

> Geographic Data Science
with
PySAL

and the

pydata stack

Sergio J. Rey

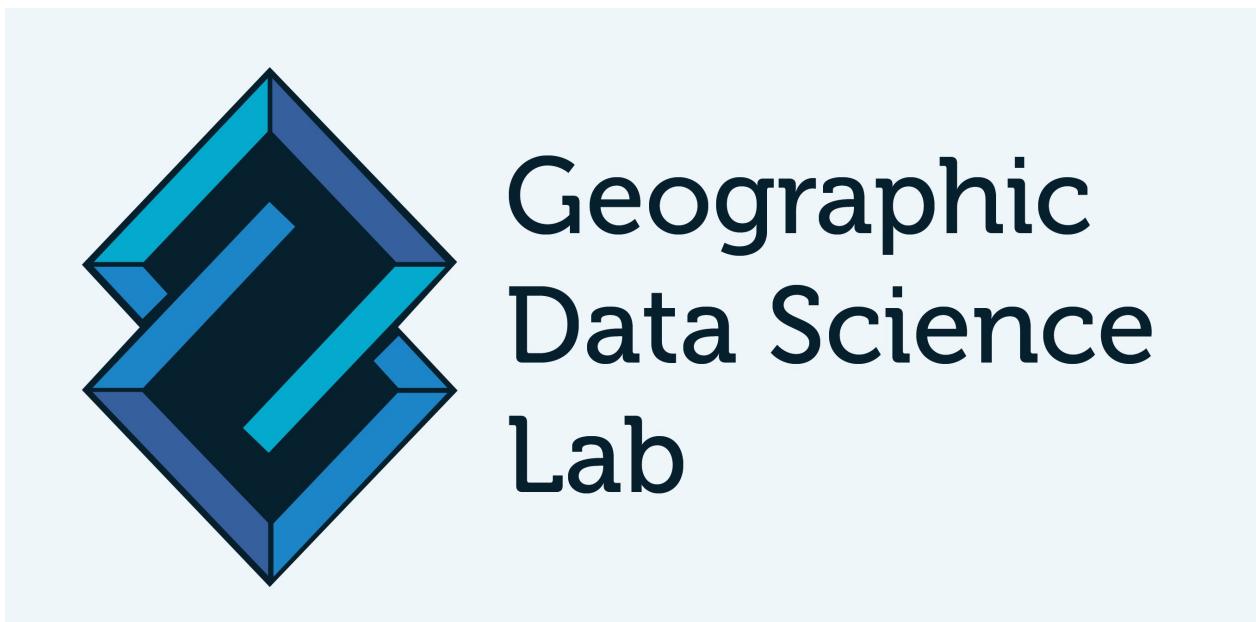
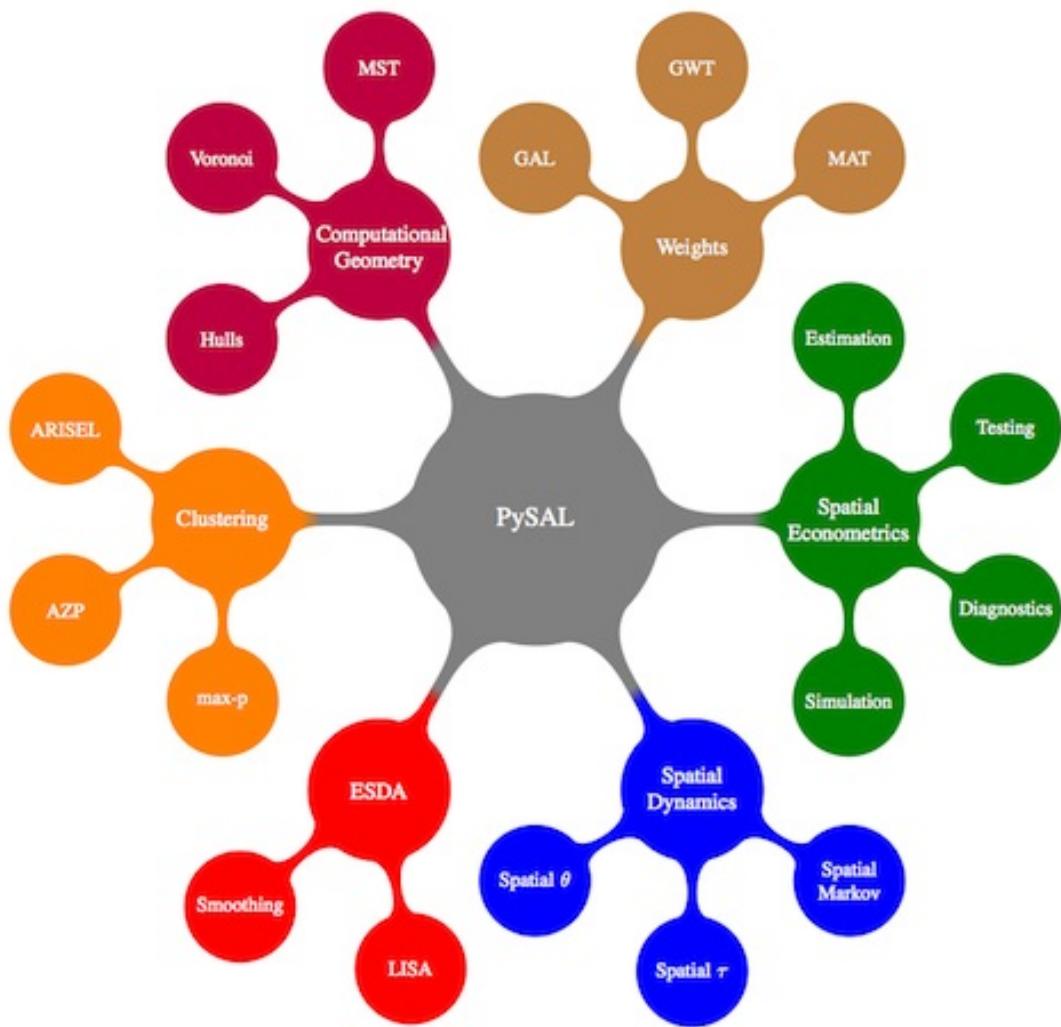
Dani Arribas-Bel

Table of Contents

Introduction	1.1
Distribution	1.2
About the authors	1.3
Install guide	1.4
Outline	1.5
Data	1.6
Part I	1.7
Software and Tools Installation	1.7.1
Spatial data processing with PySAL	1.7.2
Geovisualization	1.7.3
Spatial weights in PySAL	1.7.4
ESDA with PySAL	1.7.5
Space-time analysis	1.7.6
Part II	1.8
Points	1.8.1
Spatial clustering	1.8.2
Spatial Regression	1.8.3
Development notes	1.9

Geographic Data Science with PySAL and the pydata stack

This two-part tutorial will first provide participants with a gentle introduction to Python for geospatial analysis, and an introduction to version `PySAL` 1.11 and the related eco-system of libraries to facilitate common tasks for Geographic Data Scientists. The first part will cover munging geo-data and exploring relations over space. This includes importing data in different formats (e.g. shapefile, GeoJSON), visualizing, combining and tidying them up for analysis, and will use libraries such as `pandas`, `geopandas`, `PySAL`, or `rasterio`. The second part will provide a gentle overview to demonstrate several techniques that allow to extract geospatial insight from the data. This includes spatial clustering and regression and point pattern analysis, and will use libraries such as `PySAL`, `scikit-learn`, or `clusterpy`. A particular emphasis will be set on presenting concepts through visualization, for which libraries such as `matplotlib`, `seaborn`, and `folium` will be used.



Distribution

[URL] [PDF] [EPUB] [MOBI] [IPYNB]

License



Geographic Data Science with PySAL and the pydata stack by [Sergio J. Rey](#) and [Dani Arribas-Bel](#) is licensed under a [Creative Commons Attribution-NonCommercial-ShareAlike 4.0 International License](#).

About the authors



[Sergio Rey](#) is professor of geographical sciences and core faculty member of the GeoDa Center for Geospatial Analysis and Computation at the Arizona State University. His research interests include open science, spatial and spatio-temporal data analysis, spatial econometrics, visualization, high performance geocomputation, spatial inequality dynamics, integrated multiregional modeling, and regional science. He co-founded the Python Spatial Analysis Library (PySAL) in 2007 and continues to direct the PySAL project. Rey is a fellow of the spatial econometrics association and editor of the journal Geographical Analysis.



[Dani Arribas-Bel](#) is Lecturer in Geographic Data Science and member of the Geographic Data Science Lab at the University of Liverpool (UK). Dani is interested in understanding cities as well as in the quantitative and computational methods required to leverage the power of the large amount of urban data increasingly becoming available. He is also part of the team of core developers of PySAL, the open-source library written in Python for spatial analysis. Dani regularly teaches Geographic Data Science and Python courses at the University of Liverpool and has designed and developed several workshops at different levels on spatial analysis and econometrics, Python and open source scientific computing.

Acknowledgements

This document has also received contributions from the following people:

- Levi John Wolf.
- Wei Kan.

Install guide

The materials for the workshop and all software packages have been tested on Python 2 and 3 on the following three platforms:

- Linux (Ubuntu-Mate x64)
- Windows 10 (x64)
- Mac OS X (10.11.5 x64).

The workshop depends on the following libraries/versions:

- numpy>=1.11
- pandas>=0.18.1
- matplotlib>=1.5.1
- jupyter>=1.0
- seaborn>=0.7.0
- pip>=8.1.2
- geopandas>=0.2
- pysal>=1.11.1
- cartopy>=0.14.2
- pyproj>=1.9.5
- shapely>=1.5.16
- geopy>=1.10.0
- scikit-learn>=0.17.1
- bokeh>0.11.1
- mplleaflet>=0.0.5
- datashader>=0.2.0
- geojson>=1.3.2
- folium>=0.2.1
- statsmodels>=0.6.1
- xlrd>=1.0.0
- xlsxwriter>=0.9.2

Linux/Mac OS X

1. Install Anaconda
2. Get the most up to date version:

```
> conda update conda
```

1. Add the `conda-forge` channel:

```
> conda config --add channels conda-forge
```

1. Create an environment named `gds-scipy16` :

```
> conda create --name gds-scipy16 python=3 pandas numpy matplotlib bokeh seaborn scikit-learn jupyter
```

1. Install additional dependencies:

```
> conda install --name gds-scipy16 geopandas==0.2 mplleaflet==0.0.5 datashader==0.2.0 geojson cartopy==0.14.2 folium==0.2.1
```

1. To activate and launch the notebook:

```
> source activate gds-scipy16  
> jupyter notebook
```

Windows

1. Install [Anaconda3-4.0.0-Windows-x86-64](#)

2. open a cmd window

3. Get the most up to date version:

```
> conda update conda
```

1. Add the `conda-forge` channel:

```
> conda config --add channels conda-forge
```

1. Create an environment named `gds-scipy16` :

```
> conda create --name gds-scipy16 pandas numpy matplotlib bokeh seaborn scikit-learn jupyter geopandas==0.2 mplleaflet==0.0.5 datashader==0.2.0 geojson cartopy==0.14.2 folium==0.2.1
```

1. To activate and launch the notebook:

```
> activate gds-scipy16  
> jupyter notebook
```


Outline

Part I

1. Software and Tools Installation (10 min)
2. Spatial data processing with PySAL (45 min)
 - a. Input-output
 - b. Visualization and Mapping
 - c. Spatial weights
3. Exercise (10 min)
4. ESDA with PySAL (45 min)
 - a. Global Autocorrelation
 - b. Local Autocorrelation
 - c. Space-Time exploratory analysis
5. Exercise (10 min)

Part II

1. Point Patterns (30 min)
 - a. Point visualization
 - b. Kernel Density Estimation
2. Exercise (10 min)
3. Spatial clustering (30 min)
 - a. Geodemographic analysis
 - b. Regionalization
4. Exercise (30 min)
5. Spatial Regression (30 min)

- a. Baseline (nonspatial) regression
 - b. Exogenous and endogenous spatially lagged regressors
 - c. Prediction performance of spatial models
6. Exercise (10 min)

Data

This tutorial makes use of a variety of data sources. Below is a brief description of each dataset as well as the links to the original source where the data was downloaded from. For convenience, we have repackaged the data and included them in the compressed file with the notebooks. You can download it [here](#).

Texas counties

This includes Texas counties from the Census Bureau and a list of attached socio-economic variables. This is an extract of the national cover dataset `NAT` that is part of the example datasets shipped with `PysAL`.

AirBnb listing for Austin (TX)

This dataset contains information for [AirBnb](#) properties for the area of Austin (TX). It is originally provided by [Inside AirBnb](#). Same as the source, the dataset is released under a [CC0 1.0 Universal License](#). You can see a summary of the dataset [here](#).

Source: [Inside AirBnb](#)'s extract of AirBnb locations in Austin (TX).

Path: `data/listings.csv.gz`

Austin Zipcodes

Boundaries for Zipcodes in Austin. The original source is provided by the City of Austin GIS Division.

Source: open data from the city of Austin [\[url\]](#)

Path: `data/Zipcodes.geojson`

Part I

Software and Tools Installation

Dependencies

Participants should have installed the following dependencies:

- [Anaconda](#) or [MiniConda](#) Python distributions for Python 2.7. See installation instructions on the links.
- `git`
- A `conda` environment loaded with all the dependencies can be installed by running the `pydata.sh` script available as part of the `envs` repository ([Github link](#)). To install it, follow these instructions:
 - Clone the repository on your machine:

```
> git clone https://github.com/darribas/envs.git
```
 - Navigate into the folder:

```
> cd envs
```
 - Run the script:

```
> bash pydata.sh
```

Once installed, you need to activate the environment to run the notebooks. In Windows, open up [PowerShell](#) and type:

```
> activate pydata
```

And if you are on GNU/Linux or OSX:

```
> source activate pydata
```

Get started

Instructions to fire up a notebook here.

Spatial Data Processing with PySAL & Pandas

IPYNB

```
#by convention, we use these shorter two-letter names
import pysal as ps
import pandas as pd
import numpy as np
```

PySAL has two simple ways to read in data. But, first, you need to get the path from where your notebook is running on your computer to the place the data is. For example, to find where the notebook is running:

```
!pwd # on windows !cd
```

```
/home/serge/Dropbox/p/pysal/workshops/scipy16/gds_scipy16/content/part1
```

PySAL has a command that it uses to get the paths of its example datasets. Let's work with a commonly-used dataset first.

```
ps.examples.available()
```

```
[ 'arcgis',
  'street_net_pts',
  'Point',
  'taz',
  'Line',
  'calemp',
  'baltim',
  'south',
  'burkitt',
  'mexico',
  'sids2',
  'geodanet',
  'newHaven',
  'networks',
  'snow_maps',
  'us_income',
  '10740',
  'chicago',
  'juvenile',
  'stl',
  'desmith',
  'virginia',
  'book',
  'nat',
  'Polygon',
  'columbus',
  'wmat',
  'sacramento2']
```

```
ps.examples.explain('us_income')
```

```
{'description': 'Per-capita income for the lower 47 US states 1929-2010',
 'explanation': [' * us48.shp: shapefile ',
  ' * us48.dbf: dbf for shapefile',
  ' * us48.shx: index for shapefile',
  ' * usjoin.csv: attribute data (comma delimited file)'],
 'name': 'us_income'}
```

```
csv_path = ps.examples.get_path('usjoin.csv')
```

```
f = ps.open(csv_path)
f.header
```

```
[ 'Name',
  'STATE_FIPS',
  '1929',
  '1930',
  '1931',
  '1932',
  '1933',
  '1934',
  '1935',
  '1936',
  '1937',
  '1938',
  '1939',
  '1940',
  '1941',
  '1942',
  '1943',
  '1944',
  '1945',
  '1946',
  '1947',
  '1948',
  '1949',
  '1950',
  '1951',
  '1952',
  '1953',
  '1954',
  '1955',
  '1956',
  '1957',
  '1958',
  '1959',
  '1960',
  '1961',
  '1962',
  '1963',
  '1964',
  '1965',
  '1966',
  '1967',
  '1968',
  '1969',
  '1970',
  '1971',
  '1972',
  '1973',
  '1974',
  '1975',
  '1976',
  '1977',
```

```
'1978',
'1979',
'1980',
'1981',
'1982',
'1983',
'1984',
'1985',
'1986',
'1987',
'1988',
'1989',
'1990',
'1991',
'1992',
'1993',
'1994',
'1995',
'1996',
'1997',
'1998',
'1999',
'2000',
'2001',
'2002',
'2003',
'2004',
'2005',
'2006',
'2007',
'2008',
'2009']
```

```
y2009 = f.by_col('2009')
```

```
y2009
```

```
[32274,  
 32077,  
 31493,  
 40902,  
 40093,  
 52736,  
 40135,  
 36565,  
 33086,  
 30987,  
 40933,  
 33174,  
 35983,  
 37036,  
 31250,  
 35151,  
 35268,  
 47159,  
 49590,  
 34280,  
 40920,  
 29318,  
 35106,  
 32699,  
 37057,  
 38009,  
 41882,  
 48123,  
 32197,  
 46844,  
 33564,  
 38672,  
 35018,  
 33708,  
 35210,  
 38827,  
 41283,  
 30835,  
 36499,  
 33512,  
 35674,  
 30107,  
 36752,  
 43211,  
 40619,  
 31843,  
 35676,  
 42504]
```

Working with shapefiles

We can also work with local files outside the built-in examples.

To read in a shapefile, we will need the path to the file.

```
shp_path = '../data/texas.shp'  
print(shp_path)
```

```
../data/texas.shp
```

Then, we open the file using the `ps.open` command:

```
f = ps.open(shp_path)
```

`f` is what we call a "file handle." That means that it only *points* to the data and provides ways to work with it. By itself, it does not read the whole dataset into memory. To see basic information about the file, we can use a few different methods.

For instance, the header of the file, which contains most of the metadata about the file:

```
f.header
```

```
{'BBOX_Mmax': 0.0,  
'BBOX_Mmin': 0.0,  
'BBOX_Xmax': -93.50721740722656,  
'BBOX_Xmin': -106.6495132446289,  
'BBOX_Ymax': 36.49387741088867,  
'BBOX_Ymin': 25.845197677612305,  
'BBOX_Zmax': 0.0,  
'BBOX_Zmin': 0.0,  
'File Code': 9994,  
'File Length': 49902,  
'Shape Type': 5,  
'Unused0': 0,  
'Unused1': 0,  
'Unused2': 0,  
'Unused3': 0,  
'Unused4': 0,  
'Version': 1000}
```

To actually read in the shapes from memory, you can use the following commands:

```
f.by_row(14) #gets the 14th shape from the file
```

```
<pysal.cg.shapes.Polygon at 0x7fc5b08a3860>
```

```
all_polygons = f.read() #reads in all polygons from memory
```

```
len(all_polygons)
```

```
254
```

So, all 254 polygons have been read in from file. These are stored in PySAL shape objects, which can be used by PySAL and can be converted to other Python shape objects.

