GEOG 4/590: Geospatial Data Science

Lecture 2: Vector data analysis

Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 15:00-16:00 and Tuesday 14:00-15:00

Vector data analysis

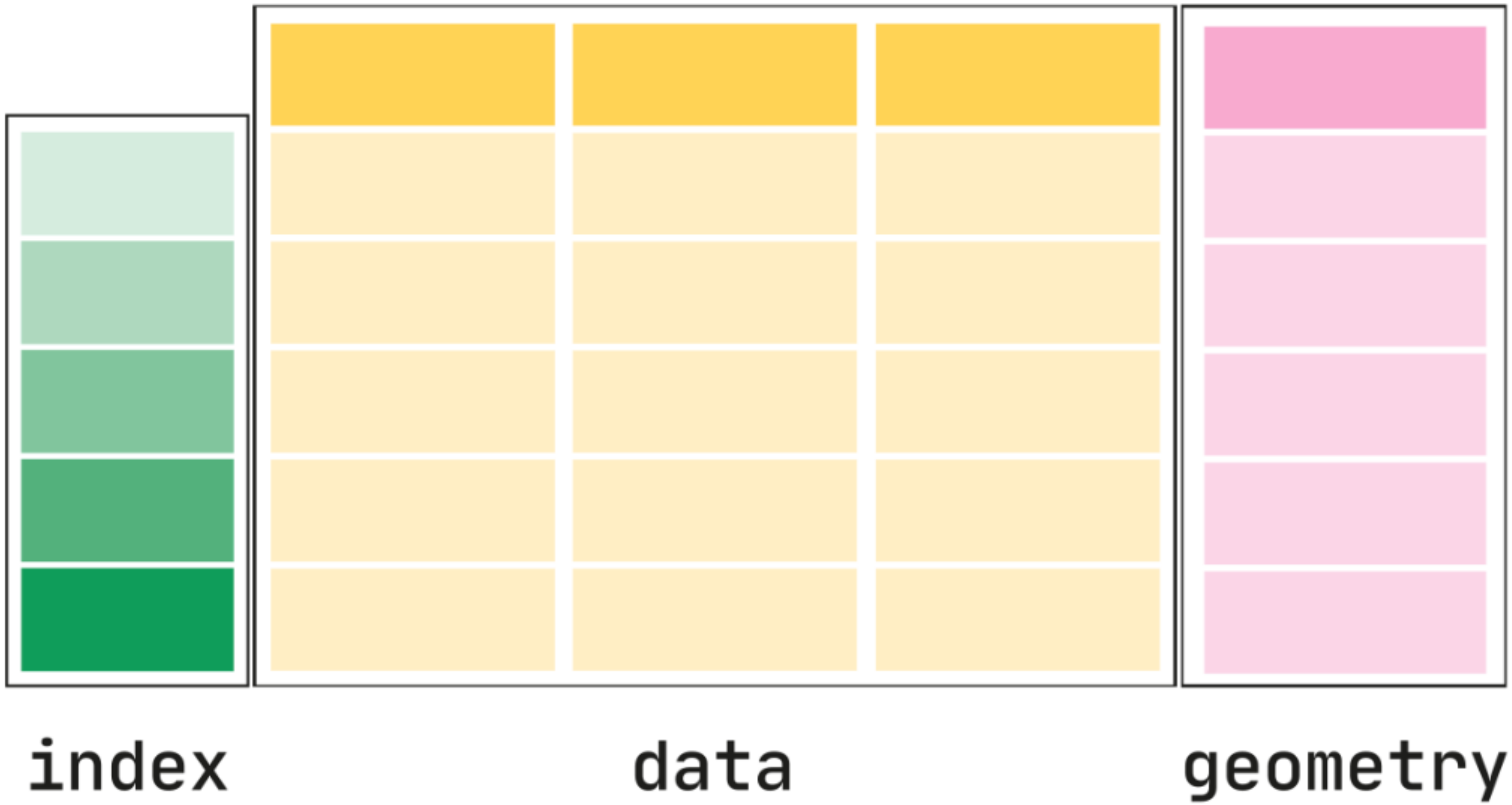
The vector data model represents space as a series of discrete entities such as such as borders, buildings, streets, and roads. There are three different types of vector data: points, lines and polygons. Online mapping applications, such as **Google Maps** and **OpenStreetMap**, use this format to display data.

