Working with Spatial Data

Research Computing Summer School 2019

Ian Percel

University of Calgary, Research Computing Services

May 30, 2019

Who is here?

- Who has programmed in Python before?
- Who has used Pandas before?
- Who is familiar with Databases and SQL?
- Who has worked with geospatial data before?
- Who has used PySAL or GeoPandas before?

What is this talk about?

- How do we do spatial analysis without a spatial DataBase like QGIS, PostGRES, or ArcGIS?
- PySAL provides computational geometry at a high level and can be integrated in to Pandas column. Is this enough?
- GeoPandas provides structures that are more useful for geographic information science (rather than having to do geometry manually)
- If we are willing to do some of our own geometric analysis, we can build our own spatial indexes [4]
- What we won't cover: fast raster computations, fast GDAL based operations, spatial statistics

Outline

Outline

Downloading this presentation

`https://westgrid.github.io/calgarySummerSchool2019/4-materials.html`

Right click on the Working with Spatial Data:**Presentation** link and Save As/download to your computer

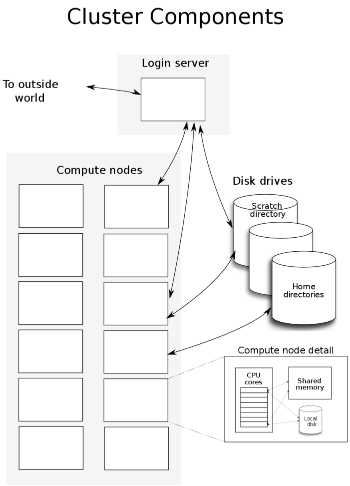
Downloading Data 1: csv data

- We will be working with US Census Data from the 5-year American Community Survey
- Specifically, we will be using the de-identified Public Use Microdata Sample (PUMS) data from 2013
- Point your browser at `https://www2.census.gov/programs-surveys/acs/data/pums/2017/5-Year/` to see the relevant FTP directory
- Download `csv_hil.zip` to your personal computer (by right clicking and choosing Save As)

Downloading Data 2: geographies

- Working with PUMS data requires the PUMA boundaries and we will be relating these back to census tracts
- Point your browser at `https://www2.census.gov/geo/tiger/TIGER2018/PUMA/` to see the relevant FTP directory
- Download `tl_2018_17_puma10.zip` to your personal computer (by right clicking and choosing Save As)
- Point your browser at `https://www2.census.gov/geo/tiger/TIGER2018/TRACT/` to see the relevant FTP directory
- Download `tl_2018_17_tract.zip` to your personal computer (by right clicking and choosing Save As)

Cluster Architecture: where we will be working



Transferring Data to ARC on a Mac

- We will transfer the data set to your account using the `rsync` utility
- On a Mac: open Terminal
- From your Terminal run the following command

```
rsync -avv path/to/file/csv_hil_18.zip userName@arc.ucalgary.ca:"~"
```

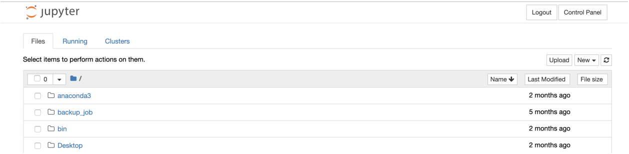
- path/to/file is the full path to the downloaded file
- on a mac desktop this would be ∼/Desktop/
- userName is your itUserName or guestUserName
- you will be prompted for a password, enter your ucalgary email password or the guest password that you have been given.
- If this is your first session signing in, you will be asked to confirm the certificate. Type yes and press enter.
- Once the transfer completes, enter the command: `ssh userName@arc.ucalgary.ca` and enter your password again
- An ASCII Art "ARC" welcome message should appear.
- type `unzip csv_hil_18.zip` and press enter
- Repeat this for the other two files that you downloaded.

Transferring Data to ARC on a Windows PC

- On a Windows PC: open MobaXterm
- To connect an SSH session, the remote host=arc.ucalgary.ca, user name= your IT user Name or guest username
- You will be prompted for a password and will need to enter either your ucalgary email password or the guest password that you have been given
- If this is your first session signing in, you will be asked to confirm the certificate. Type yes and press enter
- An ASCII Art "ARC" welcome message should appear in the terminal.
- When the SSH Session connects an FTP window will appear on the left hand side. This can be used to upload the zip file graphically.
- Once the file has been uploaded, return to the prompt in your Moba terminal, type `unzip csv_hil_18.zip` and press enter.
- Repeat the last two steps for the other two files that you downloaded.

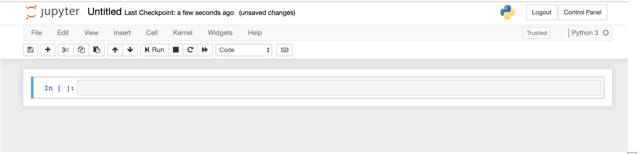
Jupyter Notebooks on ARC

- Why use Notebooks when custom installed environments are cleaner, faster, and more reliable? They're Prettier!
- `https://jupyter.ucalgary.ca:8000/hub/login`
- Use your itusername and email password to login
- Upload any data files that you need to use with the upload button
- Create a new notebook using New > Notebook: Python 3



Jupyter Notebooks on ARC

- Rename notebook by double-clicking on the work Untitled and changing it in the provided field and clicking the rename button at the bottom right of the dialogue
- To run python code, enter it in the text box / cell and press the run button (pressing enter will just create a newline) try out 3+5
- The result will be printed below the cell
- A new cell will be automatically be created below the cell that was just run



Installing Geospatial Libraries

Enter the following text in a cell and run it

```
!pip install --user -U matplotlib
!pip install --user -U geopandas
!pip install --user -U pysal
```

When it finishes, restart the kernel by clicking the circular arrow on the notebook

Importing Geospatial Libraries

Enter the following text in a cell and run it

```
import numpy as np
import matplotlib.pyplot as plt
import pandas as pd
import geopandas as gpd
import pysal as ps
import shapely.geometry
from pyproj import CRS
```

once it is done, you should not have to run any further imports until the end of the class. An unimportant warning about sqlite should appear. If you get an error about a package not being installed then you either did not run the install commands on the previous slide or you need to restart your notebook.