They typically have a few methods. So, since we've read in polygonal data, we can get some properties about the polygons. Let's just have a look at the first polygon:

```
all_polygons
```

```
[<pysal.cg.shapes.Polygon at 0x7fc5b08a3748>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a39e8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a36a0>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3710>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a38d0>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3ac8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3b38>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3ba8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3c18>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3c88>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3cf8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3d68>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3dd8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3e48>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3ef0>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3f60>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3fd0>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a20f0>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a3e80>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a22e8>,
 <pysal.cg.shapes.Polygon at 0x7fc5b08a2320>,
 <pysal.cg.shapes.Polygon at 0x7fc5b001b208>,
 <pysal.cg.shapes.Polygon at 0x7fc5b001b908>,
```

```
<pysal.cg.shapes.Polygon at 0x7fc5b001b048>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b0b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b128>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a2240>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b198>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b518>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bc50>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bd30>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bb38>,
<pysal.cg.shapes.Polygon at 0x7fc5b001ba90>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b9e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b390>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b5c0>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b4e0>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b7b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b6d8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b780>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b978>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b940>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bdd8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001be48>,
<pysal.cg.shapes.Polygon at 0x7fc5b001beb8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bf28>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bf98>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b278>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b6a0>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b7f0>,
<pysal.cg.shapes.Polygon at 0x7fc5b001bbe0>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b2e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b828>,
<pysal.cg.shapes.Polygon at 0x7fc5b001b8d0>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a40b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4128>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4198>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4208>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4278>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a42e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4358>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a43c8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4438>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a44a8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4518>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4588>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a45f8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4668>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a46d8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4748>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a47b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4828>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4898>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4908>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4978>,
```

```
<pysal.cg.shapes.Polygon at 0x7fc5b08a49e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4a58>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4ac8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4b38>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4ba8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4c18>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4c88>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4cf8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4d68>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4dd8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4e48>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4eb8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4f28>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4f98>,
<pysal.cg.shapes.Polygon at 0x7fc5b08a4fd0>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b40b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4128>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4198>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4208>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4278>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b42e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4358>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b43c8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4438>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b44a8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4518>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4588>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b45f8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4668>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b46d8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4748>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b47b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4828>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4898>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4908>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4978>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b49e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4a58>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4ac8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4b38>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4ba8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4c18>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4c88>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4cf8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4d68>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4dd8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4e48>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4eb8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4f28>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4f98>,
<pysal.cg.shapes.Polygon at 0x7fc5b08b4fd0>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd0b8>,
```

```
<pysal.cg.shapes.Polygon at 0x7fc5b08cd128>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd198>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd208>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd278>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd2e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd358>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd3c8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd438>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd4a8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd518>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd588>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd5f8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd668>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd6d8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd748>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd7b8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd828>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd898>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd908>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd978>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cd9e8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdac8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdb38>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdba8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdc18>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdc88>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdcf8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdd68>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cddd8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cde48>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdeb8>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdf28>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdf98>,
<pysal.cg.shapes.Polygon at 0x7fc5b08cdfd0>,
<pysal.cg.shapes.Polygon at 0x7fc5afff30b8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3128>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3198>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3208>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3278>,
<pysal.cg.shapes.Polygon at 0x7fc5afff32e8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3358>,
<pysal.cg.shapes.Polygon at 0x7fc5afff33c8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3438>,
<pysal.cg.shapes.Polygon at 0x7fc5afff34a8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3518>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3588>,
<pysal.cg.shapes.Polygon at 0x7fc5afff35f8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3668>,
<pysal.cg.shapes.Polygon at 0x7fc5afff36d8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3748>,
<pysal.cg.shapes.Polygon at 0x7fc5afff37b8>,
```

```
<pysal.cg.shapes.Polygon at 0x7fc5afff3828>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3898>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3908>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3978>,
<pysal.cg.shapes.Polygon at 0x7fc5afff39e8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3a58>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3ac8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3b38>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3ba8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3c18>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3c88>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3cf8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3d68>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3dd8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3e48>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3eb8>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3f28>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3f98>,
<pysal.cg.shapes.Polygon at 0x7fc5afff3fd0>,
<pysal.cg.shapes.Polygon at 0x7fc5aff940b8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94128>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94198>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94208>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94278>,
<pysal.cg.shapes.Polygon at 0x7fc5aff942e8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94358>,
<pysal.cg.shapes.Polygon at 0x7fc5aff943c8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94438>,
<pysal.cg.shapes.Polygon at 0x7fc5aff944a8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94518>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94588>,
<pysal.cg.shapes.Polygon at 0x7fc5aff945f8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94668>,
<pysal.cg.shapes.Polygon at 0x7fc5aff946d8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94748>,
<pysal.cg.shapes.Polygon at 0x7fc5aff947b8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94828>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94898>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94908>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94978>,
<pysal.cg.shapes.Polygon at 0x7fc5aff949e8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94a58>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94a20>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94ba8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94c18>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94c88>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94cf8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94d68>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94dd8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94e48>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94eb8>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94f28>,
```

```
<pysal.cg.shapes.Polygon at 0x7fc5aff94f98>,
<pysal.cg.shapes.Polygon at 0x7fc5aff94fd0>,
<pysal.cg.shapes.Polygon at 0x7fc5affb90b8>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9128>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9198>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9208>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9278>,
<pysal.cg.shapes.Polygon at 0x7fc5affb92e8>,
<pysal.cg.shapes.Polygon at 0x7fc5affb92b0>,
<pysal.cg.shapes.Polygon at 0x7fc5affb94a8>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9518>,
<pysal.cg.shapes.Polygon at 0x7fc5affb94e0>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9668>,
<pysal.cg.shapes.Polygon at 0x7fc5affb96d8>,
<pysal.cg.shapes.Polygon at 0x7fc5affb96a0>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9710>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9898>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9908>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9978>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9940>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9ac8>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9b38>,
<pysal.cg.shapes.Polygon at 0x7fc5affb9b00>]
```

```
all_polygons[0].centroid #the centroid of the first polygon
```

```
( -100.27156110567945, 36.27508640938005)
```

```
all_polygons[0].area
```

```
0.23682222998468205
```

```
all_polygons[0].perimeter
```

```
1.9582821721538344
```

While in the Jupyter Notebook, you can examine what properties an object has by using the tab key.

```
polygon = all_polygons[0]
```

```
polygon. #press tab when the cursor is right after the dot
```

```
File "<ipython-input-20-aa03438a2fa8>", line 1
polygon. #press tab when the cursor is right after the dot
      ^
SyntaxError: invalid syntax
```

Working with Data Tables

```
dbf_path = "../data/texas.dbf"
print(dbf_path)
```

```
../data/texas.dbf
```

When you're working with tables of data, like a `csv` or `dbf`, you can extract your data in the following way. Let's open the dbf file we got the path for above.

```
f = ps.open(dbf_path)
```

Just like with the shapefile, we can examine the header of the dbf file.

```
f.header
```

```
['NAME',
 'STATE_NAME',
 'STATE_FIPS',
 'CNTY_FIPS',
 'FIPS',
 'STFIPS',
 'COFIPS',
 'FIPSNO',
 'SOUTH',
 'HR60',
 'HR70',
 'HR80',
 'HR90',
 'HC60',
 'HC70',
 'HC80',
 'HC90',
 'PO60',
```

```
'P070',
'P080',
'P090',
'RD60',
'RD70',
'RD80',
'RD90',
'PS60',
'PS70',
'PS80',
'PS90',
'UE60',
'UE70',
'UE80',
'UE90',
'DV60',
'DV70',
'DV80',
'DV90',
'MA60',
'MA70',
'MA80',
'MA90',
'POL60',
'POL70',
'POL80',
'POL90',
'DNL60',
'DNL70',
'DNL80',
'DNL90',
'MFIL59',
'MFIL69',
'MFIL79',
'MFIL89',
'FP59',
'FP69',
'FP79',
'FP89',
'BLK60',
'BLK70',
'BLK80',
'BLK90',
'GI59',
'GI69',
'GI79',
'GI89',
'FH60',
'FH70',
'FH80',
'FH90']
```

So, the header is a list containing the names of all of the fields we can read. If we just wanted to grab the data of interest, `HR90`, we can use either `by_col` or `by_col_array`, depending on the format we want the resulting data in:

```
HR90 = f.by_col('HR90')
print(type(HR90).__name__, HR90[0:5])
HR90 = f.by_col_array('HR90')
print(type(HR90).__name__, HR90[0:5])
```

```
list [0.0, 0.0, 18.31166453, 0.0, 3.6517674554]
ndarray [[ 0.          ]
 [ 0.          ]
 [ 18.31166453]
 [ 0.          ]
 [ 3.65176746]]
```

As you can see, the `by_col` function returns a list of data, with no shape. It can only return one column at a time:

```
HRs = f.by_col('HR90', 'HR80')
```

```
-----
TypeError                                     Traceback (most recent call last)

<ipython-input-25-1fef6a3c3a50> in <module>()
----> 1 HRs = f.by_col('HR90', 'HR80')

TypeError: __call__() takes 2 positional arguments but 3 were given
```

This error message is called a "traceback," as you see in the top right, and it usually provides feedback on why the previous command did not execute correctly. Here, you see that one-too-many arguments was provided to `__call__`, which tells us we cannot pass as many arguments as we did to `by_col`.

If you want to read in many columns at once and store them to an array, use `by_col_array`:

```
HRs = f.by_col_array('HR90', 'HR80')
```

HRS

```
array([[ 0.          ,  0.          ],
       [ 0.          ,  10.50199538],
       [ 18.31166453,  5.10386362],
       [ 0.          ,  0.          ],
       [ 3.65176746,  10.4297038 ],
       [ 0.          ,  0.          ],
       [ 0.          ,  18.85369532],
       [ 2.59514448,  6.33617194],
       [ 0.          ,  0.          ],
       [ 5.59753708,  6.0331825 ],
       [ 17.3692707 ,  14.53139626],
       [ 5.06893755,  4.9960032 ],
       [ 9.73560868,  8.84307335],
       [ 5.66989851,  4.67049647],
       [ 0.          ,  0.          ],
       [ 4.08893052,  4.44077341],
       [ 16.49348507,  0.          ],
       [ 0.          ,  7.17154332],
       [ 0.          ,  8.1799591 ],
       [ 3.48074279,  7.8746358 ],
       [ 0.          ,  6.0397415 ],
       [ 3.67511944,  12.63104711],
       [ 8.19705726,  17.14148582],
       [ 0.          ,  0.          ],
       [ 11.19883532,  4.79616307],
       [ 8.53606487,  5.95876534],
       [ 6.30954634,  5.23450586],
       [ 19.83995768,  8.36942649],
       [ 43.51610096,  0.          ],
       [ 14.83459427,  0.          ],
       [ 7.8459064 ,  6.77920141],
       [ 5.76850971,  12.41398879],
       [ 2.21160651,  5.3564733 ],
       [ 4.7187618 ,  16.32386549],
       [ 0.          ,  0.          ],
       [ 13.6190056 ,  9.08475248],
       [ 0.          ,  0.          ],
       [ 5.78904712,  5.74382539],
       [ 11.6411725 ,  10.35132393],
       [ 11.57638838,  5.93938854],
       [ 8.66447889,  7.23170379],
       [ 7.58454876,  11.06999399],
       [ 16.12643122,  4.11776817],
       [ 0.          ,  0.          ],
       [ 0.          ,  9.41885655],
       [ 0.          ,  11.28795575],
       [ 8.68383071,  18.26906873],
```

```
[ 4.18077679,  0.        ],
[ 5.50986955,  7.17463051],
[ 20.67397147, 18.76524676],
[ 0.        ,  0.        ],
[ 0.        , 13.81692573],
[ 7.75526031,  8.85335741],
[ 13.7258939 , 20.66542674],
[ 9.54973022,  4.49964003],
[ 4.80598249,  5.01724679],
[ 5.24022789,  4.89079552],
[ 9.3250237 ,  9.65344145],
[ 3.15613527,  6.68621809],
[ 15.27205076, 15.54581351],
[ 0.        , 4.31499461],
[ 0.        ,  0.        ],
[ 3.677958 ,  3.4935108 ],
[ 0.        ,  0.        ],
[ 16.55903295, 13.85425326],
[ 4.27240878, 24.17911891],
[ 12.96260289,  6.24687656],
[ 0.        ,  0.        ],
[ 5.04362738,  6.85824018],
[ 0.        ,  4.01654818],
[ 5.78041365,  7.92173328],
[ 0.        , 29.62152801],
[ 11.11778178,  7.92841771],
[ 16.82821755, 10.78167116],
[ 4.53823463,  8.09814957],
[ 5.32162576, 13.85310171],
[ 5.14522395,  3.73616684],
[ 15.7535989 , 15.99138866],
[ 24.88112651, 22.42368558],
[ 7.81127949,  4.58883994],
[ 14.89203276, 13.77695116],
[ 0.        , 8.51426139],
[ 6.06428138,  9.65176434],
[ 0.        , 5.65834889],
[ 7.15537906, 10.99384345],
[ 0.        ,  0.        ],
[ 16.26129579,  6.17894217],
[ 3.69959304,  6.71636779],
[ 0.        , 10.13941698],
[ 4.25034534, 12.82275456],
[ 13.35470086, 35.39253539],
[ 22.97970126, 16.23307275],
[ 7.02790077,  6.36415707],
[ 8.69822382, 15.94438598],
[ 10.13378803, 16.35947213],
[ 15.88088069, 17.42271184],
[ 5.83200398,  7.39109226],
[ 4.60071541,  7.52700312],
[ 9.78469752, 12.27479928],
```

```
[ 10.30741159,  11.11792209],  
[ 11.33681683,  23.13278217],  
[  8.31669993,  11.00352113],  
[  9.01485648,   5.13347023],  
[  5.62160947,  3.03250849],  
[  0.          ,  7.11642471],  
[  6.97447343,  5.00387801],  
[  6.02627456,  5.7607005 ],  
[  2.38171793,  4.43262411],  
[  9.14599291, 17.72107035],  
[  0.          , 12.86752879],  
[  9.67949484, 14.86488601],  
[ 12.52316786, 15.0987553 ],  
[  6.21890547, 16.04878832],  
[  9.82342395, 13.32054561],  
[  7.47328301,  5.28387625],  
[  6.61157025, 12.43688282],  
[  9.74445175,  6.11989754],  
[  0.          ,  0.          ],  
[  0.          , 25.56237219],  
[  7.50391611, 12.90801426],  
[  0.          ,  0.          ],  
[  3.86428627, 23.46473585],  
[ 11.77123447, 21.66866018],  
[  0.          ,  8.42318059],  
[  3.43288706,  6.38630776],  
[  3.87923928,  9.07523369],  
[ 12.49375312, 15.63273495],  
[  4.31053063,  0.          ],  
[ 12.64382349,  2.24769611],  
[ 39.13511398,  0.          ],  
[  6.30835226, 12.65742675],  
[  0.          ,  0.          ],  
[ 11.43510578,  0.          ],  
[  7.66270572,  8.82130059],  
[ 21.17938943, 20.21602264],  
[ 16.56770109, 10.15099607],  
[ 10.95830365, 15.67420453],  
[  7.95696871, 16.48206751],  
[  0.          ,  0.          ],  
[  4.67195116,  2.34878245],  
[  8.125292  ,  7.86311883],  
[  5.08323802, 11.92520511],  
[  8.33437513, 11.38303927],  
[  0.          ,  7.24637681],  
[  7.49568998, 14.43313849],  
[  0.          ,  0.          ],  
[  7.89577576, 17.37196859],  
[  6.95458655,  7.66107408],  
[ 21.83235867, 14.94835344],  
[ 10.95050372,  0.          ],  
[  0.          , 24.05002405],
```

```
[ 15.26338885,    7.27212284],  
[  7.52785306,   11.14454474],  
[  0.          ,  15.26426254],  
[  6.1716966 ,  10.74575543],  
[  4.93060178,   8.32986256],  
[  8.73743993,   7.05467372],  
[ 13.6286201 ,   4.5605874 ],  
[  6.44703759,  15.92392912],  
[ 12.21077898,  13.08936447],  
[ 17.19605965,   7.54489211],  
[  7.50219707,   9.74627205],  
[  9.7761267 ,  16.38874094],  
[  7.26343008,  17.59633996],  
[ 17.12915382,   0.          ],  
[  0.          ,  18.78110621],  
[ 11.14827202,   0.          ],  
[  0.          ,   0.          ],  
[ 24.52182442,   7.2337963 ],  
[ 10.01241539,   4.10939202],  
[  7.20126218,  10.36955499],  
[  5.8796725 ,  3.74468722],  
[  4.37653521,   7.4795914 ],  
[ 19.4760931 ,  9.05059281],  
[  2.86590434,  9.85804416],  
[  4.06064211,  3.92049241],  
[ 12.2159785 ,   8.74584572],  
[  5.31123858,  17.18213058],  
[  2.44648318,  18.95016108],  
[ 24.26006793,   8.20411847],  
[ 24.18379686,   0.          ],  
[ 15.35921361,   4.40160218],  
[  0.          ,   0.          ],  
[ 10.04469891,   6.42508353],  
[  8.15396066,  12.39355249],  
[ 11.52573257,  12.19319205],  
[ 12.96613246,   9.13075237],  
[  4.03355921,  9.82294148],  
[  0.          ,   0.          ],  
[ 16.74480911,   7.12098554],  
[ 11.37958502,  22.65262204],  
[ 10.45396336,  14.82919464],  
[  6.37251153,  12.12231415],  
[  5.08021662,  12.3170912 ],  
[  0.          ,   0.          ],  
[  6.88687448,  3.71298617],  
[  7.34537976,  5.79105861],  
[ 14.25110446,  13.46937401],  
[ 10.35080964,  8.34943582],  
[ 15.03778243,  14.47900809],  
[  9.95272456,  10.6202209 ],  
[ 23.54931879,  32.62023941],  
[  4.56965293,   0.          ],
```

```
[ 8.40392631, 15.04381511],  
[ 27.63957988, 0. ],  
[ 3.78902698, 8.46131066],  
[ 4.50172352, 6.40216576],  
[ 5.43980852, 10.62529884],  
[ 6.31193587, 4.70543949],  
[ 13.27492367, 8.99054195],  
[ 9.24884004, 9.27749707],  
[ 8.13293053, 15.03039196],  
[ 23.24905551, 15.79498114],  
[ 18.39050982, 19.38376483],  
[ 8.54325327, 4.31704369],  
[ 12.48439451, 0. ],  
[ 7.508447 , 8.28321985],  
[ 11.42530705, 11.88301175],  
[ 0. , 0. ],  
[ 5.91179943, 10.22090923],  
[ 15.17946265, 18.03273111],  
[ 10.30169242, 13.92536007],  
[ 8.84642604, 7.05355411],  
[ 5.11286653, 12.48252447],  
[ 5.45857488, 10.64325151],  
[ 10.83188908, 21.14835572],  
[ 5.3526027 , 9.8089703 ],  
[ 7.17221841, 8.72004302],  
[ 24.66699556, 8.5719184 ],  
[ 12.37133808, 12.09043646],  
[ 8.24674254, 0. ],  
[ 0. , 6.41889723],  
[ 0. , 3.47005344],  
[ 5.30468802, 16.64745806],  
[ 31.72186271, 18.1356547 ],  
[ 40.7996736 , 0. ],  
[ 9.58497077, 11.72986129],  
[ 4.17920428, 3.58847382],  
[ 5.24851729, 11.92057491],  
[ 9.25654901, 11.08223015],  
[ 6.80862653, 10.9170933 ],  
[ 5.5890901 , 2.33754091],  
[ 18.0626516 , 26.63044926],  
[ 5.3079965 , 10.95950463],  
[ 10.30414398, 17.39897719],  
[ 5.50527405, 13.98964766],  
[ 13.04886801, 6.4499484 ],  
[ 10.77702339, 0. ],  
[ 8.12611734, 27.6854928 ],  
[ 0. , 0. ],  
[ 14.80823338, 17.11533289],  
[ 7.30031678, 8.23832776],  
[ 5.648122 , 7.62122511],  
[ 12.30201446, 11.76132146]])
```

It is best to use `by_col_array` on data of a single type. That is, if you read in a lot of columns, some of them numbers and some of them strings, all columns will get converted to the same datatype:

```
allcolumns = f.by_col_array(['NAME', 'STATE_NAME', 'HR90', 'HR80'])
```

```
allcolumns
```

```
array([['Lipscomb', 'Texas', '0.0', '0.0'],
       ['Sherman', 'Texas', '0.0', '10.501995379'],
       ['Dallam', 'Texas', '18.31166453', '5.1038636248'],
       ...,
       ['Hidalgo', 'Texas', '7.3003167816', '8.2383277607'],
       ['Willacy', 'Texas', '5.6481219994', '7.6212251119'],
       ['Cameron', 'Texas', '12.302014455', '11.761321464']],
      dtype='|<U13')
```

Note that the numerical columns, `HR90` & `HR80` are now considered strings, since they show up with the single tickmarks around them, like `'0.0'`.