Vector data analysis

The vector data model represents space as a series of discrete entities such as such as borders, buildings, streets, and roads. There are three different types of vector data: points, lines and polygons. Online mapping applications, such as **Google Maps** and **OpenStreetMap**, use this format to display data.

The Python library `GeoPandas` provides somes great tools for working with vector data. As the name suggests, `GeoPandas` extends the popular data science library `Pandas` by adding support for geospatial data. The core data structure in `GeoPandas` is the `GeoDataFrame`. The key difference between the two is that a `GeoDataFrame` can store geometry data and perform spatial operations.

Vector data analysis



The `geometry` column can contain any geometry type (e.g. points, lines, polygons) or even a mixture.

Reading files

Assuming we have a file containing both data and geometry (e.g. GeoPackage, GeoJSON, Shapefile), we can read it using `read_file`, which automatically detects the filetype and creates a `GeoDataFrame`. In the this demo, we will be working with three shapefiles containing 1) cities and towns (as points), 2) urban growth boundaries (as polygons), and 3) counties (as polygons) in Oregon.

Reading files

Assuming we have a file containing both data and geometry (e.g. GeoPackage, GeoJSON, Shapefile), we can read it using `read_file`, which automatically detects the filetype and creates a `GeoDataFrame`. In the this demo, we will be working with three shapefiles containing 1) cities and towns (as points), 2) urban growth boundaries (as polygons), and 3) counties (as polygons) in Oregon.

```python
import geopandas as gpd

cities = gpd.read_file('data/oregon_cities.shp')
cities.head()
```

Reading files

Assuming we have a file containing both data and geometry (e.g. GeoPackage, GeoJSON, Shapefile), we can read it using `read_file`, which automatically detects the filetype and creates a `GeoDataFrame`. In the this demo, we will be working with three shapefiles containing 1) cities and towns (as points), 2) urban growth boundaries (as polygons), and 3) counties (as polygons) in Oregon.

```
import geopandas as gpd

cities = gpd.read_file('data/oregon_cities.shp')
cities.head()
```

	name	lat	lon	geometry
0	Adair Village city	44.67	-123.22	POINT (-123.22000 44.67000)
1	Adams	45.77	-118.56	POINT (-118.56000 45.77000)
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
3	Albany	44.63	-123.10	POINT (-123.10000 44.63000)
4	Aloha	45.49	-122.87	POINT (-122.87000 45.49000)

DataFrame properties

We can analyze our `GeoDataFrame` using standard `Pandas` functions.

```
# Data types of each column
cities.dtypes
```

```
name             object
lat             float64
lon             float64
geometry       geometry
dtype: object
```

	name	lat	lon	geometry
0	Adair Village city	44.67	-123.22	POINT (-123.22000 44.67000)
1	Adams	45.77	-118.56	POINT (-118.56000 45.77000)
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
3	Albany	44.63	-123.10	POINT (-123.10000 44.63000)
4	Aloha	45.49	-122.87	POINT (-122.87000 45.49000)

DataFrame properties

We can analyze our `GeoDataFrame` using standard `Pandas` functions.

```
# Number of rows and columns
cities.shape
```

```
(377, 4)
```

	name	lat	lon	geometry
0	Adair Village city	44.67	-123.22	POINT (-123.22000 44.67000)
1	Adams	45.77	-118.56	POINT (-118.56000 45.77000)
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
3	Albany	44.63	-123.10	POINT (-123.10000 44.63000)
4	Aloha	45.49	-122.87	POINT (-122.87000 45.49000)

DataFrame properties

We can analyze our `GeoDataFrame` using standard `Pandas` functions.

```
# Name of columns
cities.columns
```

```
Index(['name', 'lat', 'lon', 'geometry'], dtype='object')
```

	name	lat	lon	geometry
0	Adair Village city	44.67	-123.22	POINT (-123.22000 44.67000)
1	Adams	45.77	-118.56	POINT (-118.56000 45.77000)
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
3	Albany	44.63	-123.10	POINT (-123.10000 44.63000)
4	Aloha	45.49	-122.87	POINT (-122.87000 45.49000)