Where we are going

PUMS Data:

```
import pandas as pd
import numpy as np
from pandas import DataFrame,Series

basedf=pd.read_csv('ss13hil.csv')
#what are the columns?
print(list(basedf.columns))
```

```
['insp', 'RT', 'SERIALNO', 'DIVISION', 'PUMA', 'REGION', 'ST', 'ADJHSG', 'ADJINC', 'WGTP',
'NP', 'TYPE', 'ACR', 'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP', 'FS', 'FULP', 'GASP', 'HFL',
'MHP', 'MRGI', 'MRGP', 'MRGT', 'MRGX', 'REFR', 'RMSP', 'RNTM', 'RNTP', 'RWAT', 'RWATPR', 'SINK', 'SMP',
'STOV', 'TEL', 'TEN', 'TOIL', 'VACS', 'VALP', 'VEH', 'WATP', 'YBL', 'FES', 'FINCP', 'FPARC', 'GRNTP',
'HHL', 'HHT', 'HINCP', 'HUGCL', 'HUPAC', 'HUPAOC', 'HUPARC', 'KIT', 'LNGI', 'MULTG', 'MV', 'NOC', ...]
#plus 50 more real columns and 80 replication weights
```

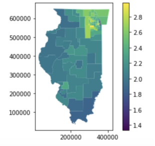
For more information see the pums data dictionary and technical documentation:

https://www2.census.gov/programs-surveys/acs/tech_docs/pums/

Where we are going: geopandas is easy to use for data analysis

What is the mean number of occupants in a census housing unit for each census tract?

```
shp_path='tl_2018_17_puma10.shp'
geo_df=gpd.read_file(shp_path)
housingdf=pd.read_csv('psam_h17.csv', dtype={'PUMA':str})
housingdf['weightedNP']=basedf['WGTP']*basedf['NP']
g = housingdf.groupby(['PUMA'])
pumaAvgNPArray=g['weightedNP'].sum() / g['WGTP'].sum()
avgNPdf=pd.DataFrame(pumaAvgNPArray, columns=['avgNP']).reset_index()
fulldf=geo_df.merge(avgNPdf,how='inner',left_on=['PUMACE10'],right_on=['PUMA'])
fig, ax = plt.subplots(1, 1)
fulldf.plot(column='avgNP', ax=ax, legend=True)
```



What is pandas?

- Pandas provides a SQL-like approach (that blends in elements of statistics and linear algebra) to analyzing tables of data [3]
- DataFrames in R are very similar
- Pandas has been adopted as a de facto standard for input and vectorization across numerous disciplines including Python data analysis with spatial components

Loading data from a csv file

```
basedf=pd.read_csv('ss13hil.csv')
basedf[['SERIALNO', 'PUMA00', 'PUMA10', 'ST', 'ADJHSG', 'ADJINC'
         'WGTP', 'NP', 'TYPE', 'ACR', 'AGS', 'BATH', 'BDSP',
         'BLD', 'BUS', 'CONP', 'ELEP', 'FS', 'FULP']].head()
```

	SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
0	2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
1	2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2	2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
3	2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
4	2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

Loading data from a csv file

```
basedf=pd.read_csv('ss13hil.csv', index_col='SERIALNO',
          usecols=['SERIALNO', 'PUMA00', 'PUMA10', 'ST',
          'ADJHSG', 'ADJINC', 'WGTP', 'NP', 'TYPE', 'ACR',
          'AGS', 'BATH', 'BDSP', 'BLD', 'BUS', 'CONP', 'ELEP',
          'FS', 'FULP'])
basedf.head()
```

	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
SERIALNO																		
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000000075	1000	-9	17	1086032	1085467	6	1	1	1.0	NaN	1.0	3.0	1.0	2.0	0.0	200.0	2.0	2.0
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000132	3510	-9	17	1086032	1085467	60	4	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	1.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0

query

- `query` takes a text string argument in the form (roughly) of a SQL WHERE clause
- Column names need to be referenced without quoting so suitable single-word names are needed
- `https://pandas.pydata.org/pandas-docs/version/0.22/indexing.html#indexing-query`

```
basedf.query('PUMA00==3515')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000061	3515	-9	17	1086032	1085467	36	0	1	NaN	NaN	2.0	2.0	8.0	NaN	0.0	NaN	NaN	NaN
2009000002489	3515	-9	17	1086032	1085467	15	1	1	2.0	1.0	1.0	2.0	2.0	2.0	0.0	50.0	1.0	1.0
2009000002611	3515	-9	17	1086032	1085467	49	2	1	NaN	NaN	1.0	1.0	6.0	NaN	0.0	50.0	2.0	2.0
2009000002724	3515	-9	17	1086032	1085467	45	1	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	40.0	2.0	2.0
2009000006025	3515	-9	17	1086032	1085467	17	2	1	NaN	NaN	1.0	2.0	4.0	NaN	0.0	100.0	2.0	2.0
2009000009853	3515	-9	17	1086032	1085467	14	4	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	150.0	2.0	2.0
2009000010773	3515	-9	17	1086032	1085467	18	2	1	1.0	NaN	1.0	3.0	3.0	2.0	0.0	110.0	2.0	2.0

query

- The query functionality can work between fields.
- However, the only operators that I would rely on are (`==`, `!=`, `<`, `>`, `<=`, `>=`, `&`, `|`)
- `query()` is by default evaluated using the `numexpr` engine, which outperforms pure python on DataFrames of more than 200,000 rows

