Data processing with Pandas & GeoPandas

CS424: Visualization & Visual Analytics

Fabio Miranda

https://fmiranda.me

UIC COMPUTER SCIENCE

Pandas

- Powerful Python package for manipulating tables.
- Built on top of numpy.
- Save time by abstracting lower-level code for manipulating, extracting, and deriving data tables.
- Easy & quick visualization with matplotlib.
- Main data structures: **Series** and **DataFrame**

Simple series

```
data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5])
```

```
data
```

```
0    0.1
1    0.2
2    0.3
3    0.4
4    0.5
dtype: float64
```

Data

Index

Explicit index

```
data = pd.Series([0.1, 0.2, 0.3, 0.4, 0.5], index = ['a', 'b', 'c', 'd', 'e'])
```

```
data
```

```
a    0.1
b    0.2
c    0.3
d    0.4
e    0.5
dtype: float64
```

Indexing & accessing data

- .loc label based:
  - A single label
  - A list of array of labels
  - A slice object with labels
  - A boolean array
  - A callable function with one argument that returns valid output for indexing (one of the above).
- .iloc integer position based:
  - An integer
  - A list of array of integers
  - A slice object with integers
  - A boolean array
  - A callable function with one argument that returns valid output for indexing (one of the above)

Indexing & accessing data

Selection using label

```
data.loc['a'] # single label
```

```
0.1
```

```
data.loc[['a','b']] # list of labels
```

```
a    0.1
b    0.2
dtype: float64
```

```
data.loc['a':'c'] # slice object with labels
```

```
a    0.1
b    0.2
c    0.3
dtype: float64
```

```
data.loc[[False,False,True,False,False]] # boolean mask
```

```
c    0.3
dtype: float64
```

```
data.loc[lambda x: x.index == 'b'] # callable function
```

```
b    0.2
dtype: float64
```

Selection using integer position

```
data.iloc[0] # scalar integer
```

```
0.1
```

```
data.iloc[[0,1]] # list of integers
```

```
a    0.1
b    0.2
dtype: float64
```

```
data.iloc[0:2] # slice object
```

```
a    0.1
b    0.2
dtype: float64
```

```
data.iloc[[False,False,True,False,False]] # boolean mask
```

```
c    0.3
dtype: float64
```

```
data.iloc[lambda x: x.index == 'b'] # callable function
```

```
b    0.2
dtype: float64
```

Dictionary as a series

```
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
```

```
population = pd.Series(population_dict)
population
```

```
California    38332521
Texas         26448193
New York      19651127
Florida       19552860
Illinois      12882135
dtype: int64
```

```
population.loc['California']
```

```
38332521
```

```
population.loc[population>20000000]
```

Accessing with boolean array

```
California    38332521
Texas         26448193
dtype: int64
```

DataFrame object

- DataFrame is a 2-dimensional labeled data structure with columns of (potentially) different types.
  - Just like a spreadsheet or SQL table, or dict of Series objects.
- DataFrame can be created with:
  - Dict of 1D arrays, lists, dicts, or Series
  - 2D numpy array
  - Series
  - Another DataFrame

Constructing a DataFrame

- From a dictionary or list of dictionaries:

```python
d = {"one": [1.0, 2.0, 3.0, 4.0]}
pd.DataFrame(d)
```

	one
0	1.0
1	2.0
2	3.0
3	4.0

```python
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
pd.DataFrame(d)
```

	one	two
0	1.0	4.0
1	2.0	3.0
2	3.0	2.0
3	4.0	1.0

```python
d = {"one": [1.0, 2.0, 3.0, 4.0], "two": [4.0, 3.0, 2.0, 1.0]}
pd.DataFrame(d, index=["a", "b", "c", "d"])
```

	one	two
a	1.0	4.0
b	2.0	3.0
c	3.0	2.0
d	4.0	1.0

- From numpy ndarray:

```python
pd.DataFrame(np.random.randint(low=0, high=10, size=(5,5)), columns=['a', 'b', 'c', 'd', 'e'])
```

	a	b	c	d	e
0	8	4	6	1	1
1	1	8	3	8	8
2	2	7	9	2	1
3	5	8	4	9	3
4	0	0	6	9	8

Constructing a DataFrame

- From dictionaries or Series

```python
population_dict = {'California': 38332521,
                   'Texas': 26448193,
                   'New York': 19651127,
                   'Florida': 19552860,
                   'Illinois': 12882135}
```

```python
area_dict = {'California': 423967,
             'Texas': 695662,
             'New York': 141297,
             'Florida': 170312,
             'Illinois': 149995}
```

```python
states = pd.DataFrame({'population': population_dict, 'area': area_dict})
states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

Viewing data & statistics

```
states.head(2)
```

	population	area
California	38332521	423967
Texas	26448193	695662

```
states.tail(2)
```

	population	area
Florida	19552860	170312
Illinois	12882135	149995

```
states.describe()
```

	population	area
count	5.000000e+00	5.000000
mean	2.337337e+07	316246.600000
std	9.640386e+06	242437.411951
min	1.288214e+07	141297.000000
25%	1.955286e+07	149995.000000
50%	1.965113e+07	170312.000000
75%	2.644819e+07	423967.000000
max	3.833252e+07	695662.000000

Computing descriptive stats for each column

Viewing sorted DataFrame

```
states
```

	population	area
California	38332521	423967
Texas	26448193	695662
New York	19651127	141297
Florida	19552860	170312
Illinois	12882135	149995

```
states.sort_index()
```

Viewing sorted by index

	population	area
California	38332521	423967
Florida	19552860	170312
Illinois	12882135	149995
New York	19651127	141297
Texas	26448193	695662

```
states.sort_values(by='area')
```

Viewing sorted by column

	population	area
New York	19651127	141297
Illinois	12882135	149995
Florida	19552860	170312
California	38332521	423967
Texas	26448193	695662

Selecting & filtering data

- Selection using integer position

```
states.iloc[0]
```

```
population    38332521
area            423967
Name: California, dtype: int64
```

- Multi-axis selection by label

```
states.loc[:, ['population']]
```

	population
California	38332521
Texas	26448193
New York	19651127
Florida	19552860
Illinois	12882135

```
states.loc[['New York', 'Illinois'], ['population']]
```

	population
New York	19651127
Illinois	12882135

Selecting & filtering data

- Boolean indexing

```
states[states['population'] > 20000000]
```

	population	area
California	38332521	423967
Texas	26448193	695662

```
states[states.index.isin(['New York'])]
```

	population	area
New York	19651127	141297

Operations

```
d = pd.DataFrame(np.random.randint(low=0, high=10, size=(5,5)), columns=['a', 'b', 'c', 'd', 'e'])
```

```
d
```

	a	b	c	d	e
0	2	9	7	7	7
1	5	8	3	7	3
2	8	3	0	0	1
3	1	9	8	0	0
4	4	0	3	9	2