Indexing

We can select specific columns based on the column values. The basic syntax is `dataframe[value]`, where `value` can be a single column name, or a list of column names.

```
# List the city names
cities['name']
```

```
0      Adair Village city
1                   Adams
2                  Adrian
3                  Albany
4                   Aloha
              ...
372          Wood Village
373              Woodburn
374               Yachats
375               Yamhill
376              Yoncalla
Name: name, Length: 377, dtype: object
```

	name	lat	lon	geometry
0	Adair Village city	44.67	-123.22	POINT (-123.22000 44.67000)
1	Adams	45.77	-118.56	POINT (-118.56000 45.77000)
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
3	Albany	44.63	-123.10	POINT (-123.10000 44.63000)
4	Aloha	45.49	-122.87	POINT (-122.87000 45.49000)

Indexing

We can select specific columns based on the column values. The basic syntax is `dataframe[value]`, where `value` can be a single column name, or a list of column names.

```
# List the latitudes and longitudes
cities[['lat','lon']]
```

	lat	lon
0	44.67	-123.22
1	45.77	-118.56
2	43.74	-117.07
3	44.63	-123.10
4	45.49	-122.87
...	...	...
372	45.54	-122.42
373	45.15	-122.86
374	44.31	-124.10
375	45.34	-123.19
376	43.60	-123.29

377 rows × 2 columns

Indexing

We can select specific rows using the `.iloc` method.

```
# Second row
cities.iloc[1]
```

```
name                           Adams
lat                            45.77
lon                          -118.56
geometry    POINT (-118.56 45.77)
Name: 1, dtype: object
```

Indexing

We can select specific rows using the `.iloc` method.

```
# Sixth to tenth rows
cities.iloc[5:10]
```

	name	lat	lon	geometry
5	Alpine	44.33	-123.36	POINT (-123.36000 44.33000)
6	Alsea	44.38	-123.60	POINT (-123.60000 44.38000)
7	Altamont	42.20	-121.72	POINT (-121.72000 42.20000)
8	Amity	45.12	-123.20	POINT (-123.20000 45.12000)
9	Annex	44.23	-116.99	POINT (-116.99000 44.23000)

Masking

We can sample of our `DataFrame` based on specific values by producing a **Boolean mask** (i.e. a list of values equal to `True` or `False`). To find cities that are East of -117.5 degrees longitude, we could write:

```
mask = cities['lon'] > -117.5
cities[mask]
```

Masking

We can sample of our `DataFrame` based on specific values by producing a **Boolean mask** (i.e. a list of values equal to `True` or `False`). To find cities that are East of -117.5 degrees longitude, we could write:

```
mask = cities['lon'] > -117.5
cities[mask]
```

	name	lat	lon	geometry
2	Adrian	43.74	-117.07	POINT (-117.07000 43.74000)
9	Annex	44.23	-116.99	POINT (-116.99000 44.23000)
97	Enterprise	45.43	-117.28	POINT (-117.28000 45.43000)
134	Halfway	44.88	-117.11	POINT (-117.11000 44.88000)
150	Huntington	44.35	-117.27	POINT (-117.27000 44.35000)
164	Jordan Valley	42.98	-117.06	POINT (-117.06000 42.98000)
165	Joseph	45.35	-117.23	POINT (-117.23000 45.35000)
190	Lostine	45.49	-117.43	POINT (-117.43000 45.49000)

Masking

It's more concise to just add the Boolean mask between square brackets. Here we find cities that contain a `z` in their name.

```
cities[cities['name'].str.contains('z')]
```

	name	lat	lon	geometry
34	Bonanza	42.20	-121.41	POINT (-121.41000 42.20000)
168	Keizer	45.00	-123.02	POINT (-123.02000 45.00000)
195	Manzanita	45.72	-123.94	POINT (-123.94000 45.72000)
206	Metzger	45.45	-122.76	POINT (-122.76000 45.45000)
302	Siletz	44.72	-123.92	POINT (-123.92000 44.72000)

Masking

Or use string matching to find a specific city.

```
cities[cities['name'] == 'Eugene']
```

	name	lat	lon	geometry
100	Eugene	44.06	-123.12	POINT (-123.12000 44.06000)

Descriptive statistics

`Pandas` provides basic functions to calculate descriptive statistics.

```
# Minimum latitude value
cities['lat'].min()
```

```
42.0
```

Descriptive statistics

`Pandas` provides basic functions to calculate descriptive statistics.

```
# Minimum latitude value
cities['lat'].min()
```

```
42.0
```

```
# Mean longitude value
cities['lon'].mean()
```

```
-122.02392572944296
```

Descriptive statistics

`Pandas` provides basic functions to calculate descriptive statistics.