```
basedf.query('BDSP==NP')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000000108	3402	-9	17	1086032	1085467	15	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	80.0	2.0	2.0
2009000000150	3518	-9	17	1086032	1085467	37	3	1	1.0	NaN	1.0	3.0	2.0	2.0	0.0	200.0	1.0	2.0
2009000000225	3402	-9	17	1086032	1085467	19	2	1	NaN	NaN	1.0	2.0	9.0	NaN	0.0	1.0	2.0	2.0
2009000000256	600	-9	17	1086032	1085467	26	2	1	NaN	NaN	1.0	2.0	7.0	NaN	0.0	90.0	1.0	2.0
2009000000335	2700	-9	17	1086032	1085467	28	1	1	NaN	NaN	1.0	1.0	5.0	NaN	0.0	60.0	2.0	2.0
2009000000461	400	-9	17	1086032	1085467	43	4	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	100.0	1.0	2.0

query

- Arithmetic is possible although I can't speak to its efficiency
- The use of `in` and `not in` operators as well as `==['a','b',...]`, although parts of this will generally be evaluated using pure python

```
basedf.query('0<BDSP<NP & PUMA00==3515')
```

SERIALNO	PUMA00	PUMA10	ST	ADJHSG	ADJINC	WGTP	NP	TYPE	ACR	AGS	BATH	BDSP	BLD	BUS	CONP	ELEP	FS	FULP
2009000002611	3515	-9	17	1086032	1085467	49	2	1	NaN	NaN	1.0	1.0	6.0	NaN	0.0	50.0	2.0	2.0
2009000013780	3515	-9	17	1086032	1085467	20	3	1	NaN	NaN	1.0	2.0	4.0	NaN	0.0	140.0	2.0	2.0
2009000022871	3515	-9	17	1086032	1085467	27	7	1	1.0	NaN	1.0	4.0	2.0	2.0	0.0	70.0	1.0	2.0
2009000024254	3515	-9	17	1086032	1085467	15	4	1	NaN	NaN	1.0	2.0	5.0	NaN	0.0	100.0	1.0	2.0
2009000030494	3515	-9	17	1086032	1085467	39	6	1	NaN	NaN	1.0	2.0	7.0	NaN	0.0	120.0	1.0	2.0
2009000046340	3515	-9	17	1086032	1085467	40	4	1	NaN	NaN	1.0	3.0	5.0	NaN	0.0	80.0	1.0	2.0

concat as JOIN

- If the indexes are overlapping and the column names are not the same *and* `axis=1` is used, that is identical to that of INNER JOIN from SQL

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'c':[7,8,9], 'd':[10,11,12]}, index=['x','y','z'])
pd.concat([df1,df2],axis=1)
```

	a	b	c	d
x	1	4	7	10
y	2	5	8	11
z	3	6	9	12

merge as JOIN

- merge is a holistic JOIN operator
- Like SQL JOINs, the options for using it are complex and take a great deal of practice to master
- We will focus on two options: on= and how=
- on determines the common column used to join the two together (a list of common columns can be specified)
- note that the indexes are not preserved. To keep them .reset_index() before joining and then set the index from that column after or join on index (not covered here)

```
df1=DataFrame({'a':[1,2,3], 'b':[4,5,6]}, index=['x','y','z'])
df2=DataFrame({'a':[1,2,3], 'c':[10,11,12]}, index=['u','v','w'])
pd.merge(df1,df2,on='a')
```

	a	b	c
0	1	4	10
1	2	5	11
2	3	6	12

merge as INNER JOIN

- how can be set to left, right, inner, or outer
- The left_on and right_on options specify the matching columns on the left and right join tables if they have different names
- Note that the default value of how is inner and this will filter out non-matching rows symmetrically

```
df1=DataFrame({'a1':[1,2,3], 'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7], 'c':[10,11,12]})
pd.merge(df1,df2,how='inner',left_on='a1',right_on='a2')
```

	a1	b	a2	c
0	1	4	1	10
1	2	5	2	11

merge as multi-key JOIN

- By passing a list to each of the on options, the corresponding keys are matched sequentially
- In this case two rows are found to match if and only if the value of a1 matches a2 and key1 matches key2

```
df1=DataFrame({'a1':[1,2,3],'key1':['R','R','C'] ,'b':[4,5,6]})
df2=DataFrame({'a2':[1,2,7],'key2':['R','D','C'], 'c':[10,11,12]})
pd.merge(df1,df2,how='outer',left_on=['a1','key1'],right_on=['a2','key2'])
```

	a1	key1	b	a2	key2	c
0	1.0	R	4.0	1.0	R	10.0
1	2.0	R	5.0	NaN	NaN	NaN
2	3.0	C	6.0	NaN	NaN	NaN
3	NaN	NaN	NaN	2.0	D	11.0
4	NaN	NaN	NaN	7.0	C	12.0

`map` for Transforming Columns

- `.map(f)` applies to a Series
- It iterates efficiently over every element of the series and applies the function `f` to that element
- Then it assembles a new Series comprised of the transformed elements in the same order with the same index and returns it
- Weak implicit typing is critical here