These methods work similarly for `.csv` files as well.

Using Pandas with PySAL

A new functionality added to PySAL recently allows you to work with shapefile/dbf pairs using Pandas. This *optional* extension is only turned on if you have Pandas installed. The extension is the `ps.pdio` module:

```
ps.pdio
```

```
<module 'pysal.contrib.pdutilities' from '/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/pysal/contrib/pdutilities/__init__.py'>
```

To use it, you can read in shapefile/dbf pairs using the `ps.pdio.read_files` command.

```
shp_path = ps.examples.get_path('NAT.shp')
data_table = ps.pdio.read_files(shp_path)
```

This reads in *the entire database table* and adds a column to the end, called `geometry`, that stores the geometries read in from the shapefile.

Now, you can work with it like a standard pandas dataframe.

```
data_table.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
0	Lake of the Woods	Minnesota	27	077	27077
1	Ferry	Washington	53	019	53019
2	Stevens	Washington	53	065	53065
3	Okanogan	Washington	53	047	53047
4	Pend Oreille	Washington	53	051	53051

5 rows × 70 columns

The `read_files` function only works on shapefile/dbf pairs. If you need to read in data using CSVs, use pandas directly:

```
usjoin = pd.read_csv(csv_path)
#usjoin = ps.pdio.read_files(csv_path) #will not work, not a shp/dbf pair
```

```
usjoin.head()
```

	Name	STATE_FIPS	1929	1930	1931	1932	1933
0	Alabama	1	323	267	224	162	166
1	Arizona	4	600	520	429	321	308
2	Arkansas	5	310	228	215	157	157
3	California	6	991	887	749	580	546
4	Colorado	8	634	578	471	354	353

5 rows × 83 columns

The nice thing about working with pandas dataframes is that they have very powerful baked-in support for relational-style queries. By this, I mean that it is very easy to find things like:

The number of counties in each state:

```
data_table.groupby("STATE_NAME").size()
```

STATE_NAME	
Alabama	67
Arizona	14
Arkansas	75
California	58
Colorado	63
Connecticut	8
Delaware	3
District of Columbia	1
Florida	67
Georgia	159
Idaho	44
Illinois	102
Indiana	92
Iowa	99
Kansas	105
Kentucky	120
Louisiana	64
Maine	16
Maryland	24
Massachusetts	12
Michigan	83
Minnesota	87
Mississippi	82
Missouri	115
Montana	55
Nebraska	93
Nevada	17

```
New Hampshire          10
New Jersey             21
New Mexico             32
New York               58
North Carolina         100
North Dakota           53
Ohio                   88
Oklahoma               77
Oregon                 36
Pennsylvania           67
Rhode Island            5
South Carolina          46
South Dakota            66
Tennessee              95
Texas                  254
Utah                   29
Vermont                 14
Virginia                123
Washington              38
West Virginia            55
Wisconsin               70
Wyoming                 23
dtype: int64
```

Or, to get the rows of the table that are in Arizona, we can use the `query` function of the dataframe:

```
data_table.query('STATE_NAME == "Arizona"')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
1707	Navajo	Arizona	04	017	04017
1708	Coconino	Arizona	04	005	04005
1722	Mohave	Arizona	04	015	04015
1726	Apache	Arizona	04	001	04001
2002	Yavapai	Arizona	04	025	04025
2182	Gila	Arizona	04	007	04007
2262	Maricopa	Arizona	04	013	04013
2311	Greenlee	Arizona	04	011	04011
2326	Graham	Arizona	04	009	04009
2353	Pinal	Arizona	04	021	04021
2499	Pima	Arizona	04	019	04019
2514	Cochise	Arizona	04	003	04003
2615	Santa Cruz	Arizona	04	023	04023
3080	La Paz	Arizona	04	012	04012

14 rows × 70 columns

Behind the scenes, this uses a fast vectorized library, `numexpr`, to essentially do the following.

First, compare each row's `STATE_NAME` column to `'Arizona'` and return `True` if the row matches:

```
data_table.STATE_NAME == 'Arizona'
```

```
0      False
1      False
2      False
3      False
4      False
5      False
6      False
7      False
8      False
9      False
10     False
11     False
12     False
13     False
14     False
15     False
16     False
17     False
18     False
19     False
20     False
21     False
22     False
23     False
24     False
25     False
26     False
27     False
28     False
29     False
...
3055    False
3056    False
3057    False
3058    False
3059    False
3060    False
3061    False
3062    False
3063    False
3064    False
3065    False
3066    False
3067    False
3068    False
3069    False
3070    False
```

```
3071    False
3072    False
3073    False
3074    False
3075    False
3076    False
3077    False
3078    False
3079    False
3080    True
3081    False
3082    False
3083    False
3084    False
Name: STATE_NAME, dtype: bool
```

Then, use that to filter out rows where the condition is true:

```
data_table[data_table.STATE_NAME == 'Arizona']
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
1707	Navajo	Arizona	04	017	04017
1708	Coconino	Arizona	04	005	04005
1722	Mohave	Arizona	04	015	04015
1726	Apache	Arizona	04	001	04001
2002	Yavapai	Arizona	04	025	04025
2182	Gila	Arizona	04	007	04007
2262	Maricopa	Arizona	04	013	04013
2311	Greenlee	Arizona	04	011	04011
2326	Graham	Arizona	04	009	04009
2353	Pinal	Arizona	04	021	04021
2499	Pima	Arizona	04	019	04019
2514	Cochise	Arizona	04	003	04003
2615	Santa Cruz	Arizona	04	023	04023
3080	La Paz	Arizona	04	012	04012

14 rows × 70 columns

We might need this behind the scenes knowledge when we want to chain together conditions, or when we need to do spatial queries.

This is because spatial queries are somewhat more complex. Let's say, for example, we want all of the counties in the US to the West of `-121` longitude. We need a way to express that question. Ideally, we want something like:

```
SELECT
    *
FROM
    data_table
WHERE
    x_centroid < -121
```

So, let's refer to an arbitrary polygon in the the dataframe's geometry column as `poly`. The centroid of a PySAL polygon is stored as an `(x, y)` pair, so the longitude is the first element of the pair, `poly.centroid[0]`.

Then, applying this condition to each geometry, we get the same kind of filter we used above to grab only counties in Arizona:

```
data_table.geometry.apply(lambda x: x.centroid[0] < -121)
```

```
0      False
1      False
2      False
3      False
4      False
5      False
6      False
7      False
8      False
9      False
10     False
11     False
12     False
13     False
14     False
15     False
16     False
17     False
18     False
19     False
20     False
21     False
22     False
23     False
24     False
25     False
```

```

26    False
27    True
28    False
29    False
...
3055  False
3056  False
3057  False
3058  False
3059  False
3060  False
3061  False
3062  False
3063  False
3064  False
3065  False
3066  False
3067  False
3068  False
3069  False
3070  False
3071  False
3072  False
3073  False
3074  False
3075  False
3076  False
3077  False
3078  False
3079  False
3080  False
3081  False
3082  False
3083  False
3084  False
Name: geometry, dtype: bool

```

If we use this as a filter on the table, we can get only the rows that match that condition, just like we did for the `STATE_NAME` query:

```
data_table[data_table.geometry.apply(lambda x: x.centroid[0] < -119)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FII
3	Okanogan	Washington	53	047	530
27	Whatcom	Washington	53	073	530

31	Skagit	Washington	53	057	530
42	Chelan	Washington	53	007	530
44	Clallam	Washington	53	009	530
47	Snohomish	Washington	53	061	530
48	Island	Washington	53	029	530
55	Douglas	Washington	53	017	530
57	Jefferson	Washington	53	031	530
68	Grant	Washington	53	025	530
71	Kitsap	Washington	53	035	530
80	King	Washington	53	033	530
85	Mason	Washington	53	045	530
89	Kittitas	Washington	53	037	530
92	Grays Harbor	Washington	53	027	530
107	Pierce	Washington	53	053	530
116	Thurston	Washington	53	067	530
119	Yakima	Washington	53	077	530
130	Pacific	Washington	53	049	530
131	Lewis	Washington	53	041	530

131	Lewis	Washington	53	041	530
139	Benton	Washington	53	005	530
169	Skamania	Washington	53	059	530
170	Cowlitz	Washington	53	015	530
171	Wahkiakum	Washington	53	069	530
181	Clatsop	Oregon	41	007	410
183	Columbia	Oregon	41	009	410
193	Clark	Washington	53	011	530
194	Klickitat	Washington	53	039	530
209	Morrow	Oregon	41	049	410
213	Gilliam	Oregon	41	021	410
...
1225	Douglas	Nevada	32	005	320
1233	El Dorado	California	06	017	060
1262	Yolo	California	06	113	061
1263	Alpine	California	06	003	060
1281	Napa	California	06	055	060
1282	Sonoma	California	06	097	060

1301	Sacramento	California	06	067	060
1316	Amador	California	06	005	060
1367	Calaveras	California	06	009	060
1381	Tuolumne	California	06	109	061
1401	Marin	California	06	041	060
1404	San Joaquin	California	06	077	060
1447	Solano	California	06	095	060
1462	Contra Costa	California	06	013	060
1466	Stanislaus	California	06	099	060
1514	Alameda	California	06	001	060
1515	Mariposa	California	06	043	060
1530	San Francisco	California	06	075	060
1538	Madera	California	06	039	060
1559	San Mateo	California	06	081	060
1574	Merced	California	06	047	060
1591	Fresno	California	06	019	060
1609	Santa Clara	California	06	085	060
1664	Santa Cruz	California	06	087	060

1731	San Benito	California	06	069	060
1745	Monterey	California	06	053	060
1834	Kings	California	06	031	060
1958	San Luis Obispo	California	06	079	060
2117	Ventura	California	06	111	061
2263	Santa Barbara	California	06	083	060

109 rows × 70 columns

Other types of spatial queries

Everybody knows the following statements are true:

1. If you head directly west from Reno, Nevada, you will shortly enter California.
2. San Diego is in California.

But what does this tell us about the location of San Diego relative to Reno?

Or for that matter, how many counties in California are to the east of Reno?

```
geom = data_table.query('(NAME == "Washoe") & (STATE_NAME == "Nevada")').geometry
```

```
lon, lat = geom.values[0].centroid
```

```
lon
```

```
-119.6555030699793
```

```
cal_counties = data_table.query('(STATE_NAME=="California")')
```

```
cal_counties[cal_counties.geometry.apply(lambda x: x.centroid[0] > lon)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
1312	Mono	California	06	051	0605
1591	Fresno	California	06	019	0601
1620	Inyo	California	06	027	0602
1765	Tulare	California	06	107	0610
1956	Kern	California	06	029	0602
1957	San Bernardino	California	06	071	0607
2117	Ventura	California	06	111	0611
2255	Riverside	California	06	065	0606
2279	Orange	California	06	059	0605
2344	San Diego	California	06	073	0607
2351	Los Angeles	California	06	037	0603
2358	Imperial	California	06	025	0602

12 rows × 70 columns

```
len(cal_counties)
```

```
58
```

This works on any type of spatial query.

For instance, if we wanted to find all of the counties that are within a threshold distance from an observation's centroid, we can do it in the following way.

But first, we need to handle distance calculations on the earth's surface.

```

from math import radians, sin, cos, sqrt, asin

def gcd(loc1, loc2, R=3961):
    """Great circle distance via Haversine formula

    Parameters
    -----
    loc1: tuple (long, lat in decimal degrees)
    loc2: tuple (long, lat in decimal degrees)
    R: Radius of the earth (3961 miles, 6367 km)

    Returns
    -----
    great circle distance between loc1 and loc2 in units of R

    Notes
    -----
    Does not take into account non-spheroidal shape of the Earth

    """
    lon1, lat1 = loc1
    lon2, lat2 = loc2
    dLat = radians(lat2 - lat1)
    dLon = radians(lon2 - lon1)
    lat1 = radians(lat1)
    lat2 = radians(lat2)

    a = sin(dLat/2)**2 + cos(lat1)*cos(lat2)*sin(dLon/2)**2
    c = 2*asin(sqrt(a))

    return R * c

def gcdm(loc1, loc2):
    return gcd(loc1, loc2)

def gcdk(loc1, loc2):
    return gcd(loc1, loc2, 6367 )

```

```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

```
gcdk(san_diego, austin)
```

```
1857.3357887898544
```

```
loc1 = (-117.1611, 0.0)
loc2 = (-118.1611, 0.0)
gcd(loc1, loc2)
```

```
69.13249167149539
```

```
loc1 = (-117.1611, 45.0)
loc2 = (-118.1611, 45.0)
gcd(loc1, loc2)
```

```
48.88374342930467
```

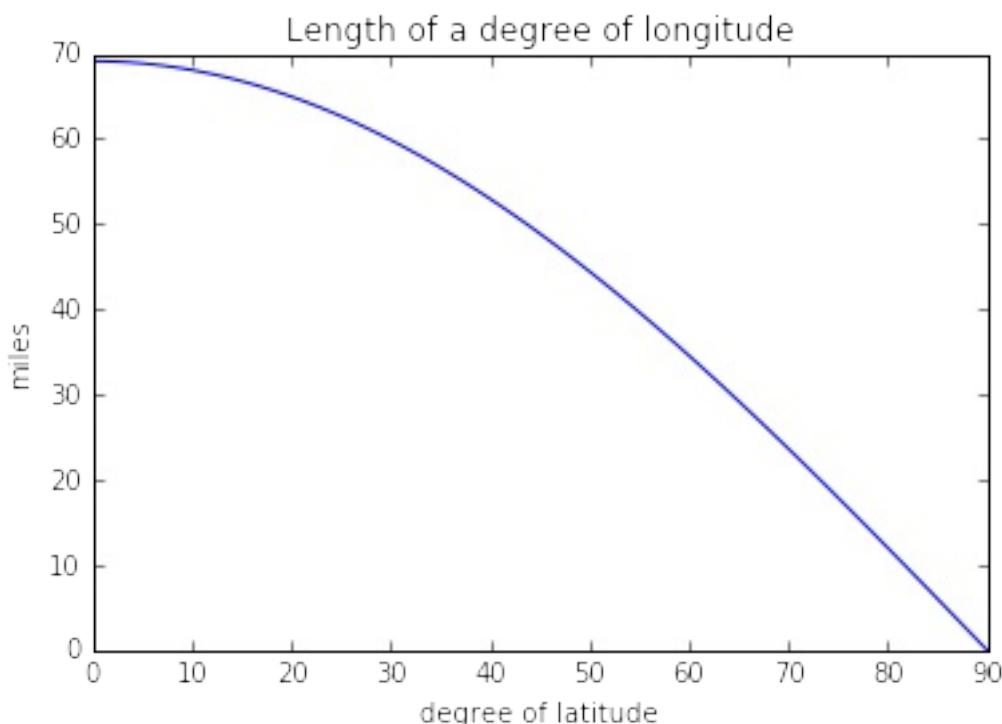
```
loc1 = (-117.1611, 89.0)
loc2 = (-118.1611, 89.0)
gcd(loc1, loc2)
```

```
1.2065130336642724
```

```
lats = range(0, 91)
onedeglon = [ gcd((-117.1611,lat),(-118.1611,lat)) for lat in lats]
```

```
import matplotlib.pyplot as plt
%matplotlib inline
plt.plot(lats, onedeglon)
plt.ylabel('miles')
plt.xlabel('degree of latitude')
plt.title('Length of a degree of longitude')
```

```
<matplotlib.text.Text at 0x7fc5a4abdba8>
```



```
san_diego = -117.1611, 32.7157
austin = -97.7431, 30.2672
gcd(san_diego, austin)
```

```
1155.474644164695
```

Now we can use our distance function to pose distance-related queries on our data table.

```
# Find all the counties with centroids within 50 miles of Austin
def near_target_point(polygon, target=austin, threshold=50):
    return gcd(polygon.centroid, target) < threshold

data_table[data_table.geometry.apply(near_target_point)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIP
2698	Burnet	Texas	48	053	4805
2716	Williamson	Texas	48	491	4849
2742	Travis	Texas	48	453	4845
2751	Lee	Texas	48	287	4828
2754	Blanco	Texas	48	031	4803
2762	Bastrop	Texas	48	021	4802
2769	Hays	Texas	48	209	4820
2795	Caldwell	Texas	48	055	4805
2798	Comal	Texas	48	091	4809
2808	Guadalupe	Texas	48	187	4818

10 rows × 70 columns

Moving in and out of the dataframe

Most things in PySAL will be explicit about what type their input should be. Most of the time, PySAL functions require either lists or arrays. This is why the file-handler methods are the default IO method in PySAL: the rest of the computational tools are built around their datatypes.

However, it is very easy to get the correct datatype from Pandas using the `values` and `tolist` commands.

`tolist()` will convert its entries to a list. But, it can only be called on individual columns (called `series` in `pandas` documentation).

So, to turn the `NAME` column into a list:

```
data_table.NAME.tolist()
```

```
['Lake of the Woods',
 'Ferry',
 'Stevens',
 'Okanogan',
 'Pend Oreille',
 'Boundary',
 'Lincoln',
 'Flathead',
 'Glacier',
 'Toole',
 'Liberty',
 'Hill',
 'Sheridan',
 'Divide',
 'Burke',
 'Renville',
 'Bottineau',
 'Rolette',
 'Towner',
 'Cavalier',
 'Pembina',
 'Kittson',
 'Roseau',
 'Blaine',
 'Phillips',
 'Valley',
 'Daniels',
 'Whatcom',
 'Bonner',
 'Ward',
 'Koochiching',
 'Skagit',
 'Williams',
 'McHenry',
 'St. Louis',
 'Roosevelt',
 'Mountrial',
 'Marshall',
 'Ramsey',
 'Walsh',
 'Beltrami',
 'Pierce',
 'Chelan',
 'Pondera',
 'Clallam',
 'Benson',
 'Chouteau',
```

```
'Snohomish',
'Island',
'Sanders',
'Lake',
'Nelson',
'Grand Forks',
'Polk',
'Pennington',
'Douglas',
'McKenzie',
'Jefferson',
'Richland',
'Teton',
'McCone',
'Shoshone',
'Spokane',
'Lake',
'Clearwater',
'Kootenai',
'Garfield',
'Red Lake',
'Grant',
'Lincoln',
'Lewis and Clark',
'Kitsap',
'Itasca',
'Sheridan',
'Wells',
'McLean',
'Eddy',
'Dunn',
'Fergus',
'Dawson',
'King',
'Cascade',
'Griggs',
'Stelle',
'Traill',
'Mason',
'Missoula',
'Petroleum',
'Powell',
'Kittitas',
'Foster',
'Mercer',
'Grays Harbor',
'Norman',
'Mahnomen',
'Mineral',
'Cass',
'Aroostook',
'Judith Basin',
```