```
d.mean()
```

Across axis 0 (rows), i.e., column mean

```
a    4.0
b    5.8
c    4.2
d    4.6
e    2.6
dtype: float64
```

```
d.mean(axis=1)
```

Across axis 1 (columns), i.e., row mean

```
0    6.4
1    5.2
2    2.4
3    3.6
4    3.6
dtype: float64
```

Operations

```
d.apply(np.cumsum)
```

NumPy's cumulative sum

	a	b	c	d	e
0	2	9	7	7	7
1	7	17	10	14	10
2	15	20	10	14	11
3	16	29	18	14	11
4	20	29	21	23	13

```
states.apply(lambda x: x['population'] / x['area'], axis=1)
```

```
California     90.413926
Texas          38.018740
New York      139.076746
Florida       114.806121
Illinois       85.883763
dtype: float64
```

Population density of each **row**

Merging tables

```
left = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [1,2]})
right = pd.DataFrame({'key': ['foo', 'bar'], 'lval': [4,5]})
```

```
left
```

	key	lval
0	foo	1
1	bar	2

```
right
```

	key	lval
0	foo	4
1	bar	5

```
pd.merge(left, right, on='key')
```

Column or index names to join on

	key	lval_x	lval_y
0	foo	1	4
1	bar	2	5

Grouping

```python
df = pd.DataFrame({'Animal': ['Falcon', 'Falcon',
                              'Parrot', 'Parrot'],
                   'Max Speed': [380., 370., 24., 26.]})
```

```python
df
```

	Animal	Max Speed
0	Falcon	380.0
1	Falcon	370.0
2	Parrot	24.0
3	Parrot	26.0

```python
df.groupby(['Animal']).mean()
```

	Max Speed
Animal	
Falcon	375.0
Parrot	25.0

Grouping

```python
arrays = [['Falcon', 'Falcon', 'Parrot', 'Parrot'],
          ['Captive', 'Wild', 'Captive', 'Wild']]
index = pd.MultiIndex.from_arrays(arrays, names=('Animal', 'Type'))
df = pd.DataFrame({'Max Speed': [390., 350., 30., 20.]}, index=index)
```

```python
df
```

		Max Speed
Animal	**Type**	
Falcon	**Captive**	390.0
	Wild	350.0
Parrot	**Captive**	30.0
	Wild	20.0

```python
df.index
```

```
MultiIndex([('Falcon', 'Captive'),
            ('Falcon',    'Wild'),
            ('Parrot', 'Captive'),
            ('Parrot',    'Wild')],
           names=['Animal', 'Type'])
```

Grouping by index:

```python
df.groupby(level=0).mean()
```

	Max Speed
Animal	
Falcon	370.0
Parrot	25.0

```python
df.groupby(level="Type").mean()
```

	Max Speed
Type	
Captive	210.0
Wild	185.0

Importing & exporting data

- Reading and writing a CSV file:

```
pd.read_csv('data.csv')
```

```
df.to_csv('data.csv')
```

- DataFrame to binary Feather format:

```
df.to_feather('data.feather')
```

Basic plotting with matplotlib

```
ts = pd.Series(np.random.randn(1000), index=pd.date_range("1/1/2000", periods=1000))
df = pd.DataFrame(np.random.randn(1000, 4), index=ts.index, columns=list("ABCD"))
df = df.cumsum()
df
```

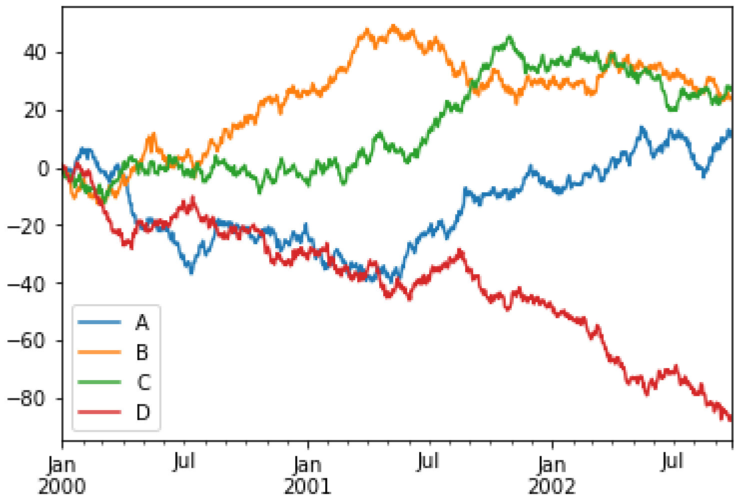
	A	B	C	D
2000-01-01	0.099142	-0.679263	-0.669535	0.971732
2000-01-02	-0.713262	-1.037180	-1.869124	0.314566
2000-01-03	-2.176599	-2.202236	-0.843755	-0.426149
2000-01-04	-1.254498	-2.075695	-2.420534	0.228423
2000-01-05	-0.251042	0.105400	-2.590070	0.277761
...	...	...	...	...
2002-09-22	11.209192	24.387028	27.601228	-87.805667
2002-09-23	12.023897	23.530602	26.630084	-88.124066
2002-09-24	10.766121	23.579338	26.731239	-87.990660
2002-09-25	11.518224	23.913193	27.140907	-86.354709
2002-09-26	12.567776	24.353585	27.994359	-86.652313

1000 rows × 4 columns

Basic plotting with matplotlib

```
plt.figure()
df.plot()
```

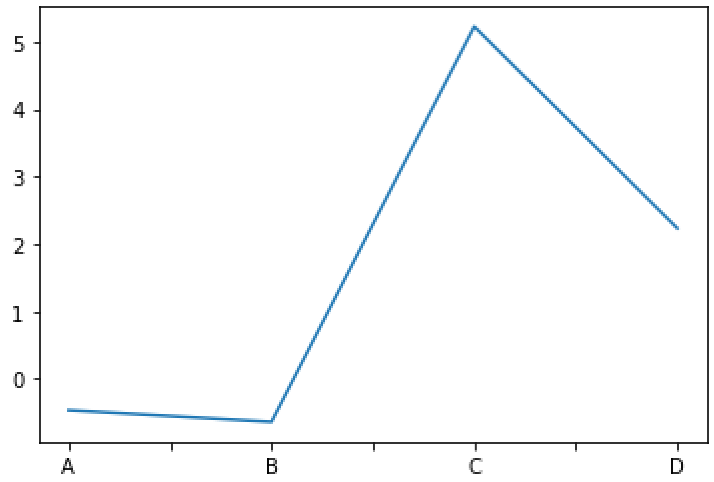
```
<AxesSubplot:>