Sometimes we want to know which row contains the specific value which we can do using `idxmax`/`idxmin`.

```
cities['lat'].idxmin()
```

```
232
```

Descriptive statistics

`Pandas` provides basic functions to calculate descriptive statistics.

Sometimes we want to know which row contains the specific value which we can do using `idxmax`/`idxmin`.

```
cities['lat'].idxmin()
```

```
232
```

```
cities.iloc[232]
```

```
name            New Pine Creek
lat                       42.0
lon                     -120.3
geometry    POINT (-120.3 42)
Name: 232, dtype: object
```

Sorting

We can sort `DataFrames` using the `sort_values` function. This function takes two arguments, `by` and `ascending` which determine which column and which order we would like to sort by.

```
# Find the ten most northerly cities in Oregon
cities.sort_values(by='lat', ascending=False).head(10)
```

Sorting

We can sort `DataFrames` using the `sort_values` function. This function takes two arguments, `by` and `ascending` which determine which column and which order we would like to sort by.

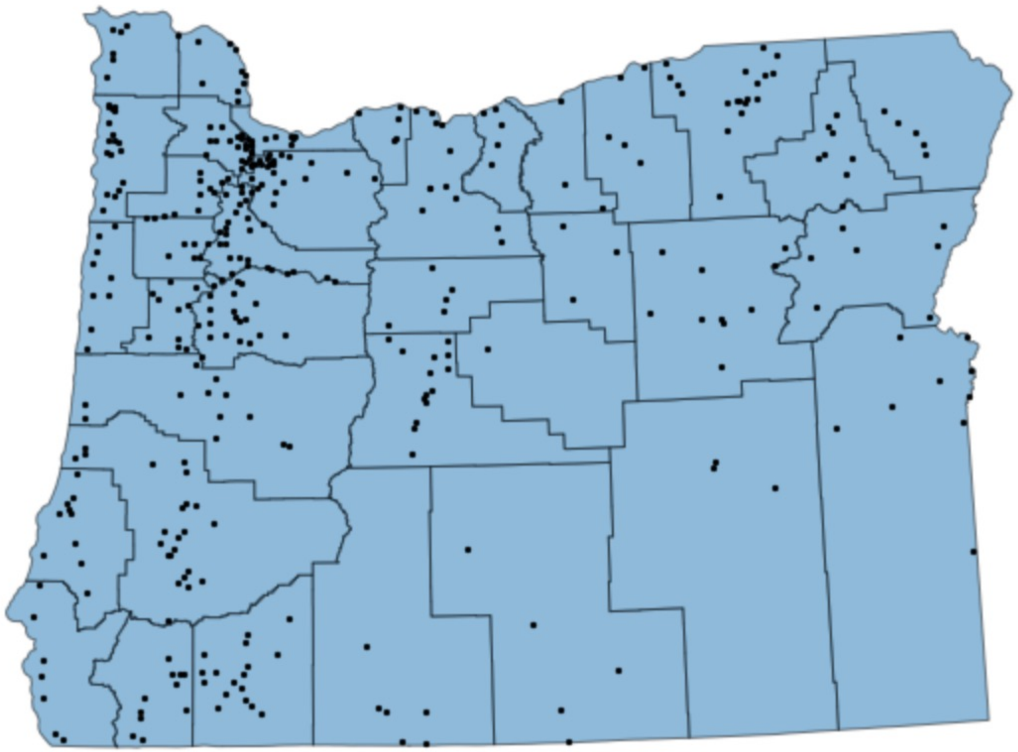
```
# Find the ten most northerly cities in Oregon
cities.sort_values(by='lat', ascending=False).head(10)
```

	name	lat	lon	geometry
13	Astoria	46.19	-123.81	POINT (-123.81000 46.19000)
354	Warrenton	46.17	-123.92	POINT (-123.92000 46.17000)
159	Jeffers Gardens	46.15	-123.85	POINT (-123.85000 46.15000)
363	Westport	46.13	-123.37	POINT (-123.37000 46.13000)
60	Clatskanie	46.10	-123.21	POINT (-123.21000 46.10000)
269	Rainier	46.09	-122.95	POINT (-122.95000 46.09000)