```
'Hubbard',
'Benewah',
'Wibaux',
'Golden Valley',
'Billings',
'Stutsman',
'Kidder',
'Burleigh',
'Pierce',
'Oliver',
'Adams',
'Whitman',
'Barnes',
'Cass',
'Prairie',
'Becker',
'Clay',
'Thurston',
'Latah',
'Meagher',
'Yakima',
'Aitkin',
'Stark',
'Morton',
'Bayfield',
'Clearwater',
'Custer',
'Rosebud',
'Granite',
'Wadena',
'Crow Wing',
'Pacific',
'Lewis',
'Broadwater',
'Carlton',
'Golden Valley',
'Douglas',
'Musselshell',
'Wheatland',
'Franklin',
'Benton',
'Grant',
'Otter Tail',
'Garfield',
'Fallon',
'Idaho',
'Ravalli',
'Ashland',
'Logan',
'Emmons',
'La Moure',
'Slope',
```

```
'Hettinger',
'Ransom',
'Richland',
'Wilkin',
'Nez Perce',
'Columbia',
'Walla Walla',
'Iron',
'Somerset',
'Piscataquis',
'Jefferson',
'Yellowstone',
'Treasure',
'Lewis',
'Asotin',
'Sioux',
'Pine',
'Penobscot',
'Skamania',
'Cowlitz',
'Wahkiakum',
'Todd',
'Morrison',
'McIntosh',
'Dickey',
'Sargent',
'Bowman',
'Adams',
'Deer Lodge',
'Mille Lacs',
'Clatsop',
'Sweet Grass',
'Columbia',
'Silver Bow',
'Washburn',
'Sawyer',
'Burnett',
'Kanabec',
'Carter',
'Stillwater',
'Grant',
'Douglas',
'Clark',
'Klickitat',
'Big Horn',
'Traverse',
'Umatilla',
'Wallowa',
'Price',
'Campbell',
'Harding',
'McPherson',
```

```
'Perkins',
'Corson',
'Brown',
'Beaverhead',
'Marshall',
'Roberts',
'Morrow',
'Union',
'Madison',
'Benton',
'Gilliam',
'Powder River',
'Stearns',
'Tillamook',
'Washington',
'Pope',
'Stevens',
'Isanti',
'Chisago',
'Sherman',
'Polk',
'Multnomah',
'Hood River',
'Wasco',
'Lemhi',
'Washington',
'Franklin',
'Barron',
'Rusk',
'Carbon',
'Walworth',
'Edmunds',
'Day',
'Big Stone',
'Sherburne',
'Ziebach',
'Dewey',
'Clackamas',
'Wright',
'Yamhill',
'Anoka',
'Kandiyohi',
'Swift',
'Taylor',
'Oxford',
'Grant',
'Meeker',
'Coos',
'Washington',
'Chippewa',
'Marion',
'Lac Qui Parle',
```

```
'Adams',
'Potter',
'Faulk',
'Hennepin',
'Spink',
'St. Croix',
'Dunn',
'Butte',
'Valley',
'Clark',
'Codington',
'Chippewa',
'Ramsey',
'Baker',
'Polk',
'Wheeler',
'Meade',
'Lincoln',
'Clark',
'Essex',
'Grand Isle',
'Franklin',
'Orleans',
'Clinton',
'Park',
'Crook',
'Big Horn',
'Campbell',
'Sheridan',
'Franklin',
'Grant',
'McLeod',
'Carver',
'Deuel',
'Dakota',
'Yellow Medicine',
'Sully',
'Hyde',
'Hand',
'Renville',
'Pierce',
'Custer',
'Eau Claire',
'Washington',
'Jefferson',
'Scott',
'Hamlin',
'Lamoille',
'Linn',
'Stanley',
'Caledonia',
'Waldo',
```

```
'Haakon',
'Fremont',
'Sibley',
'Chittenden',
'Kennebec',
'Goodhue',
'Benton',
'Redwood',
'Wood',
'Pepin',
'Teton',
'Beadle',
'Lyon',
'Lincoln',
'Lawrence',
'Buffalo',
'Trempealeau',
'Jackson',
'Clark',
'Crook',
'Johnson',
'Hughes',
'Essex',
'Le Sueur',
'Rice',
'Kingsbury',
'Brookings',
'Pennington',
'Gem',
'Brown',
'Washington',
'Androscoggin',
'Nicollet',
'Wabasha',
'Malheur',
'Hancock',
'Grafton',
'Deschutes',
'Knox',
'Boise',
'Lincoln',
'Addison',
'Carroll',
'Lane',
'Blue Earth',
'Juneau',
'Butte',
'Lyman',
'Orange',
'Waseca',
'Buffalo',
'Jerauld',
```

```
'Moody',
'Pipestone',
'Dodge',
'Sanborn',
'Murray',
'Cottonwood',
'Stelle',
'Olmsted',
'Miner',
'Lake',
'Winona',
'Weston',
'Jones',
'Cumberland',
'Washakie',
'Monroe',
'Payette',
'Hamilton',
'Watowan',
'Herkimer',
'La Crosse',
'Elmore',
'Hot Springs',
'Jefferson',
'Harney',
'Sagadahoc',
'Fremont',
'Jackson',
'Blaine',
'Teton',
'Windsor',
'Douglas',
'Aurora',
'Brule',
'Madison',
'Canyon',
'Mellette',
'Camas',
'Custer',
'Rutland',
'Faribault',
'Minnehaha',
'Rock',
'Freeborn',
'Nobles',
'Jackson',
'Martin',
'Houston',
'Mower',
'Fillmore',
'Davison',
'Hanson',
```

```
'McCook',
'York',
'Washington',
'Ada',
'Warren',
'Tripp',
'Belknap',
'Vernon',
'Shannon',
'Owyhee',
'Bonneville',
'Bingham',
'Klamath',
'Lake',
'Coos',
'Merrimack',
'Sullivan',
'Strafford',
'Richland',
'Natrona',
'Gregory',
'Niobrara',
'Charles Mix',
'Turner',
'Lincoln',
'Worth',
'Mitchell',
'Allamakee',
'Winnebago',
'Winneshiek',
'Converse',
'Osceola',
'Dickinson',
'Kossuth',
'Howard',
'Emmet',
'Lyon',
'Douglas',
'Hutchinson',
'Fall River',
'Sublette',
'Crawford',
'Saratoga',
'Bennett',
'Todd',
'Bennington',
'Lincoln',
'Fulton',
'Rockingham',
'Sioux',
'Windham',
"O'Brien",
```

```
'Cerro Gordo',
'Clay',
'Hancock',
'Palo Alto',
'Floyd',
'Chickasaw',
'Iowa',
'Grant',
'Hillsborough',
'Lincoln',
'Gooding',
'Minidoka',
'Cheshire',
'Yankton',
'Bon Homme',
'Power',
'Union',
'Clay',
'Fayette',
'Clayton',
'Montgomery',
'Caribou',
'Bannock',
'Sioux',
'Dawes',
'Sheridan',
'Jackson',
'Keya Paha',
'Boyd',
'Cherry',
'Curry',
'Rensselaer',
'Schenectady',
'Plymouth',
'Cherokee',
'Bremer',
'Butler',
'Buena Vista',
'Twin Falls',
'Pocahontas',
'Humboldt',
'Wright',
'Franklin',
'Otsego',
'Holt',
'Essex',
'Knox',
'Cedar',
'Jerome',
'Schoharie',
'Brown',
'Lafayette',
```

```
'Albany',
'Rock',
'Josephine',
'Dixon',
'Berkshire',
'Franklin',
'Middlesex',
'Worcester',
'Dubuque',
'Cassia',
'Webster',
'Delaware',
'Buchanan',
'Black Hawk',
'Goshen',
'Platte',
'Bear Lake',
'Woodbury',
'Ida',
'Sac',
'Calhoun',
'Hamilton',
'Hampshire',
'Hardin',
'Grundy',
'Dakota',
'Delaware',
'Jo Daviess',
'Columbia',
'Oneida',
'Greene',
'Suffolk',
'Carbon',
'Pierce',
'Box Butte',
'Antelope',
'Albany',
'Franklin',
'Jackson',
'Wayne',
'Hampden',
'Jones',
'Benton',
'Linn',
'Tama',
'Thorston',
'Sweetwater',
'Plymouth',
'Norfolk',
'Monona',
'Crawford',
'Carroll',
```

```
'Greene',
'Boone',
'Marshall',
'Story',
'Ulster',
'Cuming',
'Bristol',
'Stanton',
'Madison',
'Grant',
'Loup',
'Hooker',
'Garfield',
'Thomas',
'Wheeler',
'Blaine',
'Dutchess',
'Barnstable',
'Burt',
'Litchfield',
'Hartford',
'Clinton',
'Tolland',
'Windham',
'Sullivan',
'Providence',
'Cache',
'Siskiyou',
'Garden',
'Box Elder',
'Rich',
'Scotts Bluff',
'Morrill',
'Wayne',
'Del Norte',
'Humboldt',
'Elko',
'Modoc',
'Washoe',
'Cedar',
'Boone',
'Jasper',
'Polk',
'Poweshiek',
'Harrison',
'Guthrie',
'Shelby',
'Audubon',
'Iowa',
'Dallas',
'Johnson',
'Rock Island',
```

```
'Scott',
'Bristol',
'Kent',
'Colfax',
'Dodge',
'Platte',
'Arthur',
'Greeley',
'McPherson',
'Logan',
'Custer',
'Valley',
'Will',
'Lucas',
'Kendall',
'Lake',
'Porter',
'Fulton',
'Geauga',
'New London',
'Williams',
'Banner',
'Washington',
'Newport',
'Fairfield',
'Laramie',
'Wyoming',
'Washington',
'Middlesex',
'New Haven',
'Lackawanna',
'La Salle',
'Orange',
'Elk',
'Cuyahoga',
'Venango',
'Forest',
'Ottawa',
'Cameron',
'Wood',
'Pike',
'Lycoming',
'Muscatine',
'Sullivan',
'Bureau',
'Henry',
'Uinta',
'Noble',
'De Kalb',
'Putnam',
'Nance',
'Lorain',
```

```
'Mahaska',
'Pottawattamie',
'Washington',
'Marion',
'Madison',
'Warren',
'Keokuk',
'Cass',
'Adair',
'Sandusky',
'Trumbull',
'Mercer',
'Marshall',
'Henry',
'Clinton',
'Grundy',
'Humboldt',
'Butler',
'Saunders',
'Erie',
'Kosciusko',
'Starke',
'Clarion',
'Cheyenne',
'Defiance',
'Weber',
'Louisa',
'Luzerne',
'Polk',
'Douglas',
'Sherman',
'Howard',
'Lincoln',
'Merrick',
'Keith',
'Kimball',
'Jefferson',
'Morgan',
'Trinity',
'Westchester',
'Sussex',
'Summit',
'Portage',
'Mercer',
'Rockland',
'Putnam',
'Columbia',
'Kankakee',
'Whitley',
'Jasper',
'Huron',
'Allen',
```

```
'Medina',
'Summit',
'Centre',
'Clearfield',
'Monroe',
'Seneca',
'Paulding',
'Stark',
'Newton',
'Deuel',
'Passaic',
'Sarpy',
'Shasta',
'Lassen',
'Northumberland',
'Fulton',
'Butler',
'Armstrong',
'Pulaski',
'Montour',
'Hancock',
'Mills',
'Putnam',
'Montgomery',
'Adams',
'Clarke',
'Wapello',
'Jefferson',
'Union',
'Henry',
'Hamilton',
'Lucas',
'Monroe',
'Davis',
'Knox',
'Marshall',
'Union',
'Carbon',
'Mahoning',
'Lawrence',
'Bergen',
'Livingston',
'Warren',
'Tooele',
'Morris',
'Des Moines',
'Henderson',
'Warren',
'Cass',
'Ashland',
'Wabash',
'Seward',
```

```
'Lancaster',
'Buffalo',
'Dawson',
'York',
'Hall',
'Huntington',
'Iroquois',
'Miami',
'Ford',
'Moffat',
'Weld',
'Jackson',
'Perkins',
'Logan',
'Sedgwick',
'Routt',
'Larimer',
'Daggett',
'Crawford',
'Lander',
'Eureka',
'Richland',
'Wayne',
'Van Wert',
'Wyandot',
'Stark',
'Peoria',
'Northampton',
'Pershing',
'Schuylkill',
'Adams',
'Wells',
'Woodford',
'Columbian',
'Cass',
'White',
'Salt Lake',
'Allen',
'Indiana',
'Fremont',
'Page',
'Taylor',
'Ringgold',
'Nassau',
'Davis',
'Van Buren',
'Decatur',
'Wayne',
'Essex',
'Appanoose',
'Snyder',
'Uintah',
```

```
'Beaver',
'Mifflin',
'Duchesne',
'Hudson',
'Lee',
'Hardin',
'Otoe',
'Lehigh',
'Hunterdon',
'Suffolk',
'McLean',
'Somerset',
'Carroll',
'Tazewell',
'Blair',
'Benton',
'Phillips',
'Huntingdon',
'Cambria',
'Union',
'Mercer',
'Carroll',
'Fulton',
'Morrow',
'Marion',
'Saline',
'Adams',
'Clay',
'Fillmore',
'Juniata',
'Hayes',
'Chase',
'Frontier',
'Gosper',
'Auglaize',
'Kearney',
'Wasatch',
'Phelps',
'Berks',
'Westmoreland',
'Allegheny',
'Grant',
'Holmes',
'Dauphin',
'Tuscarawas',
'Hancock',
'McDonough',
'Perry',
'Hancock',
'Bucks',
'Clark',
'Scotland',
```

```
'Schuyler',
'Middlesex',
'Jefferson',
'Blackford',
'Putnam',
'Atchison',
'Jay',
'Utah',
'Nodaway',
'Mercer',
'Howard',
'Harrison',
'Tippecanoe',
'Worth',
'Knox',
'Nemaha',
'Lebanon',
'Logan',
'Gage',
'Johnson',
'Morgan',
'Union',
'Vermilion',
'Warren',
'Shelby',
'Washington',
'Grand',
'Coshocton',
'Tehama',
'Monmouth',
'Plumas',
'Delaware',
'Montgomery',
'Mason',
'Clinton',
'Yuma',
'Washington',
'Harrison',
'Mercer',
'Tipton',
'Champaign',
'Brooke',
'Madison',
'Delaware',
'Gentry',
'Sullivan',
'Fountain',
'Darke',
'Thayer',
'Jefferson',
'Adair',
'Dundy',
```

```
'Franklin',
'Webster',
'Nuckolls',
'Hitchcock',
'Harlan',
'Furnas',
'Red Willow',
'Logan',
'Bedford',
'Cumberland',
'Randolph',
'Lancaster',
'Knox',
'Franklin',
'Piatt',
'De Witt',
'Somerset',
'Schuylerville',
'Licking',
'Champaign',
'Holt',
'Richardson',
'Pawnee',
'Grundy',
'Boulder',
'Lewis',
'Chester',
'Hamilton',
'York',
'Montgomery',
'Rio Blanco',
'Guernsey',
'Adams',
'Miami',
'Ohio',
'Boone',
'Burlington',
'Vermillion',
'Menard',
'Belmont',
'Ocean',
'Fulton',
'Muskingum',
'Butte',
'Daviess',
'Franklin',
'Fayette',
'Philadelphia',
'Andrew',
'Cass',
'White Pine',
'Madison',
```

```
'Brown',
'Garfield',
'Henry',
'Adams',
'Delaware',
'Macon',
'De Kalb',
'Macon',
'Clark',
'Linn',
'Marshall',
'Greene',
'Wayne',
'Juab',
'Churchill',
'Norton',
'Phillips',
...]
```

To extract many columns, you must select the columns you want and call their `.values` attribute.

If we were interested in grabbing all of the `HR` variables in the dataframe, we could first select those column names:

```
HRs = [col for col in data_table.columns if col.startswith('HR')]

HRs
```

```
['HR60', 'HR70', 'HR80', 'HR90']
```

We can use this to focus only on the columns we want:

```
data_table[HRs]
```

	HR60	HR70	HR80	HR90
0	0.000000	0.000000	8.855827	0.000000
1	0.000000	0.000000	17.208742	15.885624
2	1.863863	1.915158	3.450775	6.462453
3	2.612330	1.288643	3.263814	6.996502
4	0.000000	0.000000	7.770008	7.478033

5	0.000000	0.000000	4.573101	4.000640
6	7.976390	5.536179	5.633168	5.720497
7	1.011173	1.689475	4.490115	2.814460
8	11.529039	9.273857	28.227324	5.500096
9	0.000000	5.708740	0.000000	6.605892
10	0.000000	0.000000	0.000000	0.000000
11	3.574045	3.840688	7.413585	1.888146
12	0.000000	0.000000	0.000000	0.000000
13	0.000000	0.000000	0.000000	0.000000
14	0.000000	0.000000	0.000000	0.000000
15	0.000000	0.000000	0.000000	0.000000
16	2.945942	0.000000	0.000000	0.000000
17	0.000000	0.000000	19.161808	5.219752
18	0.000000	0.000000	0.000000	0.000000
19	0.000000	8.117213	0.000000	0.000000
20	0.000000	0.000000	0.000000	0.000000
21	0.000000	4.864050	0.000000	0.000000
22	0.000000	0.000000	0.000000	2.218377
23	4.119804	9.910312	14.287755	14.863258
24	5.530668	0.000000	12.421589	0.000000
25	5.854801	5.811757	6.504065	4.045798
26	0.000000	10.811980	0.000000	0.000000
27	1.422131	2.439530	4.061193	2.086920
28	2.138534	2.142245	5.518079	3.756292
29	0.708135	1.138434	0.570854	1.150993
...
3055	0.000000	0.000000	2.107526	0.739919

3056	0.000000	0.970572	4.400324	0.825491
3057	0.611430	2.568301	1.974919	2.828994
3058	2.022286	2.033140	0.000000	3.987956
3059	0.850723	1.981012	2.782690	2.260130
3060	1.790825	3.732470	5.757329	4.007173
3061	0.556604	1.060271	1.010560	2.769193
3062	0.000000	1.756836	5.505395	2.907653
3063	0.427592	3.688132	2.625587	0.000000
3064	0.896660	2.704530	4.229303	2.938915
3065	1.051403	5.379304	7.057466	3.424679
3066	2.095434	6.671377	9.243279	7.285916
3067	1.872835	3.951663	5.339935	3.201065
3068	1.917913	6.382639	1.304631	0.000000
3069	1.939789	4.960564	0.000000	6.072530
3070	5.452765	15.156192	24.841012	28.268787
3071	7.520089	9.163383	10.590286	6.443839
3072	7.202448	9.746302	11.850014	12.561604
3073	8.253379	15.655752	21.173432	16.479507
3074	2.181802	3.074760	3.191133	3.300700
3075	4.902862	11.782264	7.680787	18.362582
3076	18.513376	17.133324	15.034136	12.027015
3077	4.159907	4.126434	3.967782	6.585273
3078	5.403098	5.970974	4.127839	2.586787
3079	1.121183	1.096311	2.442074	2.806112
3080	5.046682	13.152054	13.251761	5.521552
3081	3.411368	7.393533	11.453817	8.691999
3082	1.544425	6.023552	5.280349	4.367330

3083	9.302820	1.800148	3.000030	3.727712
3084	3.396162	2.284879	1.194743	2.048855

3085 rows × 4 columns

With this, calling `.values` gives an array containing all of the entries in this subset of the table:

```
data_table[['HR90', 'HR80']].values
```

```
array([[ 0.          ,  8.85582713],
       [ 15.88562351, 17.20874204],
       [ 6.46245315,  3.4507747 ],
       ...,
       [ 4.36732988,  5.2803488 ],
       [ 3.72771194,  3.00003   ],
       [ 2.04885495,  1.19474313]])
```

Using the PySAL pdio tools means that if you're comfortable with working in Pandas, you can continue to do so.

If you're more comfortable using Numpy or raw Python to do your data processing, PySAL's IO tools naturally support this.

Exercises

1. Find the county with the western most centroid that is within 1000 miles of Austin.
2. Find the distance between Austin and that centroid.



1. Solution

```
# Solution 1
import scipy.spatial.distance as d
def near_target_point(polygon, target=austin, threshold=1000):
    return gcd(polygon.centroid, target) < threshold

w1000=data_table[data_table.geometry.apply(near_target_point)]
```

```
len(w1000)
```

```
1883
```

```
g = w1000.geometry
```

```
type(g)
```

```
pandas.core.series.Series
```

```
len(g)
```

```
1883
```

```
gc = g.copy()
```

```
coords = np.array([ p.centroid for p in gc.values])
```

```
coords.min(axis=0)
```

```
array([-113.93953516, 26.13240654])
```

```
west_lon = np.array([ p.centroid for p in gc.values]).min(axis=0)[0]
```

```
west_lon
```

```
-113.93953516273253
```

```
# one liner
west_lon = np.array([p.centroid for p in w1000.geometry.values]).min(axis=0)[0]
west_lon
```

```
-113.93953516273253
```

```
w1000[w1000.geometry.apply(lambda x: x.centroid[0] == west_lon)]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
3080	La Paz	Arizona	04	012	04012

1 rows × 70 columns

2. Solution

```
lapaz = w1000[w1000.NAME=="La Paz"].geometry.values[0].centroid
```

```
gcd(lapaz, austin)
```

```
972.5962735038516
```

Choropleth Mapping

IPYNB

Introduction

When PySAL was originally planned, the intention was to focus on the computational aspects of exploratory spatial data analysis and spatial econometric methods, while relying on existing GIS packages and visualization libraries for visualization of computations. Indeed, we have partnered with [esri](#) and [QGIS](#) towards this end.