<Figure size 432x288 with 0 Axes>
```



Basic plotting with matplotlib

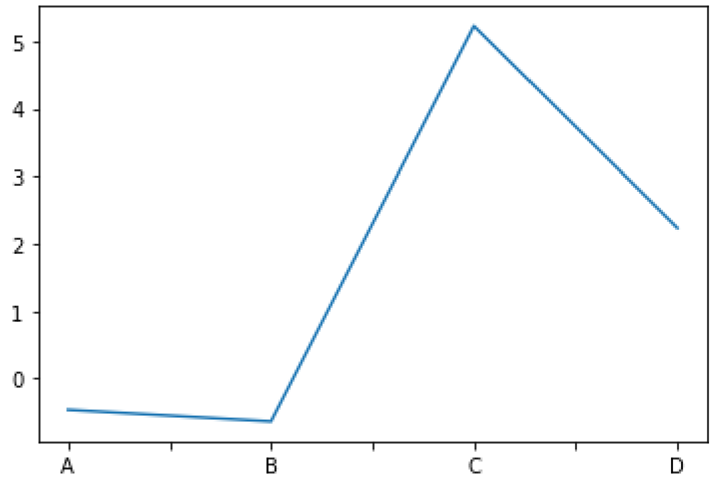
```
df.loc['1/10/2000'].plot()
```

<AxesSubplot:>




```
df.iloc[9].plot()
```

<AxesSubplot:>

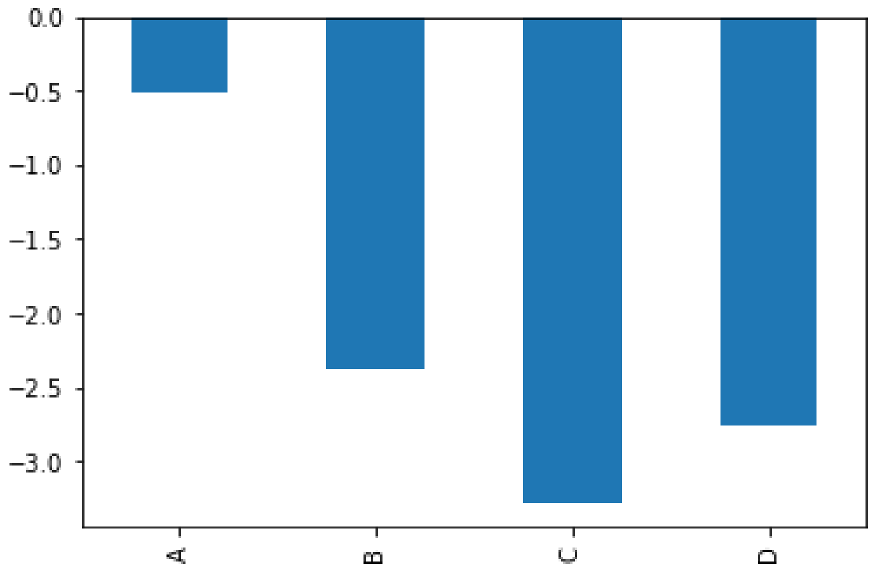


Basic plotting with matplotlib

- *bar* for bar plots
- *hist* for histogram
- *box* for boxplot
- *kde* for density plots
- *area* for area plots
- *scatter* for scatter plots
- …

```
df.iloc[9].plot(kind='bar')
```

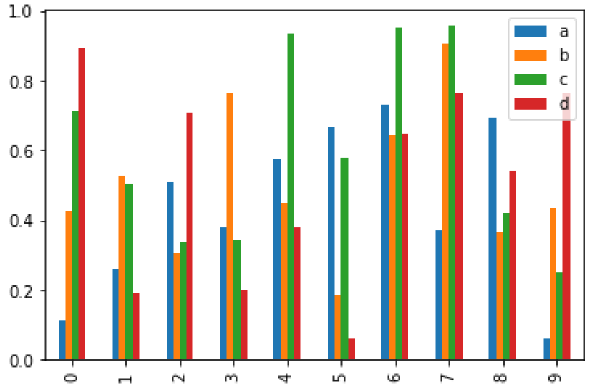
<AxesSubplot:>



Basic plotting with matplotlib

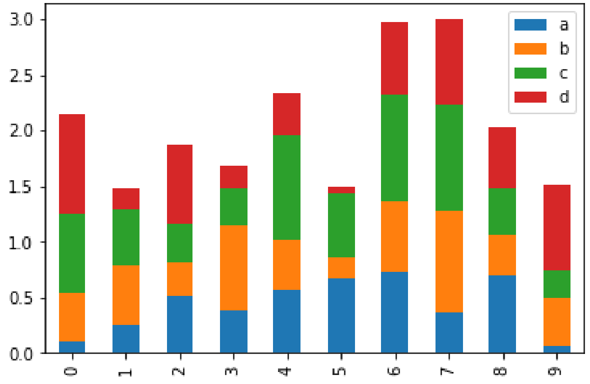
```python
df = pd.DataFrame(np.random.rand(10, 4), columns=["a", "b", "c", "d"])
df.plot(kind='bar')
```

<AxesSubplot:>



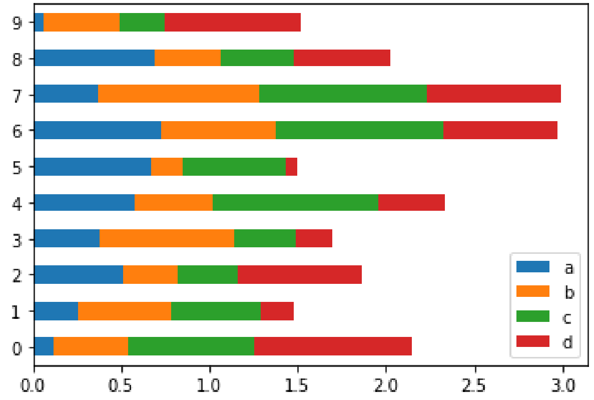

```python
df.plot(kind='bar', stacked=True)
```

<AxesSubplot:>

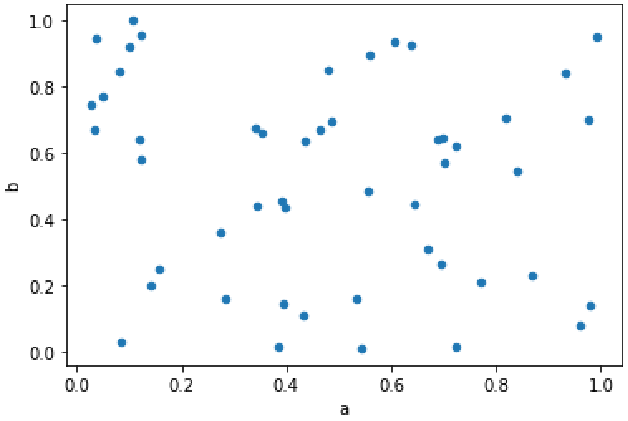



