...
1  POLYGON ((-109.82395 32.2893, -109.81389 32.20...
2  POLYGON ((-117.34169 34.31756, -117.32295 34.2...
3  POLYGON ((-116.07184 33.02341, -116.0548 32.93...
4  POLYGON ((-121.33763 37.99624, -121.31331 37.9...
```

`us_cities_100km_gdf` has inherited the same attributes as `us_cities_gdf`, except now `geometry == POLYGON`

Plot to verify buffer

```
fig, ax = plt.subplots(figsize=(8, 6))
us_states_gdf.boundary.plot(ax=ax, color='lightgray', linewidth=0.5)
us_cities_100km_gdf.plot(ax=ax, color='blue', alpha=0.3, zorder = 2)
us_cities_gdf.plot(ax=ax, markersize=1.5, zorder = 3)
ax.set_axis_off()
plt.show()
```



Using `sjoin` with 100km buffer

```
1 powerplants_within_100km_gdf = gpd.sjoin(us_powerplant_gdf,
2     us_cities_100km_gdf,
3     how='inner', predicate='intersects')
4 print(powerplants_within_100km_gdf.geom_type.value_counts())
```

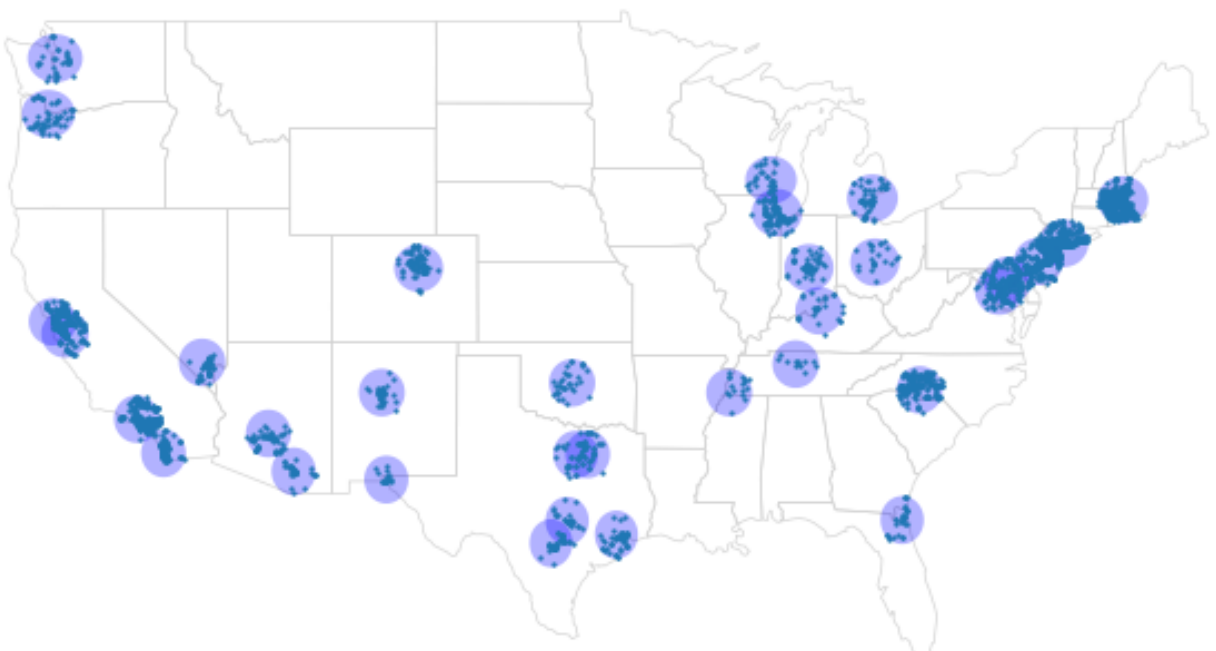
```
Point    3325
Name: count, dtype: int64
```

Discussion questions:

- what is the left geodataframe?
- What is the right?
- What will the geometry of `powerplants_within_100km_gdf` be based on?

Plot to verify the `sjoin`