However, over time we have received many requests for supporting basic geovisualization within PySAL so that the step of having to interoperate with an external package can be avoided, thereby increasing the efficiency of the spatial analytical workflow.

In this notebook, we demonstrate several approaches towards a particular subset of geovisualization methods, namely **choropleth maps**. We start with a self-contained exploratory workflow where no other dependencies beyond PySAL are required. The idea here is to support quick generation of different views of your data to complement the statistical and econometric work in PySAL. Once your work has progressed to the publication stage, we point you to resources that can be used for publication quality output.

We then move on to consider three other packages that can be used in conjunction with PySAL for choropleth mapping:

- geopandas
- folium
- cartopy
- bokeh

PySAL Viz Module

The mapping module in PySAL is organized around three main layers:

- A lower-level layer that reads polygon, line and point shapefiles and returns a Matplotlib collection.
- A medium-level layer that performs some usual transformations on a Matplotlib object (e.g.

color code polygons according to a vector of values).

- A higher-level layer intended for end-users for particularly useful cases and style preferences pre-defined (e.g. Create a choropleth).

```
%matplotlib inline
import numpy as np
import pysal as ps
import random as rdm
from pysal.contrib.viz import mapping as maps
from pylab import *
```

Lower-level component

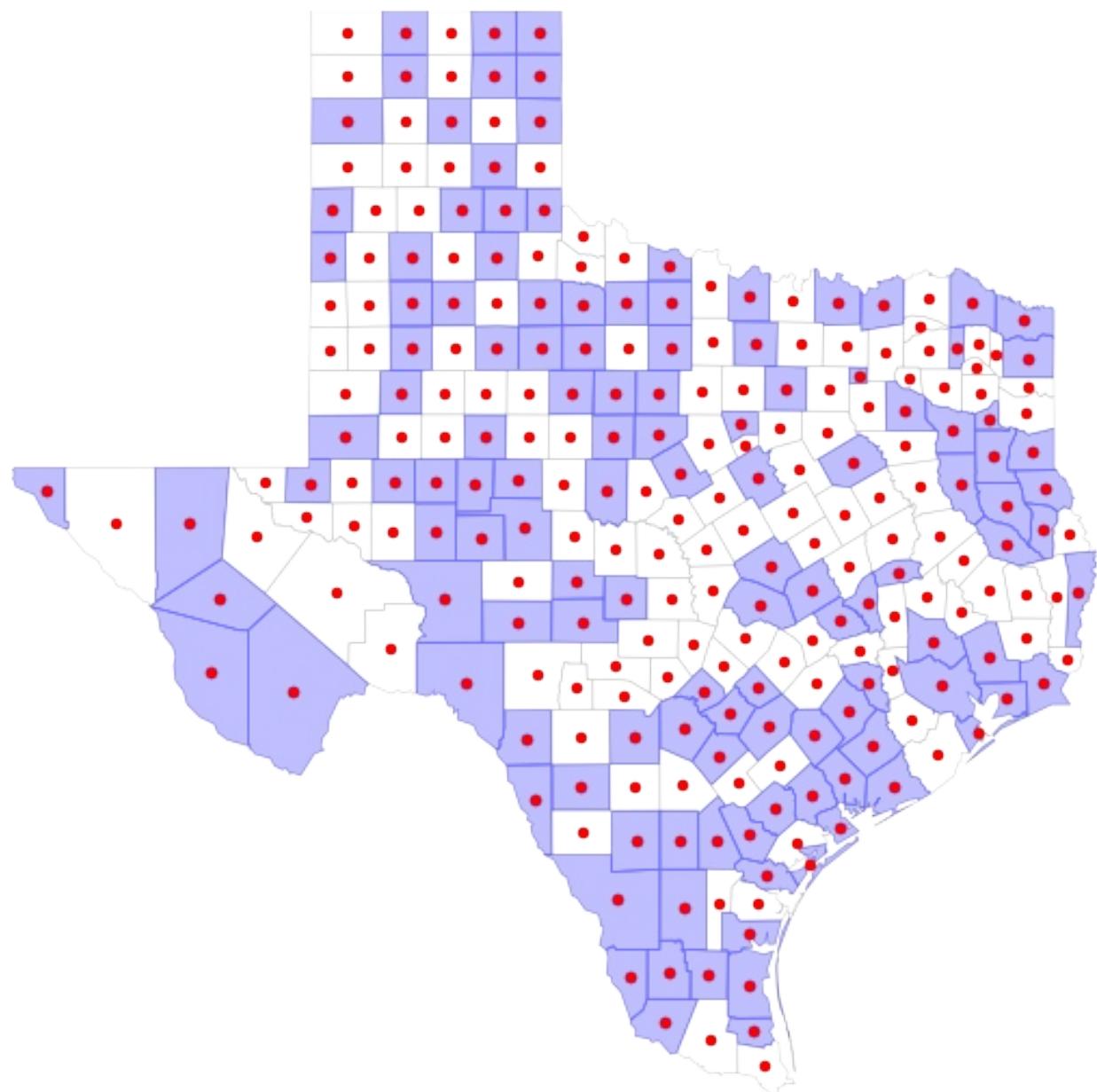
This includes basic functionality to read spatial data from a file (currently only shapefiles supported) and produce rudimentary Matplotlib objects. The main methods are:

- map_poly_shape: to read in polygon shapefiles
- map_line_shape: to read in line shapefiles
- map_point_shape: to read in point shapefiles

These methods all support an option to subset the observations to be plotted (very useful when missing values are present). They can also be overlaid and combined by using the `setup_ax` function. the resulting object is very basic but also very flexible so, for minds used to matplotlib this should be good news as it allows to modify pretty much any property and attribute.

Example

```
shp_link = '../data/texas.shp'
shp = ps.open(shp_link)
some = [bool(rdm.getrandbits(1)) for i in ps.open(shp_link)]  
  
fig = figure(figsize=(9,9))  
  
base = maps.map_poly_shp(shp)
base.set_facecolor('none')
base.set_linewidth(0.75)
base.set_edgecolor('0.8')
some = maps.map_poly_shp(shp, which=some)
some.set_alpha(0.5)
some.set_linewidth(0.)
cents = np.array([poly.centroid for poly in ps.open(shp_link)])
pts = scatter(cents[:, 0], cents[:, 1])
pts.set_color('red')  
  
ax = maps.setup_ax([base, some, pts], [shp.bbox, shp.bbox, shp.bbox])
fig.add_axes(ax)
show()
```



Medium-level component

This layer comprises functions that perform usual transformations on matplotlib objects, such as color coding objects (points, polygons, etc.) according to a series of values. This includes the following methods:

- `base_choropleth_classless`
- `base_choropleth_unique`
- `base_choropleth_classif`

Example

```
net_link = ps.examples.get_path('eberly_net.shp')
net = ps.open(net_link)
values = np.array(ps.open(net_link.replace('.shp', '.dbf')).by_col('TNODE'))

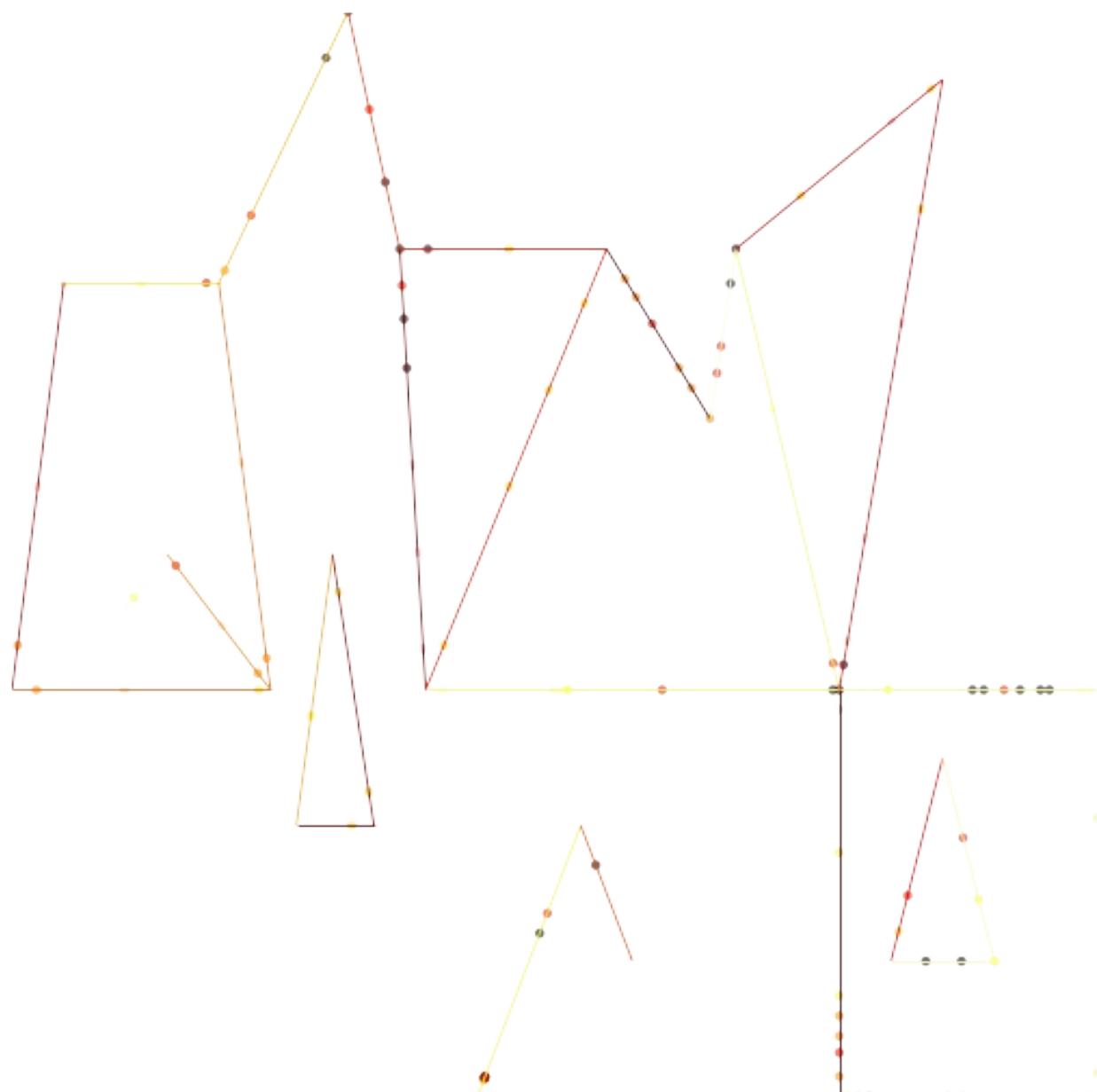
pts_link = ps.examples.get_path('eberly_net_pts_onnetwork.shp')
pts = ps.open(pts_link)

fig = figure(figsize=(9,9))

netm = maps.map_line_shp(net)
netc = maps.base_choropleth_unique(netm, values)

ptsm = maps.map_point_shp(pts)
ptsm = maps.base_choropleth_classif(ptsm, values)
ptsm.set_alpha(0.5)
ptsm.set_linewidth(0.)

ax = maps.setup_ax([netc, ptsm], [net.bbox, net.bbox])
fig.add_axes(ax)
show()
```



```
maps.plot_poly_lines('.../data/texas.shp')
```

```
calling plt.show()
```



Higher-level component

This currently includes the following end-user functions:

- `plot_poly_lines` : very quick shapfile plotting

```
shp_link = '../data/texas.shp'
values = np.array(ps.open('../data/texas.dbf').by_col('HR90'))

types = ['classless', 'unique_values', 'quantiles', 'equal_interval', 'fisher_jenkins']
for typ in types:
    maps.plot_choropleth(shp_link, values, typ, title=typ)
```

classless



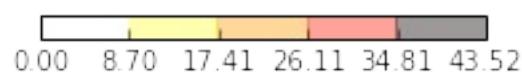
unique_values

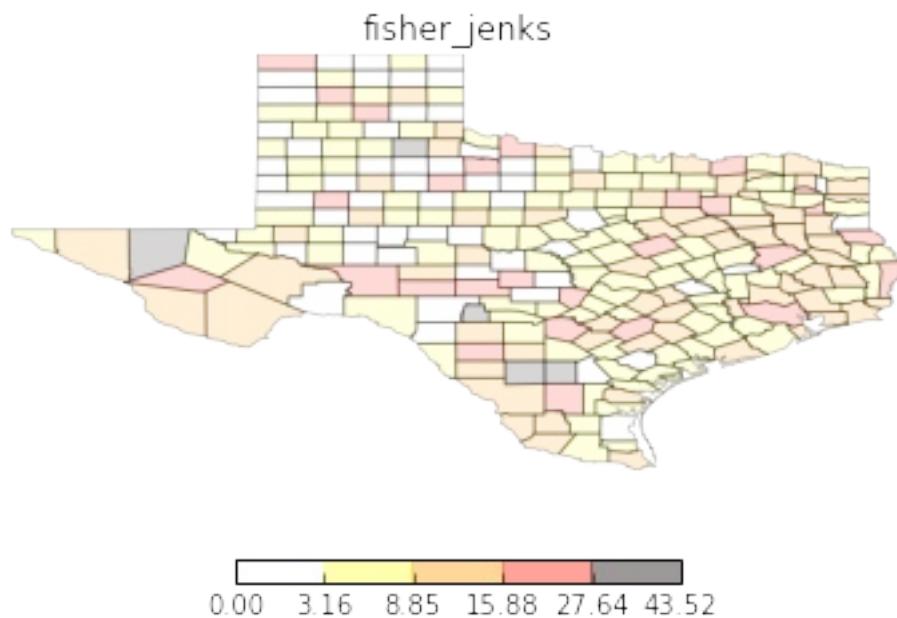


quantiles



equal_interval





PySAL Map Classifiers

```
hr90 = values
hr90q5 = ps.Quantiles(hr90, k=5)
hr90q5
```

Quantiles

Lower	Upper	Count
<hr/>		
x[i] <= 2.421		51
2.421 < x[i] <= 5.652		51
5.652 < x[i] <= 8.510		50
8.510 < x[i] <= 12.571		51
12.571 < x[i] <= 43.516		51

```
hr90q4 = ps.Quantiles(hr90, k=4)
hr90q4
```

Quantiles

Lower	Upper	Count
<hr/>		
x[i] <= 3.918		64
3.918 < x[i] <= 7.232		63
7.232 < x[i] <= 11.414		63
11.414 < x[i] <= 43.516		64

```
hr90e5 = ps.Equal_Interval(hr90, k=5)
hr90e5
```

Equal Interval

Lower	Upper	Count
<hr/>		
x[i] <= 8.703		157
8.703 < x[i] <= 17.406		76
17.406 < x[i] <= 26.110		16
26.110 < x[i] <= 34.813		2
34.813 < x[i] <= 43.516		3

```
hr90fj5 = ps.Fisher_Jenks(hr90, k=5)
hr90fj5
```

Fisher_Jenks

Lower	Upper	Count
<hr/>		
x[i] <= 3.156		55
3.156 < x[i] <= 8.846		104
8.846 < x[i] <= 15.881		64
15.881 < x[i] <= 27.640		27
27.640 < x[i] <= 43.516		4

```
hr90fj5.adcm # measure of fit: Absolute deviation around class means
```

352.10763138100003

hr90q5.adcm

361.5413784392

hr90e5.adcm

614.51093704210064

```
hr90fj5.yb[0:10] # what bin each value is placed in
```

```
array([0, 0, 3, 0, 1, 0, 0, 0, 0, 1])
```

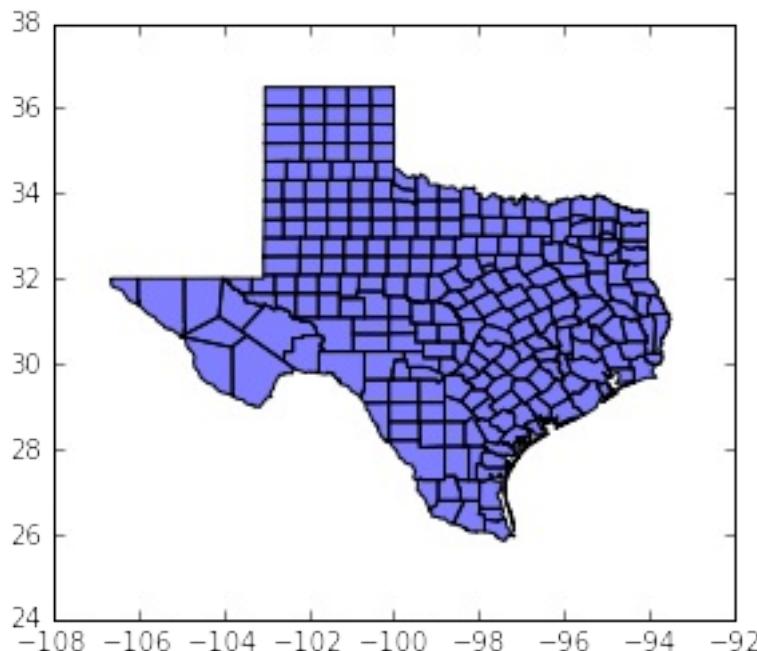
```
hr90fj5.bins # upper bounds of each bin
```

```
array([ 3.15613527,  8.84642604, 15.88088069, 27.63957988, 43.51610096])
```