```python
df.plot(kind='barh', stacked=True)
```

<AxesSubplot:>




```python
df = pd.DataFrame(np.random.rand(50, 4), columns=["a", "b", "c", "d"])
df.plot.scatter(x="a", y="b");
```



UIC COMPUTER SCIENCE

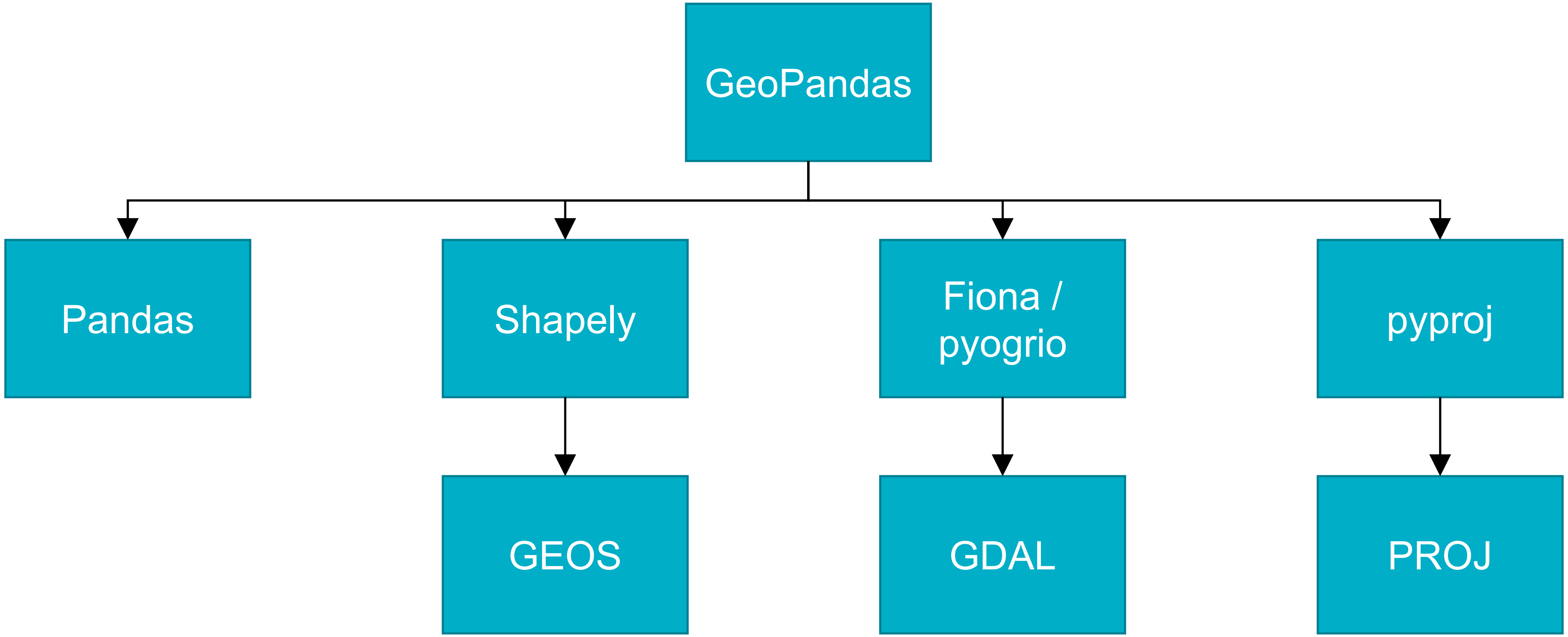
GeoPandas

- Geospatial data & Python made easier.
- Extends Pandas to work with geographic objects and spatial operations.
- Combines the power of several libraries (Pandas, geos, shapely, gdal, pyproj, rtree, …)

GeoPandas

- Read and write several geo formats (Fiona, GDAL).
- Usual DataFrame manipulation.
- Element-wise spatial operations (intersection, union, difference, …)
- Re-project data.
- Visualize geometries.
- Advanced spatial operations: spatial joins and overlays.

GeoPandas

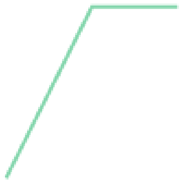

GeoPandas
Pandas
Shapely
Fiona / pyogrio
pypro j
GEOS
GDAL
PROJ

Shapely

- Python package for the manipulation of geometric objects.

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

point = Point(1, 1)
line = LineString([(0, 0), (1, 2), (2, 2)])
poly = line.buffer(1)
```

```
line
```



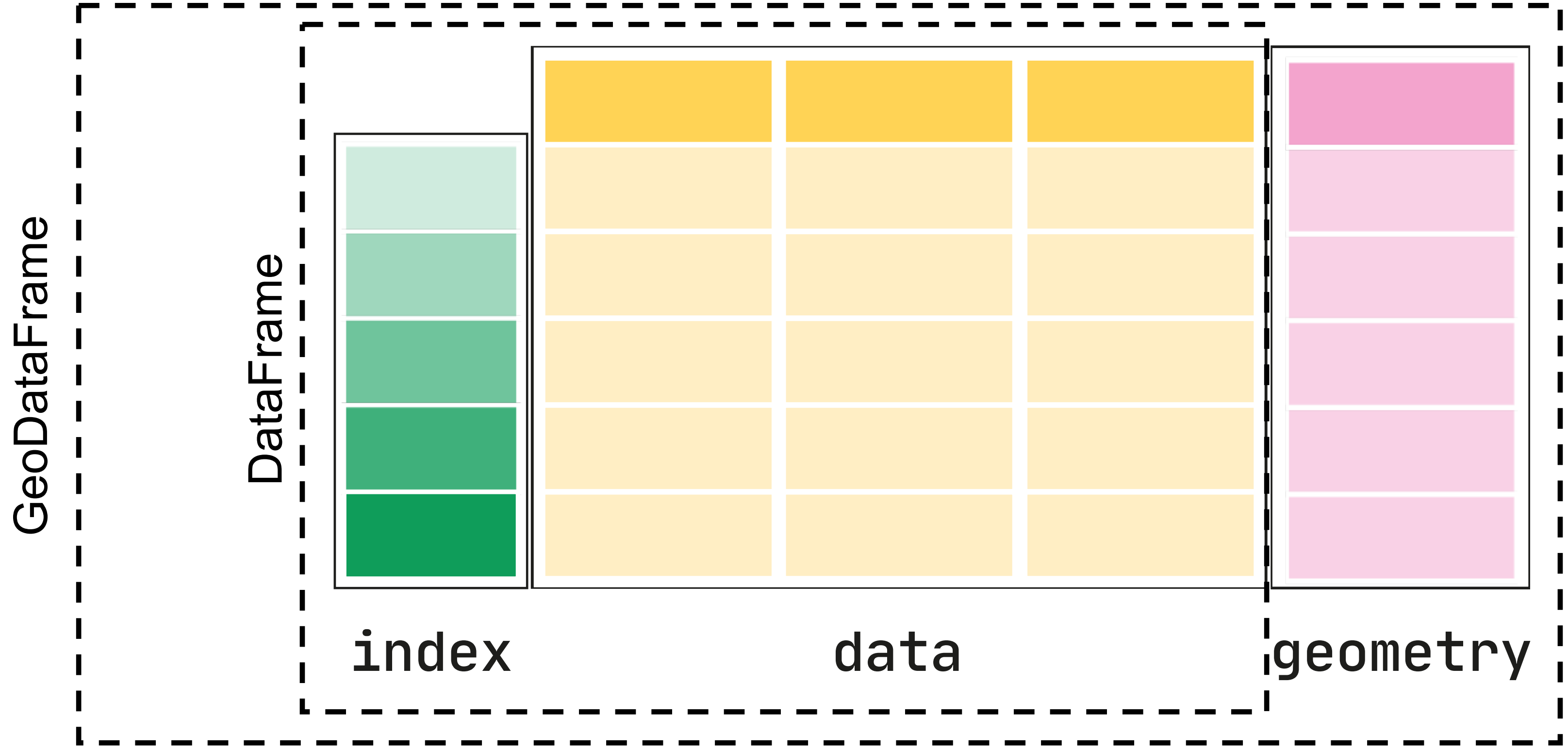
```
poly
```

```
poly.contains(point)
```

```
True
```

GeoPandas

- Core data structure in GeoPandas is the GeoDataFrame (subclass of Pandas' DataFrame).
  - It can store geometry columns and perform spatial operations.