```
ser.map(f)
df['col1'].map(f)
```

map for Transforming Columns

```
basedf['NP\_sq']=basedf['NP'].map(lambda x: x**2)
basedf['PUMA\_str']=basedf['PUMA'].map(lambda x: 'PUMA:'+str(x))
```

Split-Apply-Combine as an overall strategy

- Similar to (but more general than) GROUP BY in SQL
- General tool for bulk changes
- The splitting step breaks data into groups using any column (including the row number) [3]
- This can be accomplished using `df.groupby('year')`

year	author
2002	R Barga
1995	K Subieta
2002	D Lomet
1995	F Ferrandina,
1995	T Meyer

year	author
2002	R Barga
2002	D Lomet

year	author
1995	K Subieta
1995	F Ferrandina,
1995	T Meyer

Figure 3: Table Split/Fork

Split-Apply-Combine in more detail

- Groups produced by the split can be individually transformed by an arbitrary function [3]
- This is the essence of Apply (the DataFrame extension of Map)
- The result is combined back into a single DataFrame

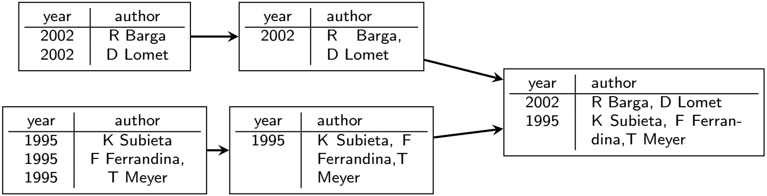


Figure 4: Table Apply + Combine for concatenation

All of this is performed by a single Python interpreter on a single machine.

apply in action

```
subdf=basedf.query('PUMA==03515').copy()
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    return x
subdf=subdf.apply(computeWeightedNP, axis=1)
totals=subdf.sum()
totals['weightedNP']/totals['WGTP']
Out: 1.70737
```

apply as CROSS APPLY

```
def computeWeightedNP(x):
    x['weightedNP']=x['NP']*x['WGTP']
    #print(x)
    totals=x.sum()
    x['avgNP']=totals['weightedNP']/totals['WGTP']
    return x
subdf.groupby(['PUMA']).apply(computeWeightedNP)
```

Problems 1

1. Start from an import of the PUMS csv using the columns 'SERIALNO', 'PUMA', 'BDSP', and 'NP' (with SERIALNO as the index and a DataType of str for the PUMA column using the dtype option). load this to a variable named basedf.
2. Use the command `basedf[['PUMA']].drop_duplicates()` to produce a list of the unique PUMA regions.
3. Using the `query` command, select only those rows where the PUMA region number matches one specific one that you chose to work with from the previous step. How many records are returned? (look below the readout of sample rows to see a number) Repeat this for 4 different PUMAs and compare the counts returned for each. (what is the total?) Can you rewrite this as a single query using the OR operator? make sure the resulting counts agree.

Problems 2

1. Select off two subsets of the data using queries that on PUMAs. The first should include 01300, 00800, and 00105. The second should include 00105, 00300, 01104.
2. Use merge to perform and inner join between the two tables on SERIALNO (HINT use reset_index() on each first to make the old index a column in each table)
3. The resulting table will only have rows that appeared in both. Which PUMAs belong to the overlap?

Problems 3

1. Write a map operation that scales NP as an exponent using the numpy function `np.exp`
2. Write a variation on the CROSS APPLY above without using apply

Solutions 3

```
import pandas as pd

housingdf=pd.read_csv('psam_h17.csv', dtype={'PUMA':str})
housingdf['weightedNP']=basedf['WGTP']*basedf['NP']
g = housingdf.groupby(['PUMA'])
pumaAvgNPArray=g['weightedNP'].sum() / g['WGTP'].sum()
avgNPdf=pd.DataFrame(pumaAvgNPArray, columns=['avgNP']).reset_index()
```

What's in a GeoDataBase?

- A GeoDB has three essential components: [1]
- Spatial features (with a Datum and Projection information)
- Attributes linked to spatial features
- A means of transforming and linking by attribute data or spatial feature
- Pandas gives us a way of managing structured attribute data, what do we need to add in order to build a usable spatial analysis data structure

PySAL for computational geometry

- PySAL provides a high level interface for shape objects and their transformation: `pysal.lib.cg`
- By using PySAL objects, we can read in and handle shape files (one of the typical formats for spatial data)
- We will examine some of the functionality that PySAL grants us in the context of an object column in a standard Pandas DataFrame
- note that PySAL is also an interface to a wide range of spatial statistical models and spatial econometric models that are not accessible from other python libraries [2]

Loading our Pandas Data and our Shape File

- We can begin by loading a standard DataFrame from the csv data that we are interested in
- Note how we have handled the PUMA column
- Then we can load the shape file separately using the `ps.lib.io.fileio.FileIO()` function
- `read()` is a more fundamental IO method than the `read_csv()` method and the file handle needs to be closed

```
import pandas as pd
import pysal as ps

housingdf=pd.read_csv('psam_h17.csv', dtype={'PUMA':str})
shp_path='tl_2018_17_puma10.shp'
f=ps.lib.io.fileio.FileIO(shp_path)
all_polygons=f.read()
f.close()
```

Examining our spatial features

- The result of reading the shape file is a list of polygons

```
type(all_polygons)
Out: list
type(all_polygons[0])
Out: pysal.lib.cg.shapes.Polygon
all_polygons[0]
Out: <pysal.lib.cg.shapes.Polygon object at 0x2b33d0388da0>
all_polygons[0].vertices
Out: [(-87.721436, 41.734862), (-87.721417, 41.734863), ...,
      (-87.721436, 41.734862)]
```

What do you think the coordinates in this file are?

Examining our spatial features

- PySAL Shapes include a substantial number of precomputed attributes and efficient methods

```
all_polygons[0].centroid
Out: (-87.73227916862234, 41.68628900126081)
all_polygons[0].perimeter
Out: 0.7424176724174986
all_polygons[0].area
Out: 0.010349874158494149
ps.lib.cg.get_shared_segments(all_polygons[0], all_polygons[14])
Out: [<pysal.lib.cg.shapes.LineSegment at 0x2b33d12196a0>,
 <pysal.lib.cg.shapes.LineSegment at 0x2b33d12196d8>,
 <pysal.lib.cg.shapes.LineSegment at 0x2b33d1219710>,...]
```

The computed segments are the shared boundary of the two polygons.

Where is the data associated with the shapes?

- In order to capture the attributes associated with the shapes themselves, we will need to read the associated .dbf file
- There are two reasons to do this:
- First, the regions probably have some interesting additional data from the census.
- Second, we don't have labels for these shapes so we can't link them to the housingdf!