GeoPandas

```
import geopandas as gpd
shp_link = "../data/texas.shp"
tx = gpd.read_file(shp_link)
tx.plot(color='blue')
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aac19e470>
```

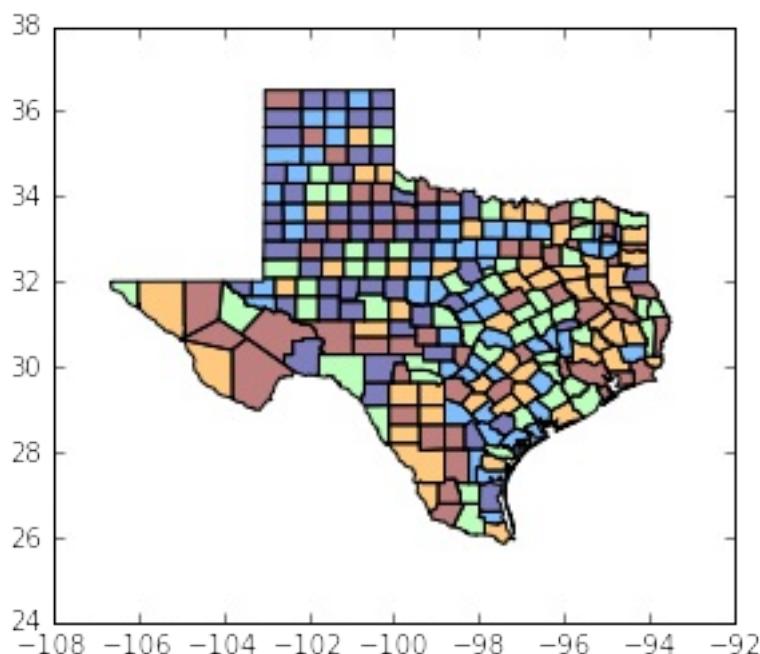


```
type(tx)
```

```
geopandas.geodataframe.GeoDataFrame
```

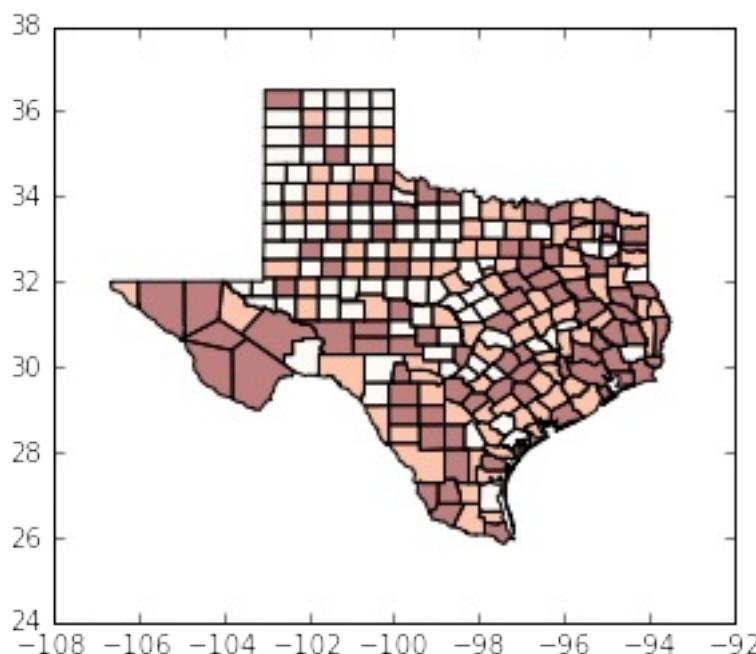
```
tx.plot(column='HR90', scheme='QUANTILES') # uses pysal classifier under the hood
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa341ac18>
```



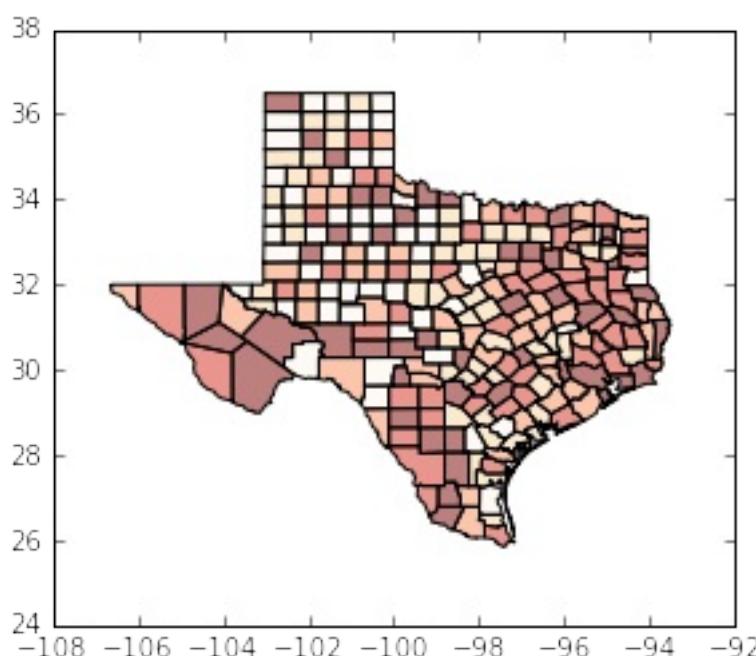
```
tx.plot(column='HR90', scheme='QUANTILES', k=3, cmap='OrRd') # we need a continuous color map
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa2a91128>
```



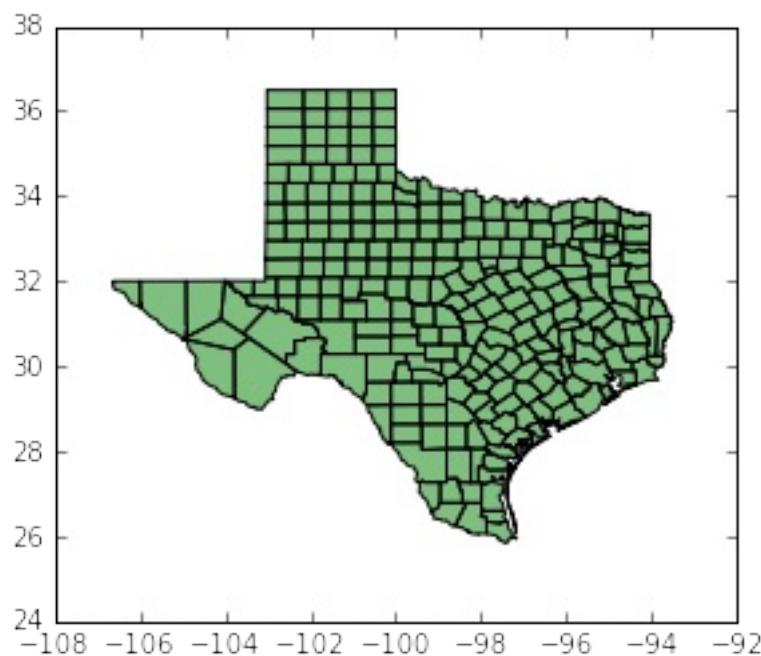
```
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd') # bump up to quintiles
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa2472c88>
```



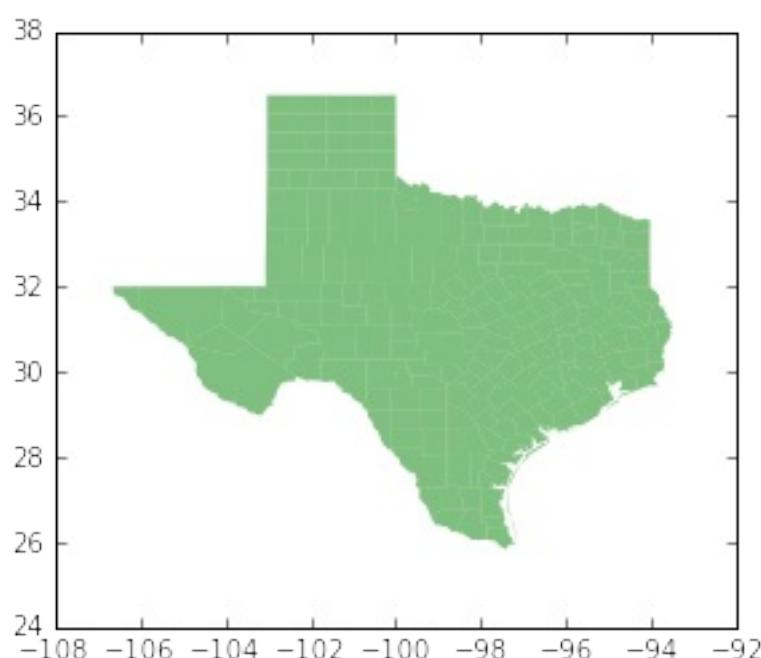
```
tx.plot(color='green') # explore options, polygon fills
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa1dd50f0>
```



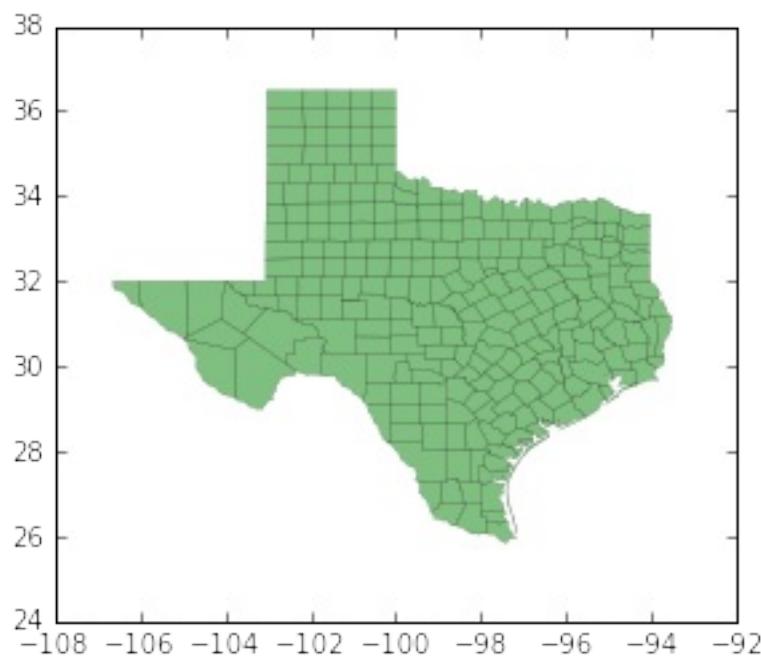
```
tx.plot(color='green', linewidth=0) # border
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa179dc88>
```



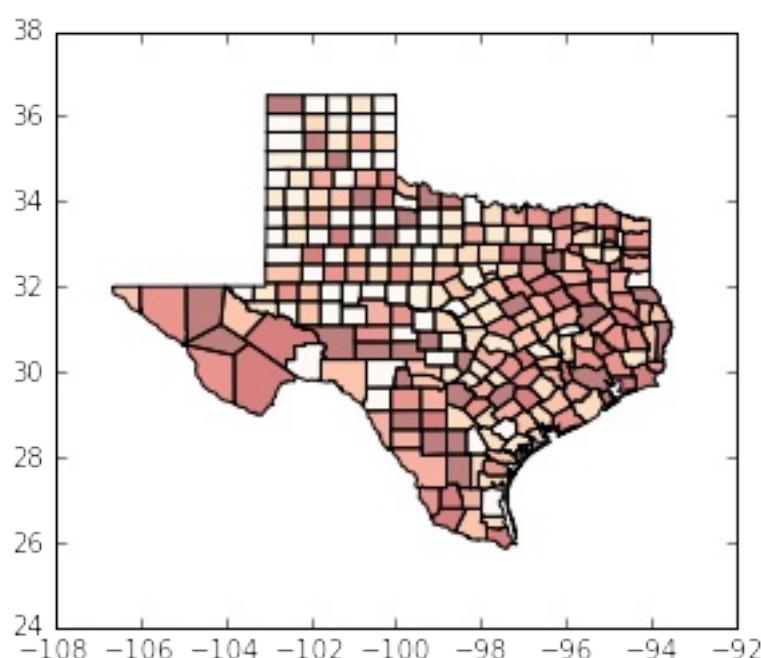
```
tx.plot(color='green', linewidth=0.1) # border
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa2468208>
```



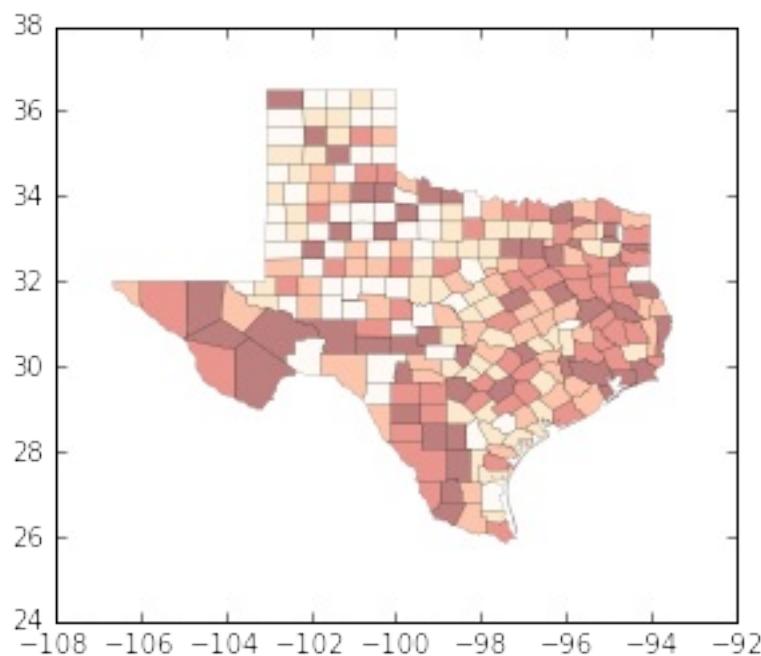
```
tx.plot(column='HR90', scheme='QUANTILES', k=9, cmap='OrRd') # now with quantiles
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa0b4e6a0>
```



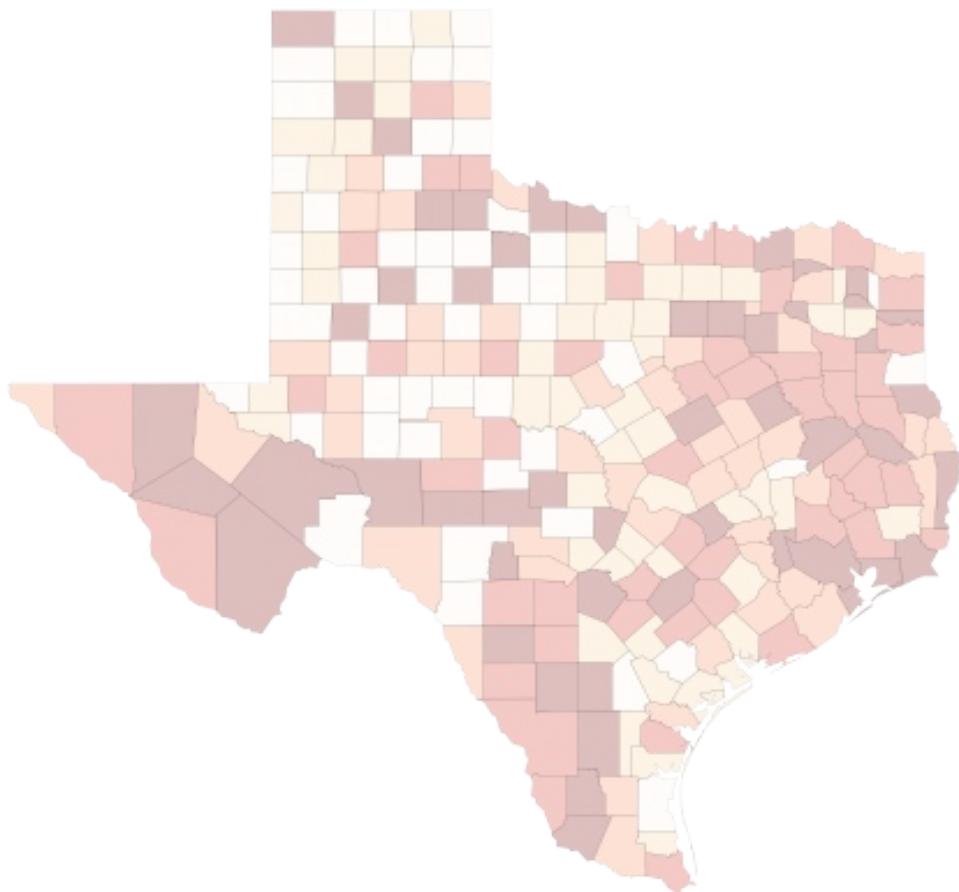
```
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd', linewidth=0.1)
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6aa04b2358>
```