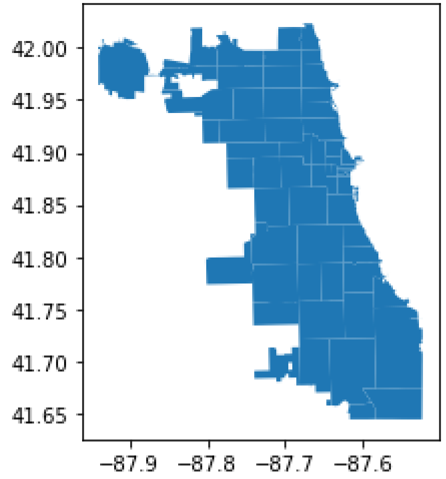


Reading and writing files

```
gdf = gpd.read_file('chicago.geojson')
```

```
gdf.plot()
```

<AxesSubplot:>




```
gdf
```

	objectid	shape_area	shape_len	zip	geometry
0	33	106052287.488	42720.0444058	60647	MULTIPOLYGON (((-87.67762 41.91776, -87.67761 ...
1	34	127476050.762	48103.7827213	60639	MULTIPOLYGON (((-87.72683 41.92265, -87.72693 ...
2	35	45069038.4783	27288.6096123	60707	MULTIPOLYGON (((-87.78500 41.90915, -87.78531 ...
3	36	70853834.3797	42527.9896789	60622	MULTIPOLYGON (((-87.66707 41.88885, -87.66707 ...
4	37	99039621.2518	47970.1401531	60651	MULTIPOLYGON (((-87.70656 41.89555, -87.70672 ...
...	...	...	...	...	...
56	57	155285532.005	53406.9156168	60623	MULTIPOLYGON (((-87.69479 41.83008, -87.69486 ...
57	58	211114779.439	58701.3253749	60629	MULTIPOLYGON (((-87.68306 41.75786, -87.68306 ...
58	59	211696050.967	58466.1602979	60620	MULTIPOLYGON (((-87.62373 41.72167, -87.62388 ...
59	60	125424284.172	52377.8545408	60637	MULTIPOLYGON (((-87.57691 41.79511, -87.57700 ...
60	61	167872012.644	53040.9070778	60619	MULTIPOLYGON (((-87.58592 41.75150, -87.58592 ...

61 rows × 5 columns

Projections

```
gdf
```

	objectid	shape_area	shape_len	zip	geometry
0	33	106052287.488	42720.0444058	60647	MULTIPOLYGON (((-87.67762 41.91776, -87.67761 ...
1	34	127476050.762	48103.7827213	60639	MULTIPOLYGON (((-87.72683 41.92265, -87.72693 ...
2	35	45069038.4783	27288.6096123	60707	MULTIPOLYGON (((-87.78500 41.90915, -87.78531 ...
3	36	70853834.3797	42527.9896789	60622	MULTIPOLYGON (((-87.66707 41.88885, -87.66707 ...
4	37	99039621.2518	47970.1401531	60651	MULTIPOLYGON (((-87.70656 41.89555, -87.70672 ...
...	...	...	...	...	...
56	57	155285532.005	53406.9156168	60623	MULTIPOLYGON (((-87.69479 41.83008, -87.69486 ...
57	58	211114779.439	58701.3253749	60629	MULTIPOLYGON (((-87.68306 41.75786, -87.68306 ...
58	59	211696050.967	58466.1602979	60620	MULTIPOLYGON (((-87.62373 41.72167, -87.62388 ...
59	60	125424284.172	52377.8545408	60637	MULTIPOLYGON (((-87.57691 41.79511, -87.57700 ...
60	61	167872012.644	53040.9070778	60619	MULTIPOLYGON (((-87.58592 41.75150, -87.58592 ...

61 rows × 5 columns

Projecting to EPSG:3395

```
gdf = gdf.to_crs("EPSG:3395")
```

```
gdf
```

	objectid	shape_area	shape_len	zip	geometry
0	33	106052287.488	42720.0444058	60647	MULTIPOLYGON (((-9760228.181 5120114.708, -976...
1	34	127476050.762	48103.7827213	60639	MULTIPOLYGON (((-9765706.326 5120843.341, -976...
2	35	45069038.4783	27288.6096123	60707	MULTIPOLYGON (((-9772181.764 5118831.519, -977...
3	36	70853834.3797	42527.9896789	60622	MULTIPOLYGON (((-9759053.446 5115807.386, -975...
4	37	99039621.2518	47970.1401531	60651	MULTIPOLYGON (((-9763449.188 5116805.817, -976...
...	...	...	...	...	...
56	57	155285532.005	53406.9156168	60623	MULTIPOLYGON (((-9762139.685 5107055.000, -976...
57	58	211114779.439	58701.3253749	60629	MULTIPOLYGON (((-9760833.554 5096312.536, -976...
58	59	211696050.967	58466.1602979	60620	MULTIPOLYGON (((-9754228.915 5090933.835, -975...
59	60	125424284.172	52377.8545408	60637	MULTIPOLYGON (((-9749017.532 5101851.550, -974...
60	61	167872012.644	53040.9070778	60619	MULTIPOLYGON (((-9750019.820 5095367.709, -975...

61 rows × 5 columns

Projections

```
gdf
```

	objectid	shape_area	shape_len	zip	geometry
0	33	106			.67761 ...
1	34	127			.72693 ...
2	35	450			.78531 ...
3	36	708			.66707 ...
4	37	990			.70672 ...
...	...				...
56	57	155			.69486 ...
57	58	21			.68306 ...
58	59	211			.62388 ...
59	60	125424284.172	52377.8545408	60637	MULTIPOLYGON (((-87.57691 41.79511, -87.57700 ...
60	61	167872012.644	53040.9070778	60619	MULTIPOLYGON (((-87.58592 41.75150, -87.58592 ...