```
dfpoly=pd.DataFrame(all_polygons,columns=['polygon'])
dbf_path='tl_2018_17_puma10.dbf'
f2=ps.lib.io.fileio.FileIO(dbf_path)
dbheader=f2.header
dbfile=f2.read()
f2.close()
pd.DataFrame(dbfile, columns=dbheader).head()
```

	STATEFP10	PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10
0	17	03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725
1	17	03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232

Producing a complete data set

- The coordinates of the centroid of the first elements seem to agree
- Let's try and combine them by `pd.concat` and then join to the csv file

```
geo_df1=pd.concat([pd.DataFrame(dbfile, columns=dbheader),dfpoly],axis=1)
geo_df1.head()
```

	STATEFP10	PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	polygon
0	17	03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725	<pysal.lib.cg.shapes.Polygon object at 0x2b33d...
1	17	03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232	<pysal.lib.cg.shapes.Polygon object at 0x2b33d...
2	17	03700	1703700	Kendall & Grundy Counties PUMA	G6120	S	1912412909	37262592	+41.4200983	-088.4339373	<pysal.lib.cg.shapes.Polygon object at 0x2b33d...
3	17	02000	1702000	McLean County PUMA	G6120	S	3064559693	7853695	+40.4945594	-088.8445391	<pysal.lib.cg.shapes.Polygon object at 0x2b33d...

```
pd.merge(geo_df1, housingdf,how='inner',left_on=['PUMACE10'],right_on=['PUMA'])
```

Is this enough to get things done?

- This is a useful structure (especially for doing complex spatial statistics)
- We can automate complex problems for filtering and producing attributes derived from spatial features
- We have two problems that remain unsolved by this: projection and fast spatial indexing

```
geo_df1['computed_area']=geo_df1['polygon'].map(lambda x: x.area)
geo_df1['total_listed_area']=geo_df1.ALAND10+geo_df1.AWATER10
```

Problems

1. Following the slides, assemble a dataframe from the csv, dbf, and shp files name it geo_df1
2. Use the code from the "Is this enough..." slide to compute the area from the polygons and the area listed in the dbf
3. Calculate the ratio of these. Is it possible that this is simply a change of units of measure? No. These are unprojected shape files.
4. Use the polygon boundary intersection function `ps.lib.cg.get_shared_segments(poly1,poly2)` and the `.map` function with a boolean index to find all of the records that have shared boundaries with the first polygon (i.e. adjacent regions)

Solution 4

```
poly1=geo_df1['polygon'][0]
f=lambda poly2: ps.lib.cg.get_shared_segments(poly1,poly2)
geo_df1['sharedSegments']=geo_df1['polygon'].map(f)
def listFilter(x):
    if x==[]:
        return False
    else:
        return True

geo_df[geo_df['sharedSegments'].map(listFilter)]
```

GeoDataBases made considerably easier

- GeoPandas supports almost all Pandas operations in one form or another
- GeoPandas provides easy projection handling
- GeoPandas provides R-Tree indexing of GeoDataFrames to accelerate spatial filtering and joining

What is a GeoDataFrame?

- A GeoDataFrame is mostly structured like a DataFrame but has a single column that is a GeoSeries
- This links each attribute record to a unique geospatial feature
- The GeoSeries column can have any name but by default it is `geometry`
- The objects in the geometry column are Shapely objects (in our case Polygons)
- The GeoSeries and GeoDataFrame have a single common `crs` attribute for characterizing the Coordinate Reference System and projection data
- The GeoSeries and GeoDataFrame have a common spatial index attribute `sindex` that implements an R-Tree for the GeoSeries

Loading data to a GeoDataFrame

- Is vastly easier than manually assembling linked spatial data for a Pandas DataFrame
- Automatically identifies the corresponding .prj and .dbf files and incorporates them using `fiona`
- Can still be done manually if something special is needed (http://geopandas.org/gallery/create_geopandas_from_pandas.html#sphx-glr-gallery-create-geopandas-from-pandas-py)

```
import geopandas as gpd

shp_path='tl_2018_17_puma10.shp'
geo_df=gpd.read_file(shp_path)
```

Examining our GeoDataFrame

- The GeoDataFrame has (in one very short step) all of the information that we manually built into our PySAL supported DF from the .shp and .dbf files

```
geo_df.head()
```

	STATEFP10	PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry
0	17	03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725	POLYGON ((-87.721436 41.734862, -87.721417 41....
1	17	03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232	POLYGON ((-87.860787 41.557522, -87.857298 41....
2	17	03700	1703700	Kendall & Grundy Counties PUMA	G6120	S	1912412909	37262592	+41.4200983	-088.4339373	POLYGON ((-88.26809799999999 41.724544, -88.26...
3	17	02000	1702000	McLean County PUMA	G6120	S	3064559693	7853695	+40.4945594	-088.8445391	POLYGON ((-88.92933099999999 40.75333699999999...
4	17	01602	1701602	Menard, Logan, De Witt, Piatt, Moultrie, Shelb...	G6120	S	9254202416	88234698	+39.7699432	-089.2258108	POLYGON ((-88.494249 39.215001, -88.4992949999...