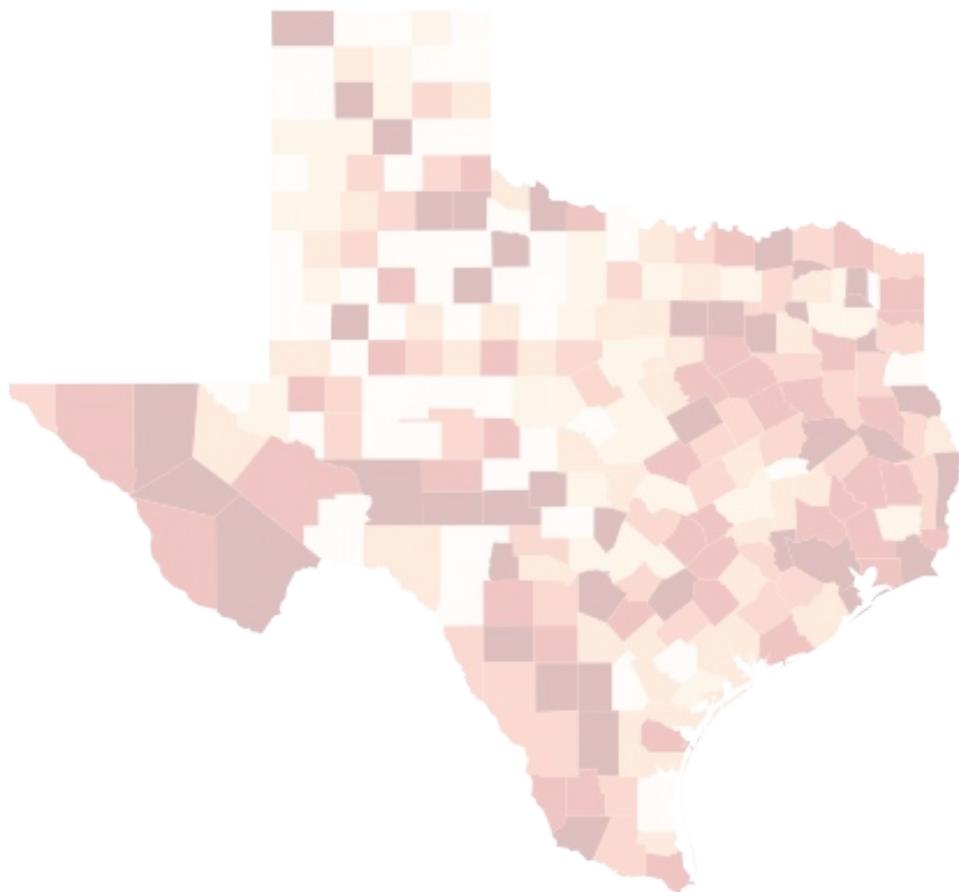


```
import matplotlib.pyplot as plt # make plot larger

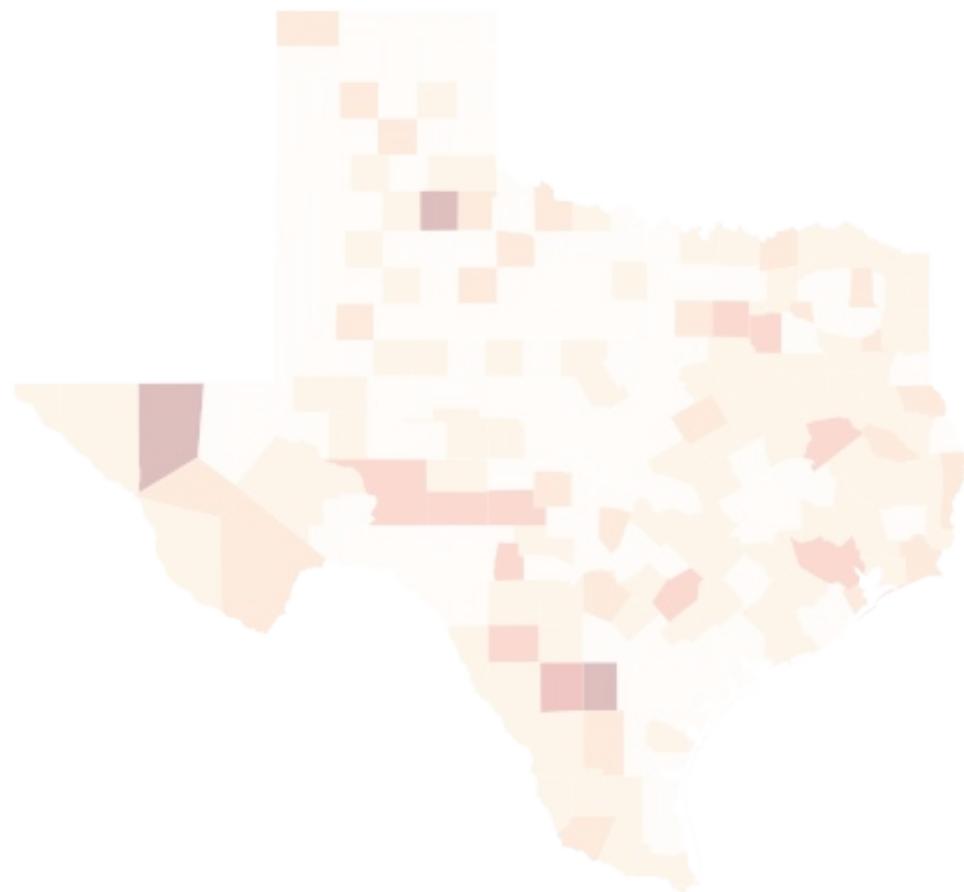
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='QUANTILES', k=5, cmap='OrRd', linewidth=0.1, ax=ax)
ax.set_axis_off()
plt.show()
```



```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='QUANTILES', \
        k=6, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white')
ax.set_axis_off()
plt.show()
```

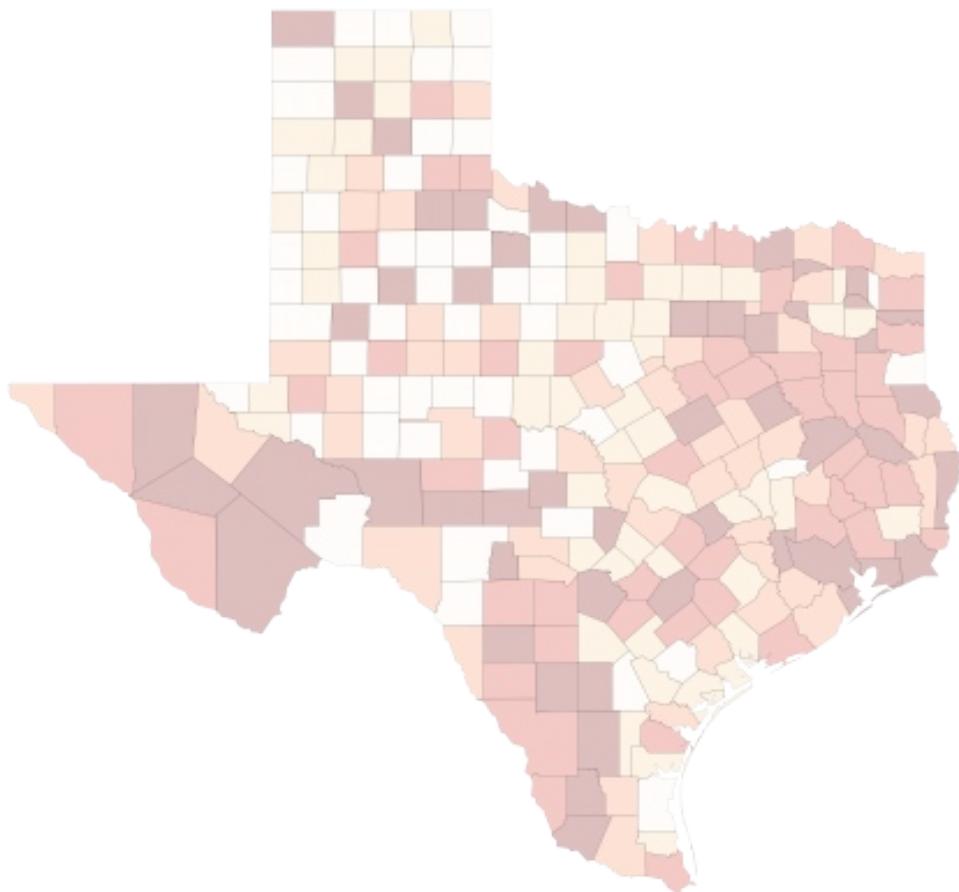


```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='equal_interval', \
        k=6, cmap='OrRd', linewidth=0.1, ax=ax, \
        edgecolor='white')
ax.set_axis_off()
plt.show()
```



```
# try deciles
f, ax = plt.subplots(1, figsize=(9, 9))
tx.plot(column='HR90', scheme='QUANTILES', k=10, cmap='OrRd', linewidth=0.1, ax=ax)
ax.set_axis_off()
plt.show()
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/geopandas/geodataframe.py:447: UserWarning: Invalid k: 10 (2 <= k <= 9), setting k=5 (default)
  return plot_dataframe(self, *args, **kwargs)
```



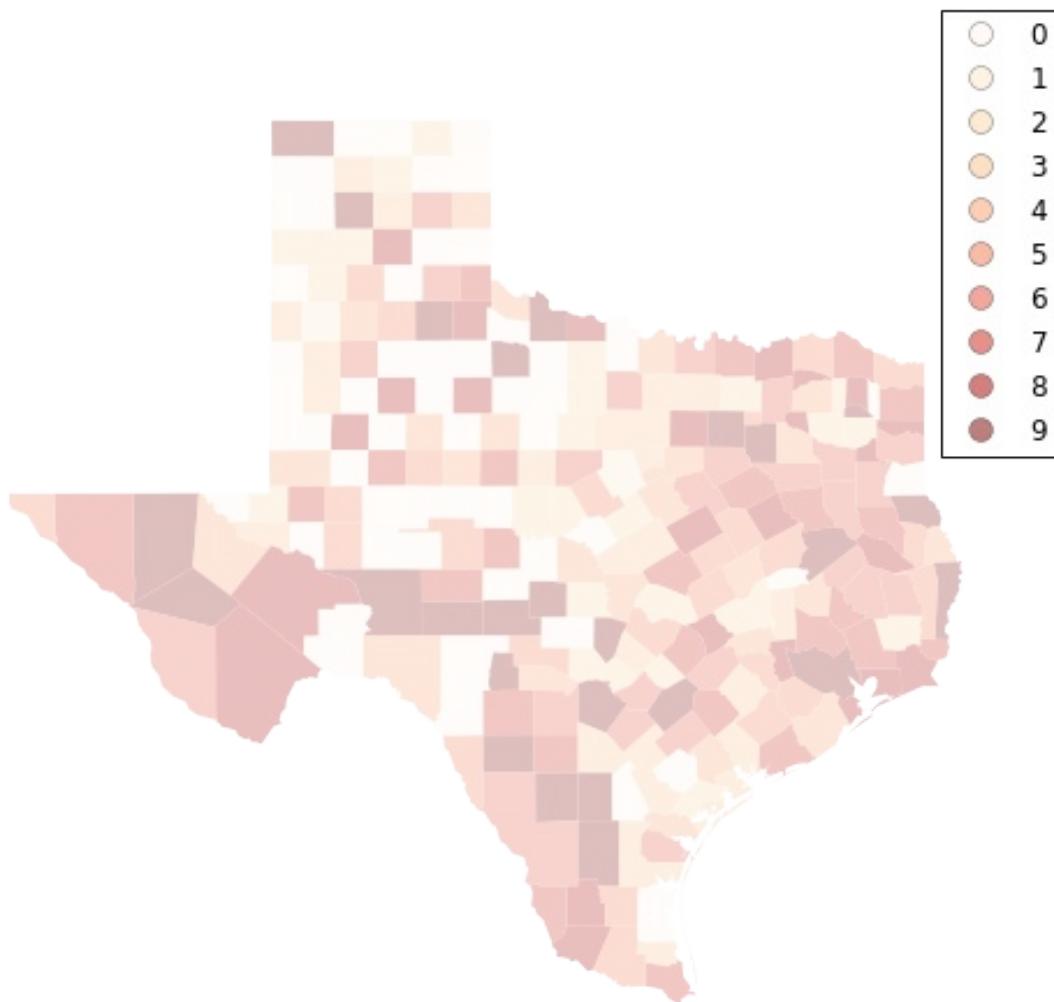
```
# ok, let's work around to get deciles
q10 = ps.Quantiles(tx.HR90, k=10)
q10.bins
```

```
array([ 0.          ,  2.42057708,  4.59760916,  5.6524773 ,
       7.23234613,  8.50963716, 10.30447074, 12.57143011,
      16.6916767 , 43.51610096])
```

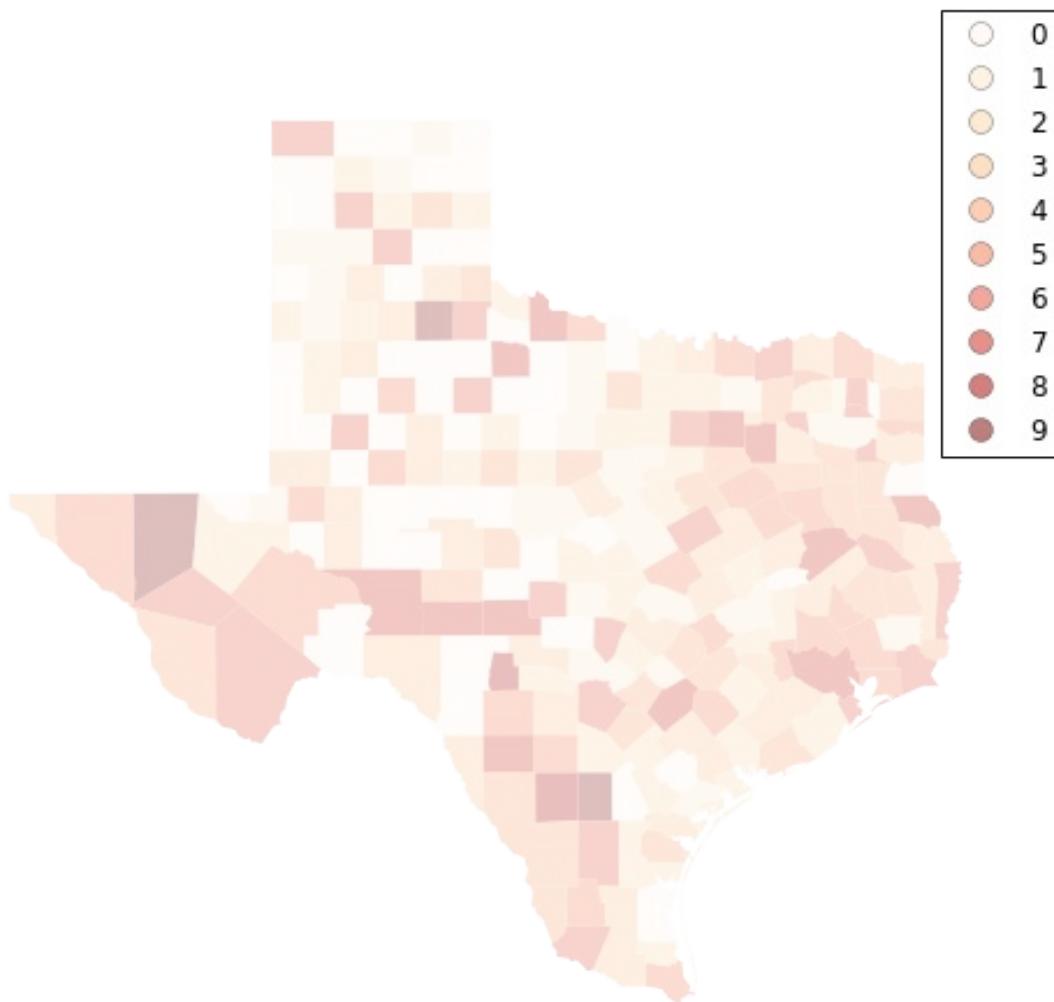
```
q10.yb
```

```
array([0, 0, 9, 0, 2, 0, 0, 2, 0, 3, 9, 3, 6, 4, 0, 2, 8, 0, 0, 2, 0, 2, 5,
       0, 7, 6, 4, 9, 9, 8, 5, 4, 1, 3, 0, 8, 0, 4, 7, 7, 6, 5, 8, 0, 0, 0,
       6, 2, 3, 9, 0, 0, 5, 8, 6, 3, 3, 6, 2, 8, 0, 0, 2, 0, 8, 2, 8, 0, 3,
       0, 4, 0, 7, 9, 2, 3, 3, 8, 9, 5, 8, 0, 4, 0, 4, 0, 8, 2, 0, 2, 8, 9,
       4, 6, 6, 8, 4, 3, 6, 7, 7, 5, 6, 3, 0, 4, 4, 1, 6, 0, 6, 7, 4, 6, 5,
       4, 6, 0, 0, 5, 0, 2, 7, 0, 2, 2, 7, 2, 8, 9, 4, 0, 7, 5, 9, 8, 7, 5,
       0, 3, 5, 3, 5, 0, 5, 0, 5, 4, 9, 7, 0, 8, 5, 0, 4, 3, 6, 8, 4, 7, 9,
       5, 6, 5, 9, 0, 7, 0, 9, 6, 4, 4, 2, 9, 2, 2, 7, 3, 2, 9, 9, 8, 0, 6,
       5, 7, 8, 2, 0, 9, 7, 7, 4, 3, 0, 4, 5, 8, 7, 8, 6, 9, 2, 5, 9, 2, 2,
       3, 4, 8, 6, 5, 9, 9, 6, 7, 5, 7, 0, 4, 8, 6, 6, 3, 3, 7, 3, 4, 9, 7,
       5, 0, 0, 3, 9, 9, 6, 2, 3, 6, 4, 3, 9, 3, 6, 3, 8, 7, 5, 0, 8, 5, 3,
       7])
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=q10.yb).plot(column='cl', categorical=True, \
    k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
    edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



```
fj10 = ps.Fisher_Jenks(tx.HR90, k=10)
fj10.bins
#labels = ["%0.1f"%l for l in fj10.bins]
#labels
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=fj10.yb).plot(column='cl', categorical=True, \
    k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
    edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



```
fj10.adcm
```

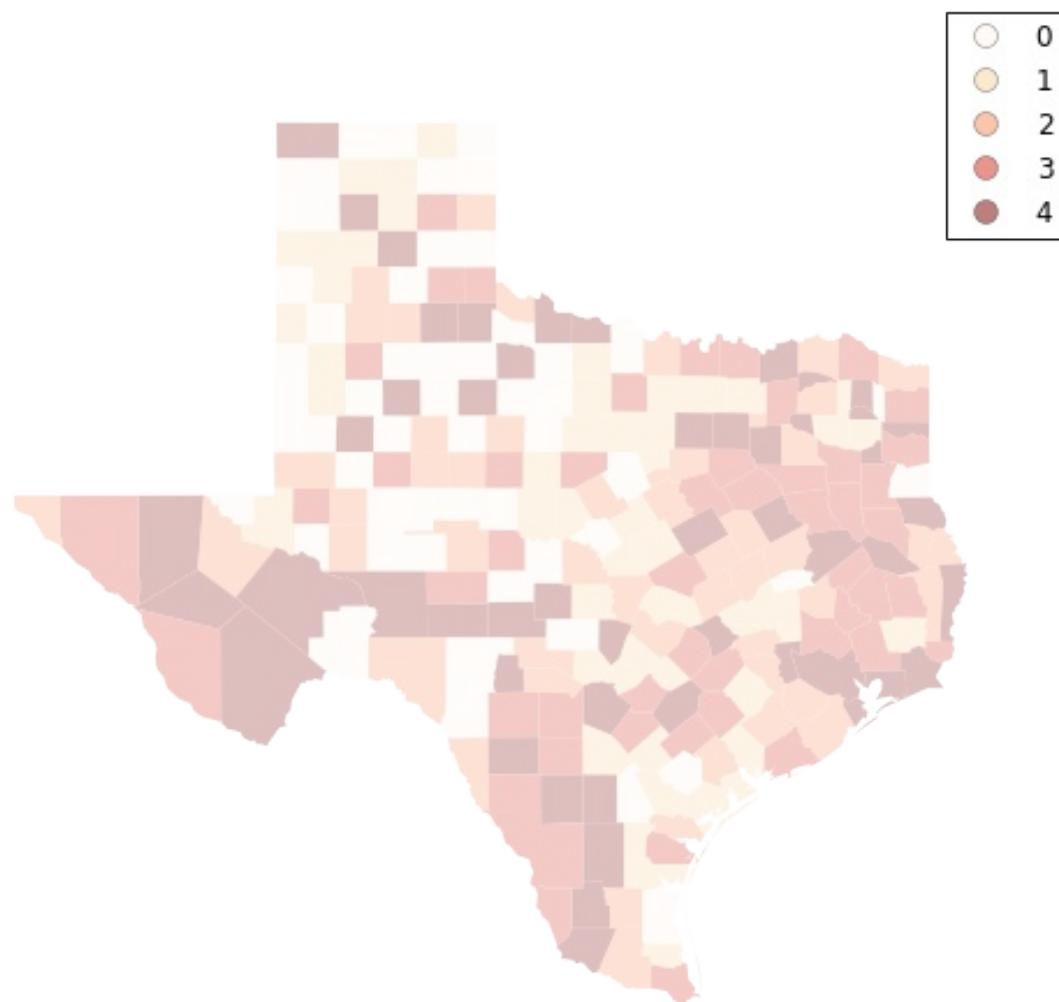
```
133.99950285589998
```

```
q10.adcm
```

```
220.80434598560004
```

```
q5 = ps.Quantiles(tx.HR90, k=5)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=q5.yb).plot(column='cl', categorical=True, \
    k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
    edgecolor='white', legend=True)
ax.set_axis_off()
plt.show()
```