61 rows × 5 columns

```
<Geographic 2D CRS: EPSG:4326>
Name: WGS 84
Axis Info [ellipsoidal]:
- Lat[north]: Geodetic latitude (degree)
- Lon[east]: Geodetic longitude (degree)
Area of Use:
- name: World.
- bounds: (-180.0, -90.0, 180.0, 90.0)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Projecting to EPSG:3395

```
gdf = gdf.to_crs("EPSG:3395")
```

```
gdf
```

	objectid	shape	...	geometry
0	33	1060522		976...
1	34	1274760		976...
2	35	4506903		977...
3	36	7085383		975...
4	37	9903962		976...
...	...			...
56	57	1552855		976...
57	58	2111147		976...
58	59	2116960		975...
59	60	1254242		974...
60	61	167872012.644	53040.9070778 60619	MULTIPOLYGON (((-9750019.820 5095367.709, -975...

61 rows × 5 columns

```
<Derived Projected CRS: EPSG:3395>
Name: WGS 84 / World Mercator
Axis Info [cartesian]:
- E[east]: Easting (metre)
- N[north]: Northing (metre)
Area of Use:
- name: World between 80°S and 84°N.
- bounds: (-180.0, -80.0, 180.0, 84.0)
Coordinate Operation:
- name: World Mercator
- method: Mercator (variant A)
Datum: World Geodetic System 1984 ensemble
- Ellipsoid: WGS 84
- Prime Meridian: Greenwich
```

Simple operations

- Accessing & plotting geometry area

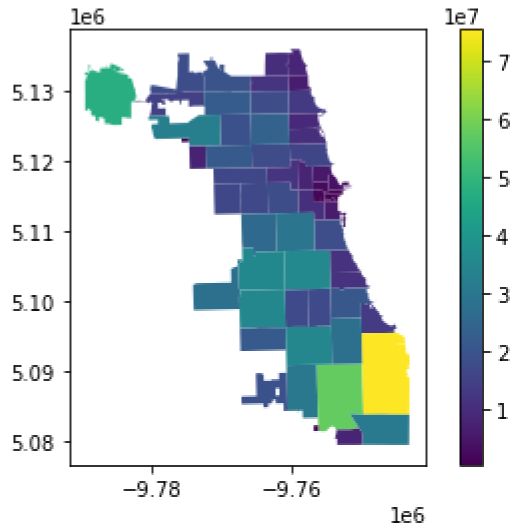
```
gdf.area
```

```
0     1.774279e+07
1     2.132685e+07
2     7.540024e+06
3     1.184732e+07
4     1.656003e+07
          ...
56    2.592093e+07
57    3.515998e+07
58    3.521726e+07
59    2.089192e+07
60    2.793029e+07
Length: 61, dtype: float64
```

```
gdf['area'] = gdf.area
```

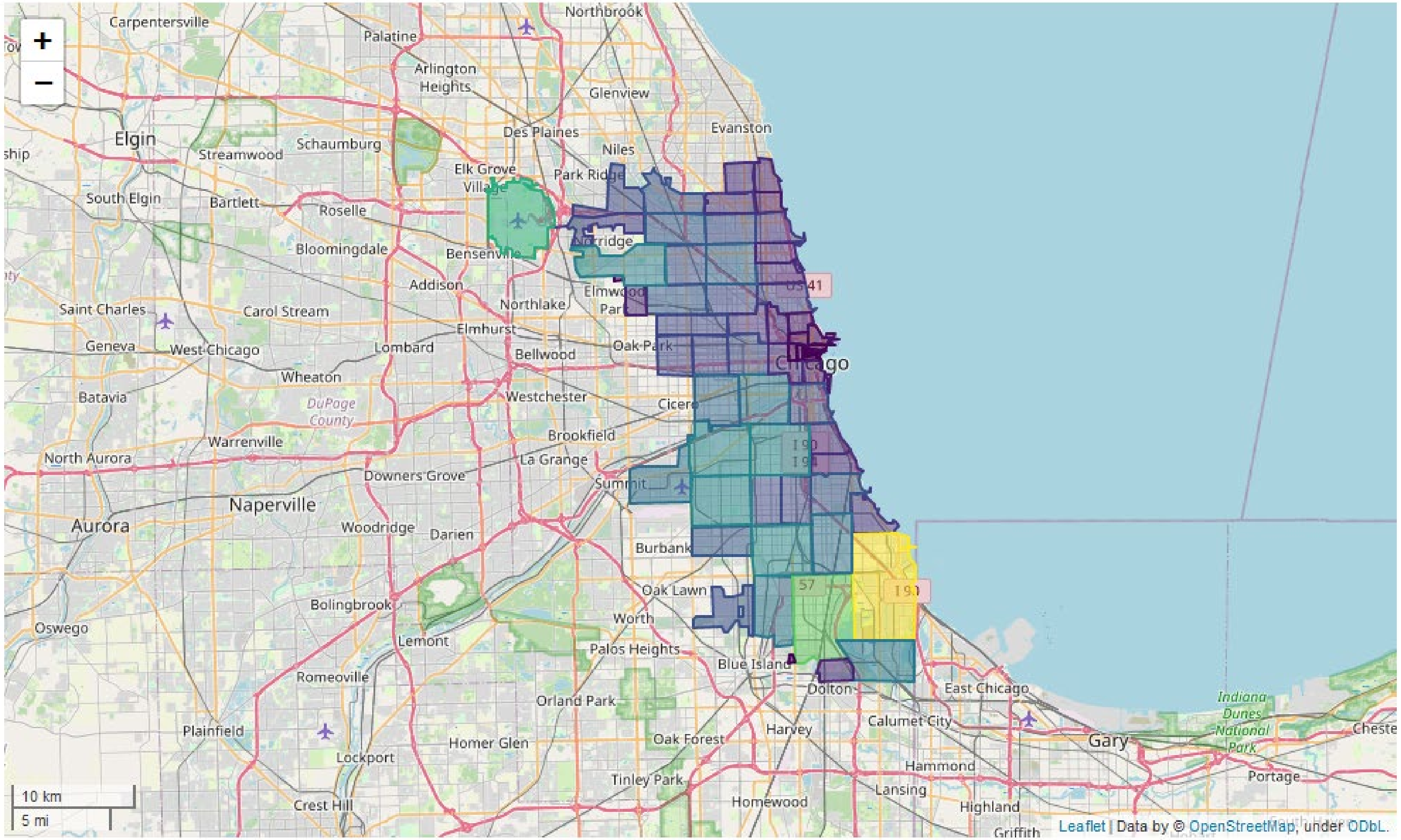
```
gdf.plot("area", legend=True)
```

```
<AxesSubplot:>
```



Simple operations

```
gdf.explore("area", legend=False)
```



UIC COMPUTER SCIENCE

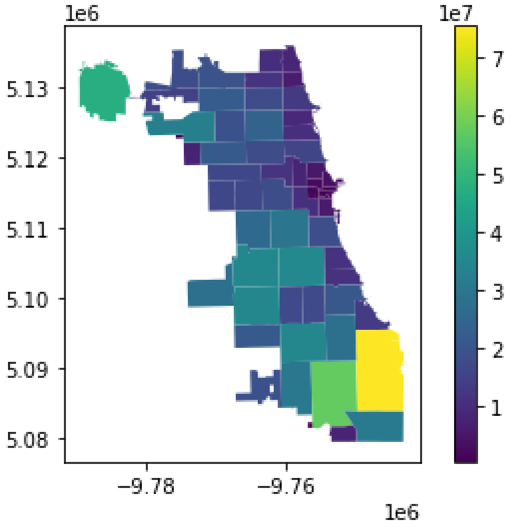
Simple operations

Changing geometry