```
fig, ax = plt.subplots(figsize=(8, 6))
us_states_gdf.boundary.plot(ax=ax, color='lightgray', linewidth=0.5)
us_cities_100km_gdf.plot(ax=ax, color='blue', alpha=0.3, zorder = 2)
powerplants_within_100km_gdf.plot(ax=ax, markersize=1.5, zorder = 3)
ax.set_axis_off()
plt.show()
```



Inspect the dataset

```
1 print(powerplants_within_100km_gdf.columns)
```

```
Index(['country', 'country_long', 'name', 'gppd_idnr', 'capacity_mw',
       'latitude', 'longitude', 'primary_fuel', 'other_fuel1', 'other_fuel2',
       'other_fuel3', 'commissioning_year', 'owner', 'source', 'url',
       'geolocation_source', 'wepp_id', 'year_of_capacity_data',
       'generation_gwh_2013', 'generation_gwh_2014', 'generation_gwh_2015',
       'generation_gwh_2016', 'generation_gwh_2017', 'generation_gwh_2018',
       'generation_gwh_2019', 'generation_data_source',
       'estimated_generation_gwh_2013', 'estimated_generation_gwh_2014',
       'estimated_generation_gwh_2015', 'estimated_generation_gwh_2016',
       'estimated_generation_gwh_2017', 'estimated_generation_note_2013',
       'estimated_generation_note_2014', 'estimated_generation_note_2015',
       'estimated_generation_note_2016', 'estimated_generation_note_2017',
       'geometry', 'index_right', 'city_fips', 'cbsa_fips', 'city', 'st_fips',
       'pop_2010'],
      dtype='object')
```

After the `sjoin`, we now have gwh from power plants *and* city names

2. Sum up total generation by city

```
generation_by_city = (powerplants_within_100km_gdf
    .groupby('city_fips')['generation_gwh_2019']
    .sum()
    .reset_index(name='total_generation_gwh_100km')
)

us_cities_gdf = us_cities_gdf.merge(
    generation_by_city,
    on='city_fips',
    how='left'
)
```

- For each `city_fips`, sum up `generation_gwh_2019` among nearby power plants
- Then use a regular `pandas` merge to merge this new variable back in to `us_cities_gdf`

Inspecting us_cities_gdf

```
1 print(us_cities_gdf.head())
```

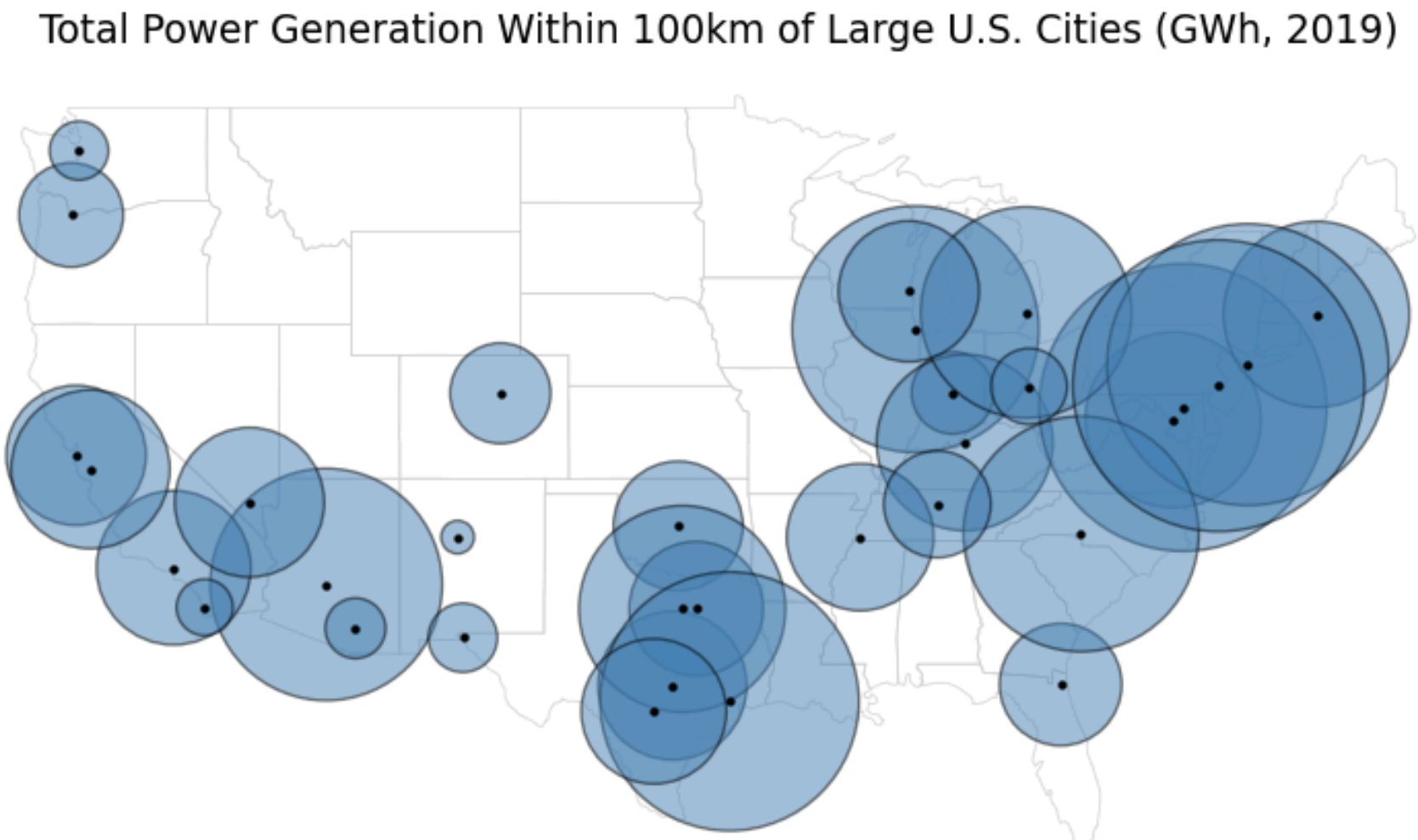
```
  city_fips cbsa_fips           city st_fips   pop_2010  \
0   0455000     38060        Phoenix      04  1445632.0
1   0477000     46060         Tucson      04   520116.0
2   0644000     31080    Los Angeles      06  3792621.0
3   0666000     41740      San Diego      06  1307402.0
4   0667000     41860  San Francisco      06   805235.0