Geometric properties

The special thing about a `GeoDataFrame` is that it contains a `geometry` column. We can therefore apply spatial methods to these data. To demonstrate we will use our Oregon county shapefile.

```
# Read shapefile
counties = gpd.read_file('data/orcntypoly.shp')
```



Projections

`GeoDataFrames` have their own **CRS** which can be accessed using the `crs` method. The CRS tells `GeoPandas` where the coordinates of the geometries are located on the Earth's surface.

```
counties.crs
```

Projections

`GeoDataFrames` have their own **CRS** which can be accessed using the `crs` method. The CRS tells `GeoPandas` where the coordinates of the geometries are located on the Earth's surface.

```
counties.crs
```

```
<Compound CRS: EPSG:5498>
Name: NAD83 + NAVD88 height
Axis Info [ellipsoidal|vertical]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
- H[up]: Gravity-related height (metre)
Area of Use:
- name: United States (USA) - CONUS and Alaska - onshore - Alabama; Alaska mainland; Arizona;
- bounds: (-168.26, 24.41, -66.91, 71.4)
Datum: North American Datum 1983
- Ellipsoid: GRS 1980
- Prime Meridian: Greenwich
Sub CRS:
- NAD83
- NAVD88 height
```

Projections

We can reproject a our data using the `to_crs` method.

```
counties_reproject = counties.to_crs('EPSG:32610')
counties_reproject.crs
```

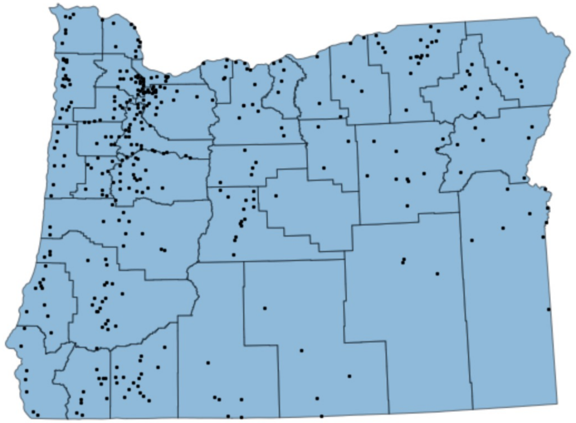
```
<Derived Projected CRS: EPSG:32610>
Name: WGS 84 / UTM zone 10N
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: Between 126°W and 120°W, northern hemisphere between equator and 84°N, onshore and off
- bounds: (-126.0, 0.0, -120.0, 84.0)
Coordinate Operation:
- name: UTM zone 10N
- method: Transverse Mercator
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Geometric properties

Now our data has a **projected CRS**, we can calculate the area of each county with no warnings.

```
counties_reproject['area'] = counties_reproject['geometry'].area
counties_reproject.head()
```

	county	geometry	area
0	Josephine County	POLYGON ((481193.348 4727817.470, 481193.815 4...	4.246561e+09
1	Curry County	POLYGON ((433624.799 4737686.263, 433624.695 4...	5.162416e+09
2	Jackson County	POLYGON ((558466.270 4760676.349, 558468.456 4...	7.249726e+09
3	Coos County	POLYGON ((433624.799 4737686.263, 433230.247 4...	4.684074e+09
4	Klamath County	POLYGON ((634510.330 4830646.470, 634516.612 4...	1.588783e+10

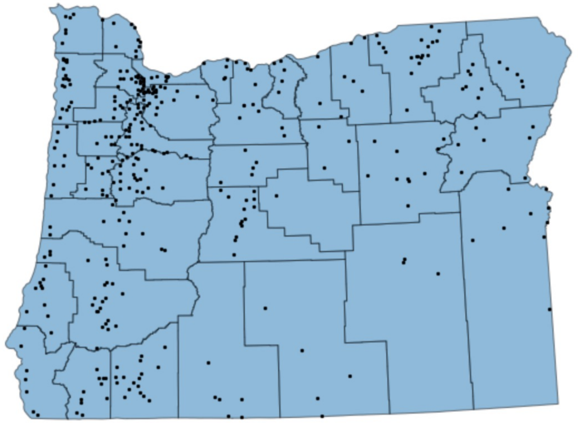


Geometric properties

There are other spatial methods we can apply to polygons such as the length of the outer edge (i.e. perimeter).