```
geo_df.geometry.head()
```

```
0    POLYGON ((-87.721436 41.734862, -87.721417 41....
1    POLYGON ((-87.860787 41.557522, -87.857298 41....
2    POLYGON ((-88.26809799999999 41.724544, -88.26...
3    POLYGON ((-88.92933099999999 40.75333699999999...
4    POLYGON ((-88.494249 39.215001, -88.4992949999...
Name: geometry, dtype: object
```

What is different about the geometry column

- It is a Shapely object not a PySAL cg.Shape object
- It has a Coordinate Reference System (crs) imported from the linked .prj file

```
type(geo_df.geometry[0])
Out: shapely.geometry.polygon.Polygon
geo_df.crs
Out: {'init': 'epsg:4269'}
```

This raises a question about what Shapely objects can do differently from PySAL `cg.Shape` and how we can analyze the projection using the crs attribute

CRS data

- First we need to get a handle on what the crs value means and if it agrees with the .prj file provided

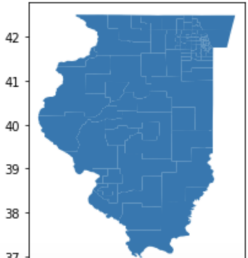
```
from pyproj import CRS

wkt_str='GEOGCS["GCS_North_American_1983",DATUM["D_North_America
crs_utm = CRS.from_string(wkt_str)
crs_utm.to_proj4()
Out: +proj=longlat +datum=NAD83 +no_defs +type=crs
crs_utm.to_epsg()
Out: 4269
```

This establishes that the WKT string from the .prj file has been correctly loaded to the crs. We can learn more about the projection in use by looking it up on `https://spatialreference.org/ref/epsg/nad83/` However, we can already tell by examining the proj4 string that the data is unprojected because proj=longlat

What is the difference?

- Unprojected data will have the same topological properties but different distances and directions
- We have demonstrated above the the original data comes out with the wrong areas relative to the dbf file
- How bad does it really look? The ratio of length to width goes from 1.4 to 1.75

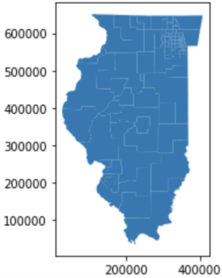
```
geo_df.plot()
```



Changes of Projection

- Here we chose a semi-arbitrary projection that works on much of North America but it tailored to the eastern part of Illinois
- Generally care is required in choosing your projection, but the most important thing is consistency
- Differently projected data is fundamentally not comparable

```
geo_df=geo_df.to_crs({'init': 'epsg:26971'})
geo_df.plot()
```



Shapely Polygons

- More fundamental then PySAL polygons
- Full set theoretic machinery: Intersections, Unions, Contains, Differencing
- More geometrically technical options in general but very efficient
- Easy to convert back and forth with PySAL

```
import shapely.geometry
poly1=geo_df.geometry[0]
type(poly1)
Out: shapely.geometry.polygon.Polygon
poly2=ps.lib.cg.asShape(poly1)
type(poly2)
Out: pysal.lib.cg.shapes.Polygon
poly3=shapely.geometry.polygon.Polygon(shapely.geometry.asShape(poly2))
type(poly3)
Out: shapely.geometry.polygon.Polygon
poly1==poly3
Out: True
```

Problems 1

1. Perform the projection of the `geo_df` GeoDataFrame that was outlined in the slides
2. Using the shapely area function through GeoPandas (i.e. `geo_df.geometry.area`) redo the area computation exercise from the last section
3. What is the percent difference in the projected area of each PUMA from the stated land+water areas? What is the maximum observed difference?
4. Use query to find the record with the maximum area difference use the `.plot()` function to plot the PUMA with the biggest error.
5. Use query to plot the PUMAs with a percent error greater than 0.1, greater than 0.08, and greater than 0.05
6. What part of Illinois is it that has the lowest accuracy of projection?

Problems 2

1. Redo the neighbouring PUMA identification problem but with a GeoDataFrame (Hint: you will need to convert the Shapely polygons to PySAL polygons to use the same method)

Solution 1

```
geo_df['statedArea']=geo_df.ALAND10+geo_df.AWATER10
geo_df['computedArea']=geo_df.geometry.area
geo_df['areaDiff']=geo_df['statedArea']-geo_df['computedArea']
geo_df['abs_areaDiff']=geo_df['areaDiff'].abs()
geo_df['frac_areaDiff']=geo_df['abs_areaDiff']/geo_df['statedArea']
geo_df['perc_areaDiff']=geo_df['frac_areaDiff']*100
geo_df['perc_areaDiff'].max()
Out:0.10317568566038449
geo_df.query('perc_areaDiff>0.1')
geo_df.query('perc_areaDiff>0.1').plot()
geo_df.query('perc_areaDiff>0.08').plot()
geo_df.query('perc_areaDiff>0.05').plot()
```

Western Illinois is the worst part of the projection

Solution 2

```
poly1=ps.lib.cg.asShape(geo_df.geometry[0])
f=lambda poly2: ps.lib.cg.get_shared_segments(poly1,ps.lib.cg.asShape(poly2))
geo_df['sharedSegments']=geo_df.geometry.map(f)

def listFilter(x):
    if x==[]:
        return False
    else:
        return True

geo_df[geo_df['sharedSegments'].map(listFilter)]
```

Extending a GeoDataFrame

- Joining the PySAL DataFrame to the csv data was obvious with `pd.merge`
- However, merge with the spatial frame in the right position returns a DataFrame which would mean giving up our spatial indexing, CRS, and plotting!
- GeoPandas has its own implementations of many standard pandas analysis functions that accept the same options
- Let's start from the reduced DataFrame that was computed earlier using the PUMS weights

```
housingdf['weightedNP']=housingdf['WGTP']*housingdf['NP']
g = housingdf.groupby(['PUMA'])
pumaAvgNPArray=g['weightedNP'].sum() / g['WGTP'].sum()
avgNPdf=pd.DataFrame(pumaAvgNPArray, columns=['avgNP']).reset_index()
```

Extending a GeoDataFrame

- AvgNPdf has the same number of records as our geo_df table because we have made use of groupby
- Join is 1-1 and can be done as an INNER JOIN
- The merge function returns a GeoDataFrame