                     geometry  total_generation_gwh_100km
0   POINT (-112.08906 33.5715)                71285.407290
1  POINT (-110.87812 32.15433)                 4824.438325
2  POINT (-118.40647 34.11347)                31489.796582
3   POINT (-117.12247 32.8307)                 4202.462840
4  POINT (-122.44305 37.75616)                25909.678010
```

3. Plot total generation

```
fig, ax = plt.subplots(figsize=(10, 12))

us_states_gdf.boundary.plot(ax=ax, color='lightgray', linewidth=0.5)

us_cities_gdf.plot(
    ax=ax, color='steelblue', edgecolor= 'black', alpha=0.5,
    markersize=us_cities_gdf['total_generation_gwh_100km'].fillna(0) / 10,
    zorder = 2)

us_cities_gdf.plot(ax=ax, markersize=5, color= 'black', zorder = 3)

ax.set_title("Total Power Generation Within 100km of Large U.S. Cities (GWh
ax.text(0.5, -0.04,
    "Marker size is proportional to total power generation within 100 km (G
ax.set_axis_off()
plt.show()
```

Encodes size of marker around each city to scale with `total_generation_gwh_100km`

3. Plot total generation



Use case 4: spatial calculations – summary

- When combining plots, always check if your CRS's are the same – it won't necessarily throw an error
- If calculating distance, make sure you're using a CRS with the correct unit
- Can use spatial methods to construct "tools" (like a `buffer`) to do calculations that don't show up in the final plot

Advanced spatial methods – summary

We've covered several use cases for advanced spatial methods:

1. Joining to characterize spatial relationships
2. Communicating spatial patterns
3. Making production-quality maps
4. Doing spatial calculations

There are *many* more methods that we won't have time to cover.

Overall Takeaways

- Build step by step, and inspect the GeoDataFrame + plot at each step
- Spatial methods can be computationally intensive
  - Pre-process your data as much as possible
  - Consider multiple ways to do something; time them
- Like any other kind of visualization, maps can suffer from overplotting or poor encoding
  - Still have to be thoughtful about your audience, headline message, and submessages
  - Make many plots!