```
counties_reproject['perimeter'] = counties_reproject['geometry'].length
counties_reproject.head()
```

	county	geometry	area	perimeter
0	Josephine County	POLYGON ((481193.348 4727817.470, 481193.815 4...	4.246561e+09	331219.627981
1	Curry County	POLYGON ((433624.799 4737686.263, 433624.695 4...	5.162416e+09	438569.090594
2	Jackson County	POLYGON ((558466.270 4760676.349, 558468.456 4...	7.249726e+09	378774.110802
3	Coos County	POLYGON ((433624.799 4737686.263, 433230.247 4...	4.684074e+09	341177.961347
4	Klamath County	POLYGON ((634510.330 4830646.470, 634516.612 4...	1.588783e+10	619485.806580



Geometric properties

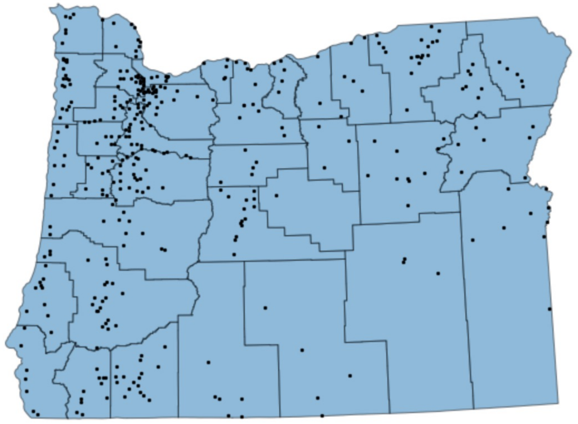
Our `cities` `GeoDataFrame` also has geometric properties. We can access the latitude and longitude using the `x` and `y` methods.

```
cities['geometry'].x
```

```
0      -123.22
1      -118.56
2      -117.07
3      -123.10
4      -122.87
```

```
cities['geometry'].y
```

```
0       44.67
1       45.77
2       43.74
3       44.63
4       45.49
```



Measure distance

We can measure the distance between two points, provided they have a projected CRS.

```
cities_reproject = cities.to_crs('EPSG:32610')
```

Measure distance

We can measure the distance between two points, provided they have a projected CRS.

```
cities_reproject = cities.to_crs('EPSG:32610')
```

```
eugene = cities_reproject[cities_reproject['name'] == 'Eugene'].reset_index()
bend = cities_reproject[cities_reproject['name'] == 'Bend'].reset_index()
```

Measure distance

We can measure the distance between two points, provided they have a projected CRS.

```
cities_reproject = cities.to_crs('EPSG:32610')
```

```
eugene = cities_reproject[cities_reproject['name'] == 'Eugene'].reset_index()
bend = cities_reproject[cities_reproject['name'] == 'Bend'].reset_index()
```

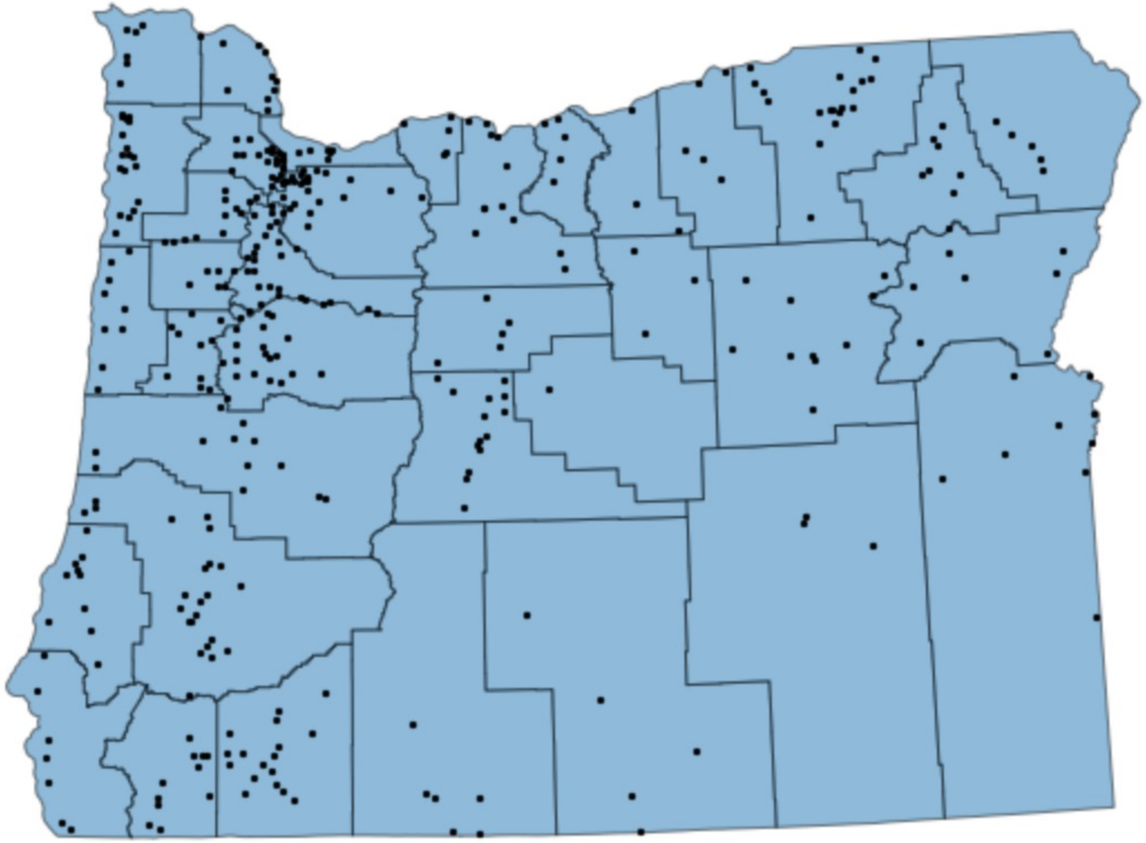
```
eugene.distance(bend).values[0] / 1000
```