```
gdf['area'] = gdf.area
```

```
gdf.plot("area", legend=True)
```

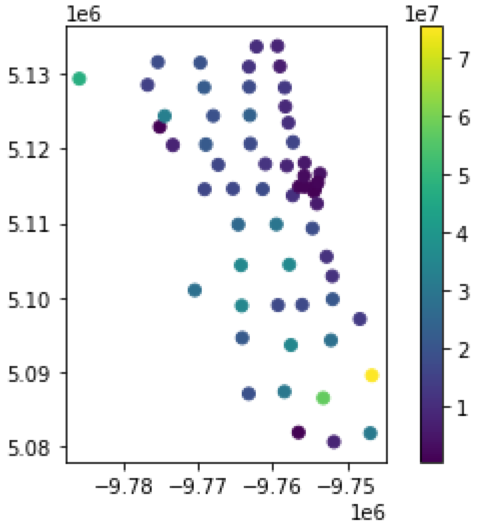
<AxesSubplot:>




```
gdf['centroid'] = gdf.centroid
```

```
gdf = gdf.set_geometry('centroid')
gdf.plot("area", legend=True)
```

<AxesSubplot:>



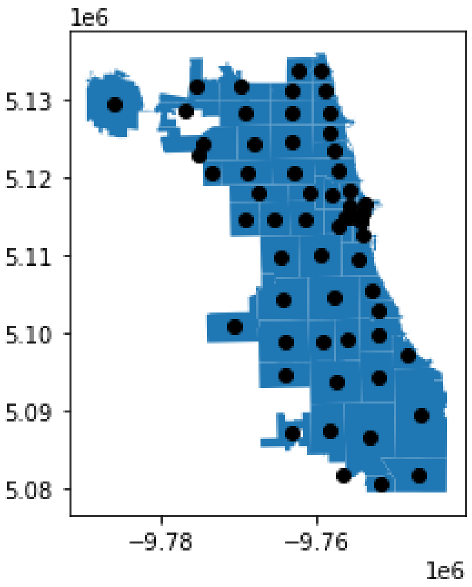
Simple operations

- Plotting centroids and zip areas:

```
gdf = gdf.set_geometry('geometry')
ax = gdf['geometry'].plot()
gdf['centroid'].plot(ax=ax, color="black")
```

Setting geometry back

```
<AxesSubplot:>
```



Geometry operations

```
gdf["convex_hull"] = gdf.convex_hull
```

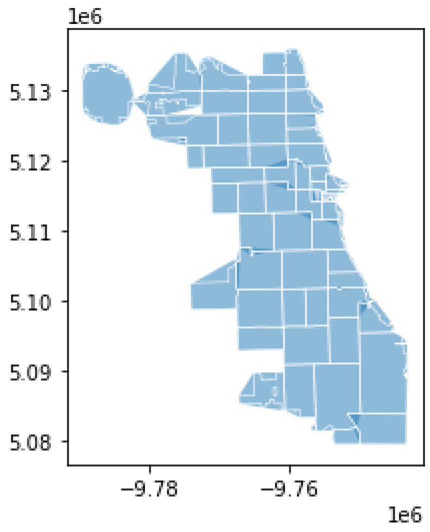
New column with convex hull

```
gdf['boundary'] = gdf.boundary
```

New column with boundary (i.e., outline)

```
ax = gdf["convex_hull"].plot(alpha=.5)
gdf['boundary'].plot(ax=ax, color="white", linewidth=.5)
```

```
<AxesSubplot:>
```



Geometry operations

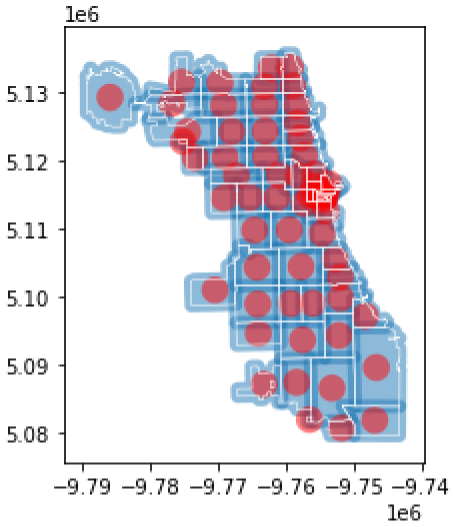
```
gdf["buffered"] = gdf.buffer(1000)
gdf["buffered_centroid"] = gdf["centroid"].buffer(2000)
```

Creating buffer around zip areas

Creating buffer around centroids

```
ax = gdf["buffered"].plot(alpha=.5)
gdf["buffered_centroid"].plot(ax=ax, color="red", alpha=.5)
gdf["boundary"].plot(ax=ax, color="white", linewidth=.5)
```

```
<AxesSubplot:>
```



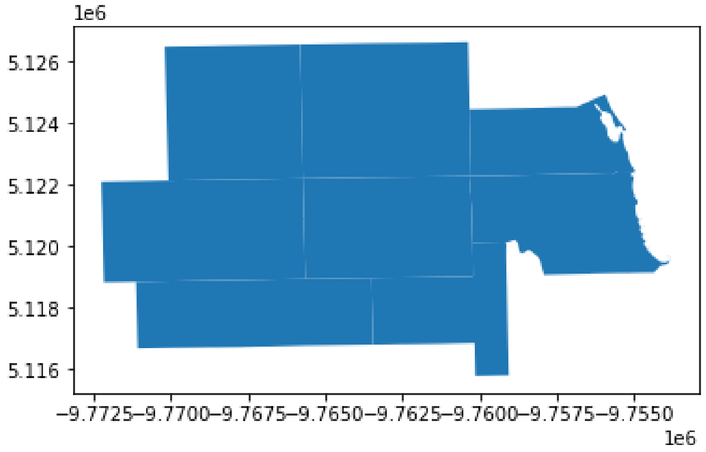
Geometry relations

- GeoPandas offers a series of operations for geometry relations:
    - Crosses, intersects, overlaps, covers, within, touches, …

```
selected = gdf.iloc[0]['geometry']
```

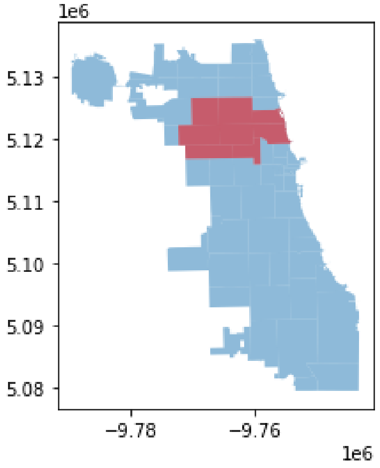
```
gdf[gdf['buffered'].intersects(selected)].plot()
```