```
fulldf=geo_df.merge(avgNPdf,how='inner',left_on=['PUMACE10'],right_on=['PUMA'])
type(fulldf)
Out: geopandas.geodataframe.GeoDataFrame
fulldf.head()
```

PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry	PUMA	avgNP
03411	1703411	Cook County (South Central)--Worth & Calumet T...	G6120	S	94129155	1576803	+41.7313634	-087.7167725	POLYGON ((350904.9028309903 562835.2311880648,...	03411	2.278784
03107	1703107	Will County (Northeast)--Frankfort, Homer & Ne...	G6120	S	243596923	478565	+41.5528956	-087.9168232	POLYGON ((339419.9439735664 543066.0907343754,...	03107	2.698186
03700	1703700	Kendall & Grundy Counties PUMA	G6120	S	1912412909	37262592	+41.4200983	-088.4339373	POLYGON ((305427.9081081446 561510.3646416094,...	03700	2.696594
02000	1702000	McLean County PUMA	G6120	S	3064559693	7853695	+40.4945594	-088.8445391	POLYGON ((249670.2754834539 453821.0664099103,...	02000	2.203863

GeoDataFrame Queries

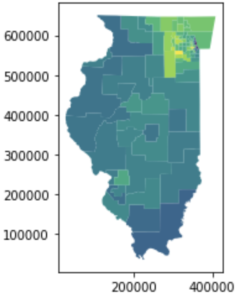
- Joined data can subsequently be filtered as usual

```
fulldf.query('avgNP>2.9')
```

PUMACE10	GEOID10	NAMELSAD10	MTFCC10	FUNCSTAT10	ALAND10	AWATER10	INTPTLAT10	INTPTLON10	geometry	PUMA	avgNP
03106	1703106	Will County (Northwest)--DuPage & Wheatland To...	G6120	S	185174720	2902950	+41.6828015	-088.1450808	(POLYGON ((321438.3752390264 561908.3529506001...	03106	2.965908
03527	1703527	Chicago City (Southwest)--Gage Park, Garfield ...	G6120	S	34025246	48984	+41.7862967	-087.7415184	POLYGON ((351295.6188415109 570459.8131469378,...	03527	2.978911

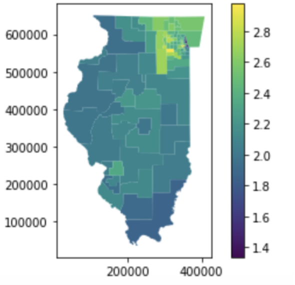
Choropleth Plotting

```
fulldf.plot(column='avgNP')
```



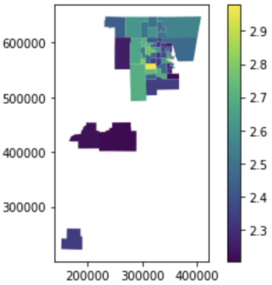
Choropleth Plotting

```
fig, ax = plt.subplots(1, 1)
fulldf.plot(column='avgNP', ax=ax, legend=True)
```



Filtered Choropleth Plotting

```
fig, ax = plt.subplots(1, 1)
fulldf.query('avgNP>2.2').plot(column='avgNP', ax=ax, legend=True)
```



Spatial Index Based Filtering

- Sometimes, we want to analyze explicit spatial subsets
- We could define a mask and test for inclusion row by row
- It is much easier to use the spatial index that already exists

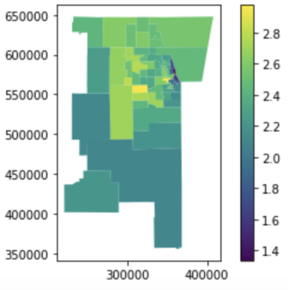
```
fig, ax = plt.subplots(1, 1)
fulldf.cx[250000:,450000:]
```

returns only records with some portion of the polygon east of 250000 and north of 450000 (in the projected coordinate system)

Spatial Index Based Filtering

- Plotting works the same way and allows us to focus our attention on areas of interest

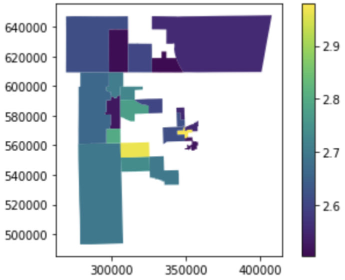
```
fig, ax = plt.subplots(1, 1)
fulldf.cx[250000:,450000:].plot(column='avgNP', ax=ax, legend=True)
```



Spatial Index Based Filtering

- The result can be combined with relational / numerical filtering of values to find records of interest

```
fig, ax = plt.subplots(1, 1)
filteredData=fulldf.cx[250000:,450000:].query('avgNP>2.5')
filteredData.plot(column='avgNP', ax=ax, legend=True)
```



Problems 1

1. Using the housingdf imported earlier, modify the code from the "Extending a GeoDataFrame" slides to compute a new weighted average data set for the column BDSP (bedrooms per housing unit sampled)
2. Continue the analysis using the process outlined in this section. First create a GeoDataFrame that includes both the PUMA shape data and the PUMS csv data for housing in Illinois
3. Create a choropleth map of the data with a legend to better understand the distribution of values and how hot spots cluster spatially in Illinois

Problems 2

1. Explore plotting subsets that you query with an expression similar to the one used above to examine large average NP values, but do this for your average BDSP data
2. Use spatial indexing to zoom in on a relevant geographic region and plot the result
3. Finally, apply a more stringent query filter to see the subset of PUMAs in the high concentration region that have the most dramatic numbers for average bedrooms per household

Spatial Joins for linking geographies

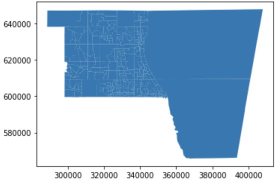
- It is normal to deal with multiple spatial feature sets in geospatial analysis
- Often, different data is attached to each feature and in order to link data across scales or express connective relationships it is necessary to perform spatial joins
- Spatial Joins can be thought of as a way of forming a join between two tables of discrete features while using complex spatial relationships as the join criterion rather than using matching keys
- To understand this we will need a second data set that we can join to the first