```
144.97607871968486
```

Plot

If our data are in the same projection system, we can plot them together.

```
ax = counties_reproject.plot(figsize=(10, 10), alpha=0.5, edgecolor='k')
cities_reproject.plot(ax=ax, color='black', markersize=5)
```



Spatial joins

One of the most useful things about `GeoPandas` is that it contains functions to perform **spatial joins** to combine two `GeoDataFrames` based on the **spatial relationships** between their geometries.

The order of the two `GeoDataFrames` is quite important here, as well as the `how` argument. A **left** outer join implies that we are interested in **retaining the geometries** of the `GeoDataFrame` on the left, i.e. the **point** locations of the cities. We then retain attributes of the **right** `GeoDataFrame` if they intersect and drop them if they don't.

Spatial joins

The following would provide the county attributes for all our cities based on a spatial intersection.

```
cities_reproject.sjoin(counties_reproject, how="left").head()
```

	name	lat	lon	geometry	index_right	county	ar
0	Adair Village city	44.67	-123.22	POINT (482561.392 4946316.184)	12	Benton County	1.755207e+
1	Adams	45.77	-118.56	POINT (845212.127 5078087.252)	33	Umatilla County	8.385719e+
2	Adrian	43.74	-117.07	POINT (977541.425 4860113.062)	10	Malheur County	2.581882e+
3	Albany	44.63	-123.10	POINT (492067.910 4941854.290)	13	Linn County	5.969370e+
4	Aloha	45.49	-122.87	POINT (510158.282 5037393.753)	27	Washington County	1.880526e+

Which county contains the most cities/towns?

We would do this would be to use the `groupby` function.

```
join = cities_reproject.sjoin(counties_reproject, how="left")
```

Which county contains the most cities/towns?

We would do this would be to use the `groupby` function.

The first argument `groupby` accepts is the column we want group our data into (`county` in our case). Next, it takes a column (or list of columns) to summarize. Finally, this function does nothing until we specify **how** we want to group our data (`count`).

It's actually nice to **reset the index** after using `groupby` so that we end up with a DataFrame (rather than a Series).

```
grouped = join.groupby('county')['name'].count().reset_index()
grouped.nlargest(n=10, columns='name')
```

	county	name
23	Marion County	25
33	Washington County	24
2	Clackamas County	23
9	Douglas County	23
21	Linn County	23
29	Umatilla County	19
28	Tillamook County	18
14	Jackson County	17
8	Deschutes County	14
19	Lane County	12

Next time: Network data analysis

Email: jryan4@uoregon.edu
Office: 163A Condon Hall
Office hours: Monday 15:00-16:00 and Tuesday 14:00-15:00