```
<AxesSubplot:>
```




```
intersected = gdf[gdf['buffered'].intersects(selected)]
ax = gdf.plot(alpha=.5)
intersected.plot(ax=ax, color="red", alpha=.5)
```

```
<AxesSubplot:>
```



Geometry relations

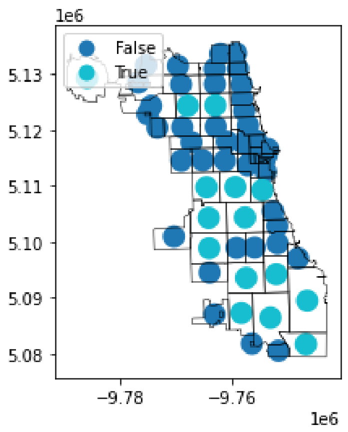
```
gdf["within"] = gdf["buffered_centroid"].within(gdf)
gdf["within"]
```

Whether buffered centroid is within zip area

```
0     False
1     False
2     False
3     False
4     False
      ...
56     True
57     True
58     True
59    False
60     True
Name: within, Length: 61, dtype: bool
```

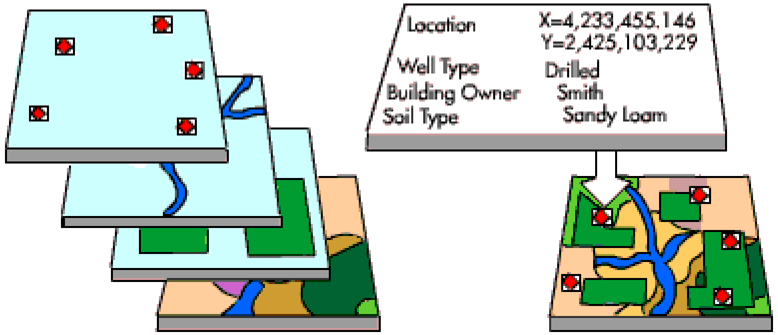
```
gdf = gdf.set_geometry("buffered_centroid")
ax = gdf.plot("within", legend=True, categorical=True, legend_kwds={'loc': "upper left"})
gdf["boundary"].plot(ax=ax, color="black", linewidth=.5)
```

```
<AxesSubplot:>
```



Spatial joins

- A spatial join combines two GeoDataFrames based on the spatial relationship between their geometries.
- Example: spatial join between point layer (e.g., taxi pickups) and a polygon layer (e.g., zip codes).



Spatial join

```python
x_min, y_min, x_max, y_max = gdf.total_bounds

n = 1000

x = np.random.uniform(x_min, x_max, n)
y = np.random.uniform(y_min, y_max, n)

gdf_points = gpd.GeoDataFrame(geometry=gpd.points_from_xy(x, y), crs=gdf.crs)
gdf_points = gdf_points[gdf_points.within(gdf.unary_union)]
```

Bounds of GeoDataFrame

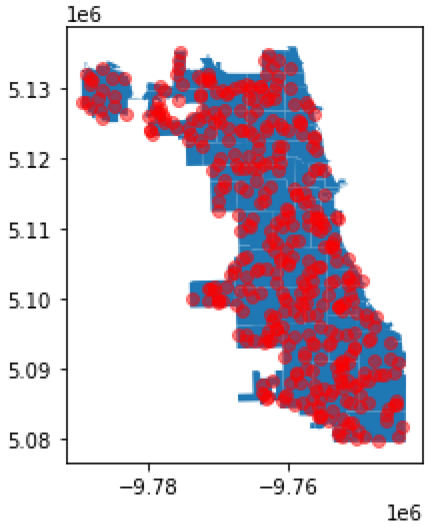
Sample size

Creating GeoDataFrame

Only keeping points within polygons

```python
ax = gdf.plot()
gdf_points.plot(ax=ax, color="red", alpha=.5)
```

```
<AxesSubplot:>
```



Spatial join

- Spatial join: for each point, is it *within* what zip code?

```
gpd.sjoin(gdf_points, gdf, predicate='within')
```

	geometry	index_right	objectid	shape_area	shape_len	zip
1	POINT (-9774274.365 5122584.288)	51	52	194062612.162	77647.3180069	60634
133	POINT (-9774348.810 5124145.541)	51	52	194062612.162	77647.3180069	60634
164	POINT (-9776525.134 5125184.977)	51	52	194062612.162	77647.3180069	60634
207	POINT (-9770521.404 5126432.046)	51	52	194062612.162	77647.3180069	60634
253	POINT (-9777824.534 5126280.441)	51	52	194062612.162	77647.3180069	60634
...	...	...	...	...	...	...
578	POINT (-9761501.347 5118140.579)	3	36	70853834.3797	42527.9896789	60622
917	POINT (-9763366.621 5118966.647)	3	36	70853834.3797	42527.9896789	60622
714	POINT (-9755003.744 5114813.246)	40	26	4847124.8171	14448.1749926	60602
756	POINT (-9772253.752 5121497.783)	2	35	45069038.4783	27288.6096123	60707
894	POINT (-9759652.063 5132688.902)	8	1	49170578.9623	33983.9133065	60626

392 rows × 6 columns

Spatial join

- Grouping by zip code value to obtain number of points within that area.

```python
result = gpd.sjoin(gdf_points, gdf, predicate='within').groupby('zip').count()
```

Grouping by zip code

```python
result = result.filter(['geometry'])
result = result.rename(columns={'geometry': 'count'})
```

```python
result
```

zip	count
60602	1
60605	1
60607	7
60608	12
60609	11
60610	3
60612	4
60613	4
60614	2
60615	8
60616	12
60617	20

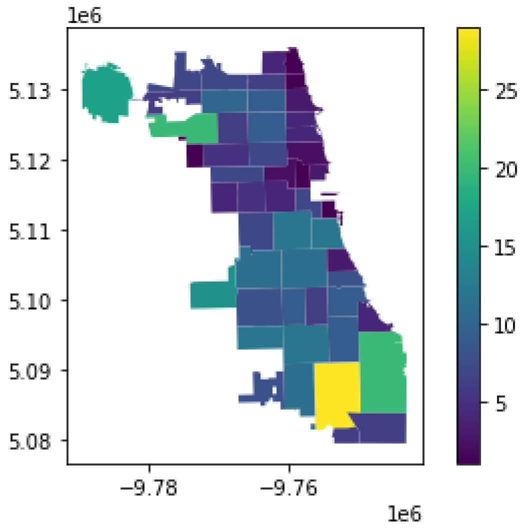
Spatial join

```
merged = pd.merge(result, gdf, right_on='zip', left_index=True)
```

Merging with previous zip code GeoDataFrame

```
merged = merged.set_geometry('geometry')
merged.plot('count', legend=True)
```

<AxesSubplot:>



Spatial join

Aggregating points over spatial regions:

1. Spatial join: map between point and polygon.
2. Group by: aggregate (sum, count, mean, …) by polygon.
3. Merge / join: map between aggregations and polygons.