```
shp_path_t='tl_2018_17_tract.shp'
dft=gpd.read_file(shp_path_t)
dft=dft.to_crs({'init': 'epsg:26971'})
```

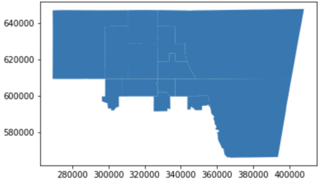
Examining our two geographies

- If two feature sets were the same, comparing them would be uninteresting (or at least very easy)
- It is important to make sure that both are using the same projection

```
dft.cx[300000:,600000:].plot()
```




```
fulldf.cx[300000:,600000:].plot()
```



Adding some simple data to the tract level geography

- In the name of expedience, we will append some randomly generated data to our tract GeoDataFrame

```
import numpy as np
dft['tract_score']=np.random.normal(1000,150,dft.shape[0])
dft.head()
```

TRACTCE	GEOID	NAME	NAMELSAD	MTFCC	FUNCSTAT	ALAND	AWATER	INTPTLAT	INTPTLON	geometry	tract_score
011700	17091011700	117	Census Tract 117	G5020	S	2370100	102060	+41.1294653	-087.8735796	POLYGON ((337416.9310474549 496233.9953135318,...	1253.039315
011800	17091011800	118	Census Tract 118	G5020	S	1790218	55670	+41.1403452	-087.8760059	POLYGON ((336873.9649618285 497112.4753127118,...	892.692410
400951	17119400951	4009.51	Census Tract 4009.51	G5020	S	5170038	169066	+38.7277628	-090.1002620	POLYGON ((145283.7247863071 227488.7024631576,...	724.447989
400952	17119400952	4009.52	Census Tract 4009.52	G5020	S	5751222	305905	+38.7301928	-090.0827510	POLYGON ((146843.1383274837 229402.0122085095,...	1035.095988
950300	17189950300	9503	Census Tract 9503	G5020	S	30383680	349187	+38.3567671	-089.3783135	POLYGON ((205526.7301528867 182696.9930974582,...	1107.998299

R-Tree Dependency

- In order for `sjoin` to work, GeoPandas requires an additional library: `rtree`
- R-Tree is a wrapper for a c-type `libspatialindex`
- To go further with this example, we will need to compile that and link it to our python distribution
- All of the options for accessing this take more time than we have so the remainder of this talk will be a demonstration
- I will post instructions on how to follow up on this on your own on ARC after the talk.

Basic Join Syntax

- `gpd.sjoin(df1,df2,how=, op=)`
- `how` is analogous to how for relational joins except that it also specified which geometry column is retained
- Options: left (df1 geometry is kept and all records from df1), right (df2 geometry is kept and all records from df2), inner (df1 geometry is kept but only matching records from df1)
- `on` is implicit since there is only one GeoSeries per GeoDataFrame
- op determines the spatial rule for matching (explanation below for the left and inner cases)
- Options: intersects (any overlap), contains (df1 object entirely surrounds df2 object), within (df1 object is entirely surrounded by df2 object)

Basic Join Syntax

```
joined_data=gpd.sjoin(tractdf,pumsdf,how='left',op='intersects')
```

Produces a spatially joined GeoDataFrame where the geometry column retained is the tract level geometry and any records associated with PUMAs that it intersects would be appended.

Join Example: Multiple Matches

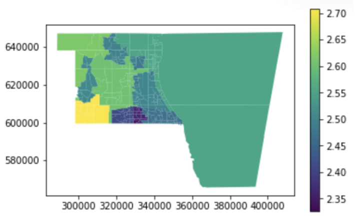
```
joined_data.shape[0]
Out:493
tractdf.shape[0]
Out:287
joined_data.query('TRACTCE=="803500"')
```

	TRACTCE	NAMELSAD	tract_scor	avgNP
1	803500	Census Tract 8035	1158.954562	2.412800
2	803500	Census Tract 8035	1158.954562	2.327185

This result (the joined_data table) has the same number of tracts included as the original filtered tract data, 286.

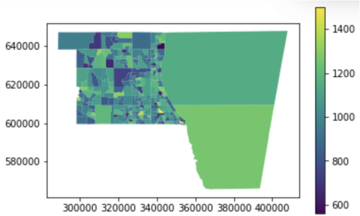
Join Example: Grouped Data

```
def f(x):
    a=x['avgNP'].max()
    y=x.query('avgNP=='+str(a))
    return y

result=joined_data.groupby(['TRACTCE']).apply(f)
fig, ax = plt.subplots(1, 1)
result.plot(column='avgNP', ax=ax, legend=True)
```



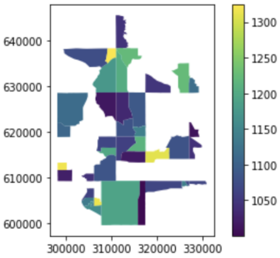
Join Example: Appended Columns

```
fig, ax = plt.subplots(1, 1)
result.plot(column='tract_scor', ax=ax, legend=True)
```



Join Example: Filtering on Joined Data

```
fig, ax = plt.subplots(1, 1)
filt_result=result.query('avgNP>2.6&tract_scor>1000')
filt_result.plot(column='tract_scor', ax=ax, legend=True)
```



[1] Paul Bolstad. *GIS Fundamentals*. University of Minnesota, 2012.

[2] Noel A. C. Cressie. *Statistics for Spatial Data*. Wiley, 2015.

[3] Wes McKinney. *Python for Data Analysis*. O'Reilly, 2013.

[4] Hanan Samet. *Foundations of Multidimensional and Metric Data Structures*. Morgan Kaufmann, 2006.