Folium

In addition to using matplotlib, the `viz` module includes components that interface with the [folium](#) library which provides a Pythonic way to generate [Leaflet](#) maps.

```
import pysal as ps
import geojson as gj
from pysal.contrib.viz import folium_mapping as fm
```

First, we need to convert the data into a JSON format. JSON, short for "Javascript Serialized Object Notation," is a simple and effective way to represent objects in a digital environment. For geographic information, the [GeoJSON](#) standard defines how to represent geographic information in JSON format. Python programmers may be more comfortable thinking of JSON data as something akin to a standard Python dictionary.

```
filepath = '../data/texas.shp'[:-4]
shp = ps.open(filepath + '.shp')
dbf = ps.open(filepath + '.dbf')
```

```
js = fm.build_features(shp, dbf)
```

Just to show, this constructs a dictionary with the following keys:

```
js.keys()
```

```
dict_keys(['bbox', 'features', 'type'])
```

```
js.type
```

```
'FeatureCollection'
```

```
js.bbox
```

```
[-106.6495132446289, 25.845197677612305, -93.50721740722656, 36.49387741088867]
```

```
js.features[0]
```

```
{"bbox": [-100.5494155883789, 36.05754852294922, -99.9978867], "geometry": {"coordinates": [[[[-100.0068664550781, 0.00114440917969, 36.49251937866211], [-99.997154235839, 100.54059600830078, 36.058135986328125], [-100.549415588, [-100.00686645507812, 36.49387741088867]]], "type": "Polygon"}, "K60": 0.029359953, "BLK70": 0.0286861733, "BLK80": 0.0267356, "CNTY_FIPS": "295", "COFIPS": 295, "DNL60": 1.293879, "DNL80": 1.3953635084, "DNL90": 1.2153856529, "DV60": 2709475333, "DV80": 3.5164835165, "DV90": 6.1016949153, "FH80": 3.8353601497, "FH90": 6.0935799782, "FI295": 295, "FP59": 22.4, "FP69": 12.1, "FP79": 10.851262862, "GI": 0.2869290401, "GI69": 0.378218563, "GI79": 0.40700498, "HC60": 0.0, "HC70": 0.0, "HC80": 0.0, "HC90": 0.0, "HR60": 0.0, "HR90": 0.0, "MA60": 32.4, "MA70": 34.3, "MA80": 52241206, "L59": 8.5318847402, "MFIL69": 8.9704320743, "MFIL79": 9.843, "NAME": "Lipscomb", "P060": 3406, "P070": 348643, "POL60": 8.1332938612, "POL70": 8.1565102261, "POL80": 8.0529330368, "PS60": -1.514026445, "PS70": -1.449058083, "S90": -1.571799202, "RD60": -0.917851658, "RD70": -0.60211, "RD90": -0.605606852, "SOUTH": 1, "STATE_FIPS": "48", "STFIPS": 48, "UE60": 2.0, "UE70": 1.7, "UE80": 1.9411764}, "type": "Feature"}
```

Then, we write the json to a file:

```
with open('./example.json', 'w') as out:  
    gj.dump(js, out)
```

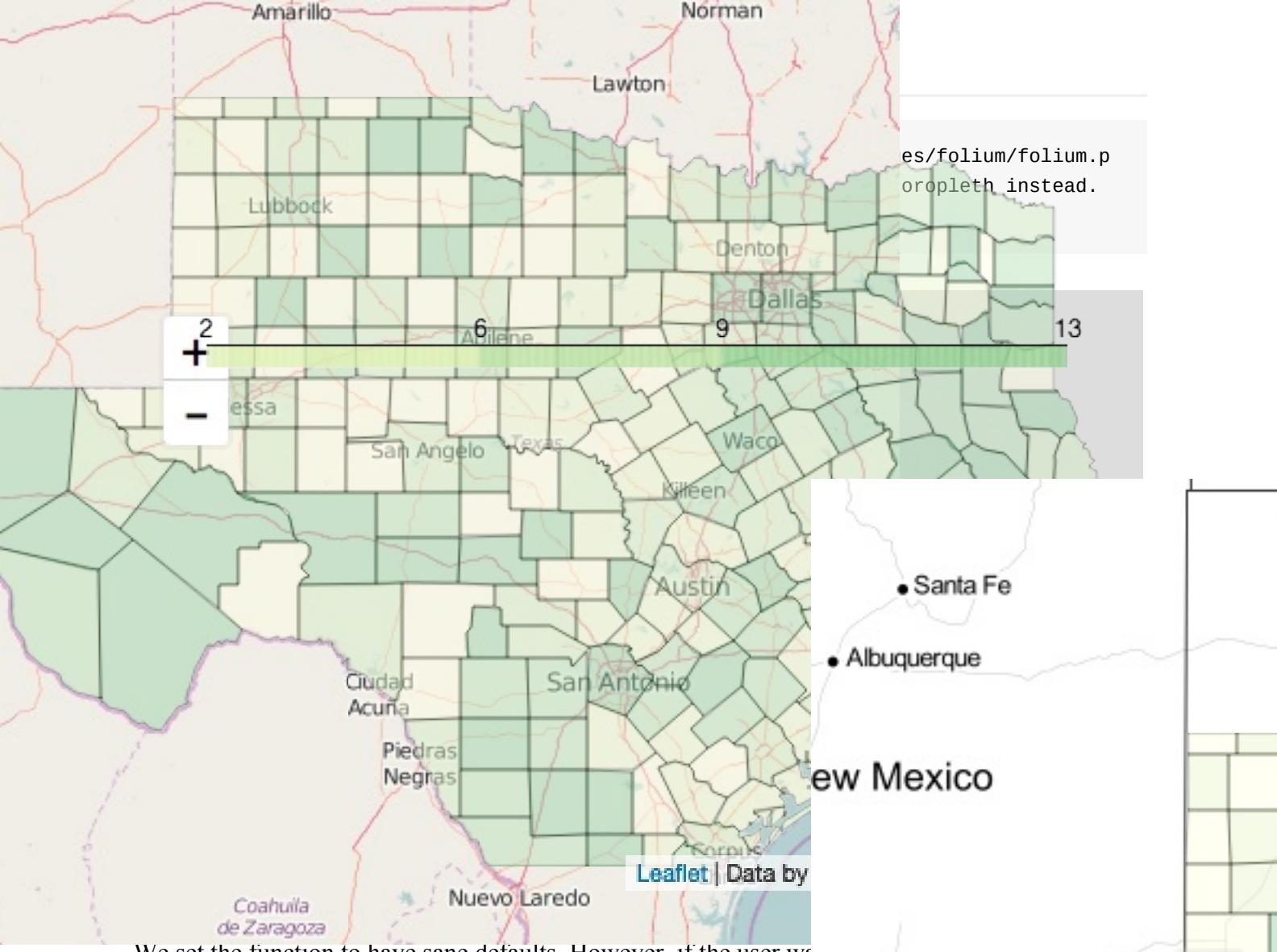
Mapping

Let's look at the columns that we are going to map.

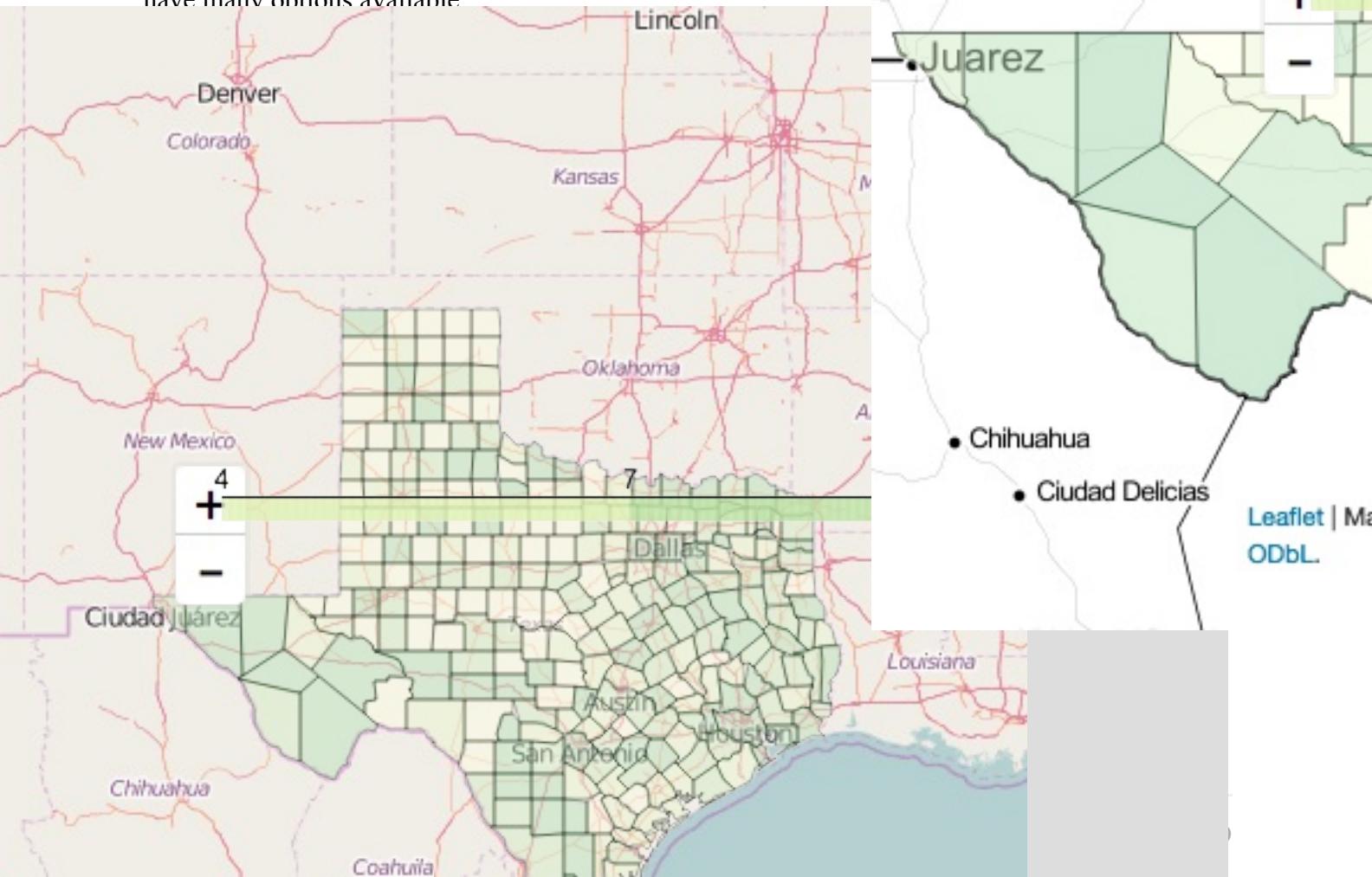
```
list(js.features[0].properties.keys())[:5]  
  
['PS60', 'FH80', 'BLK60', 'MFIL69', 'GI79']
```

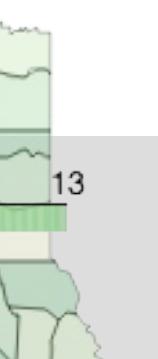
We can map these attributes by calling them as arguments to the choropleth_map function.

```
fm.choropleth_map?  
  
fm.choropleth_map('./example.json', 'FIPS', 'HR90', zoom_start=2)
```



We set the function to have sane defaults. However, if the user wants to have many options available,

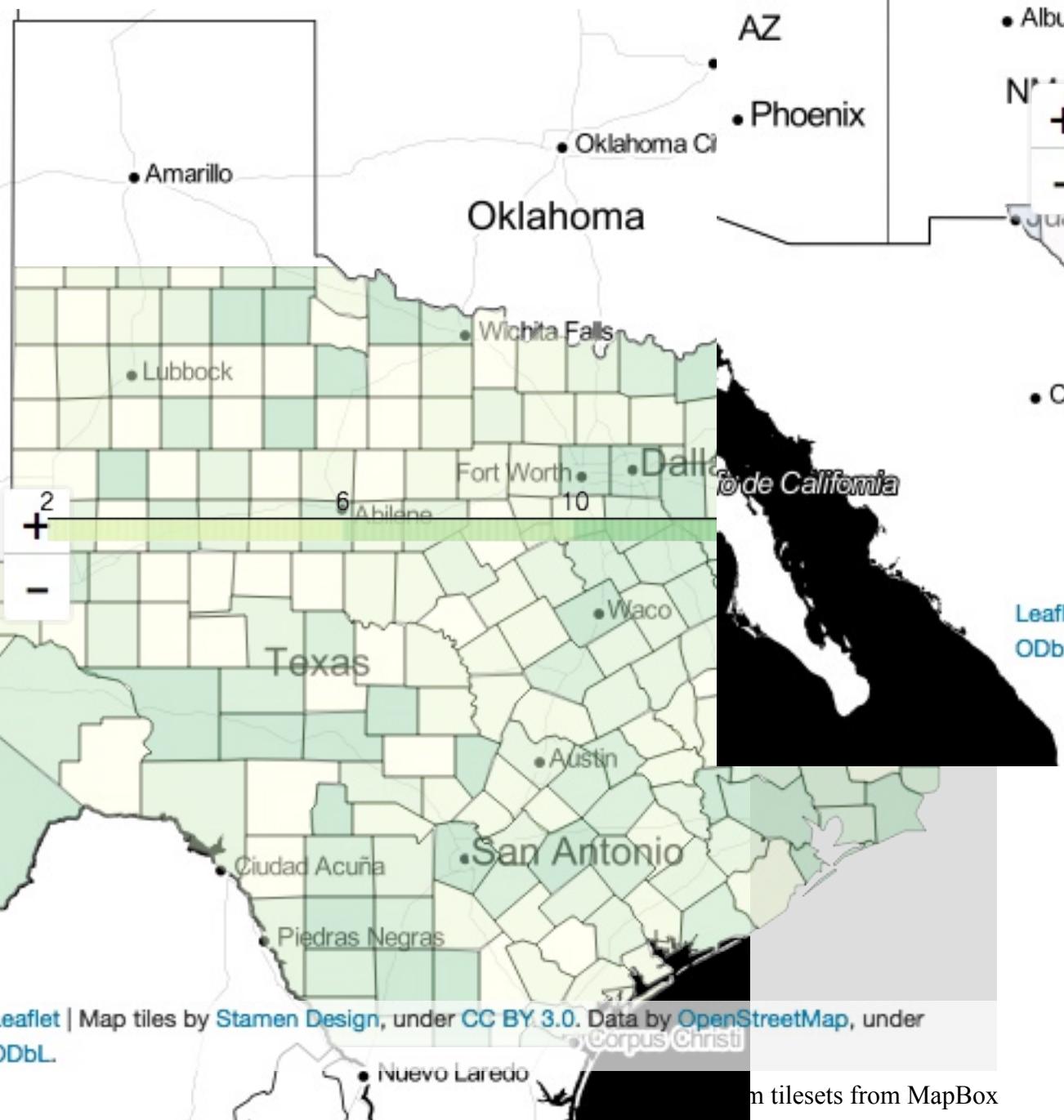


 Salt Lake City

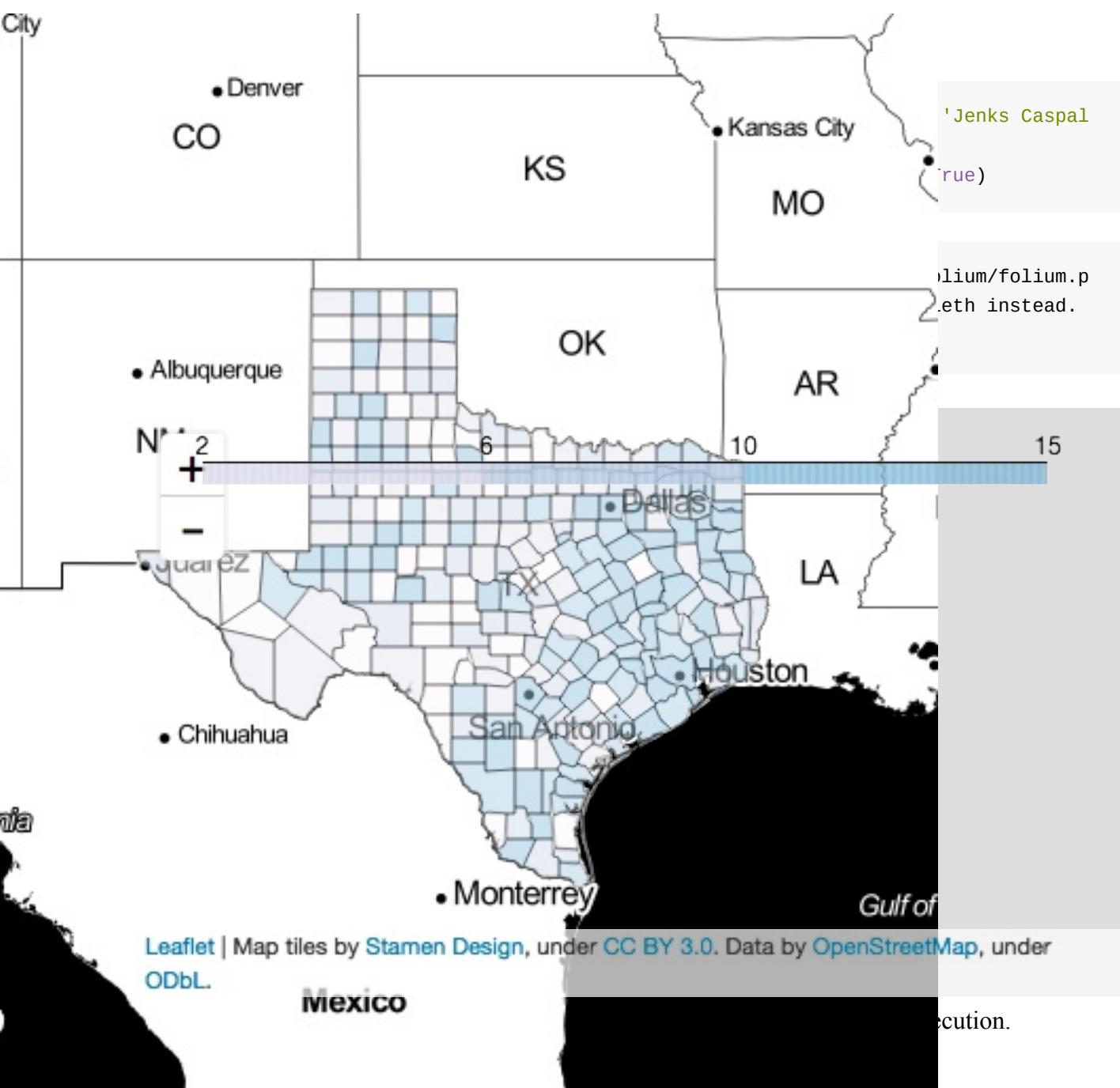
UT

CO
Denver

Most PySAL classifiers are supported.



are not supported (yet).



Cartopy

Next we turn to [cartopy](#).

```
import matplotlib.patches as mpatches
import matplotlib.pyplot as plt
import cartopy.crs as ccrs
import cartopy.io.shapereader as shapereader

reader = shapereader.Reader("../data/texas.shp")
```

```
def choropleth(classes, colors, reader, legend=None, title=None, fileName=None, dpi=600):
    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)
    if title:
        plt.title(title)
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

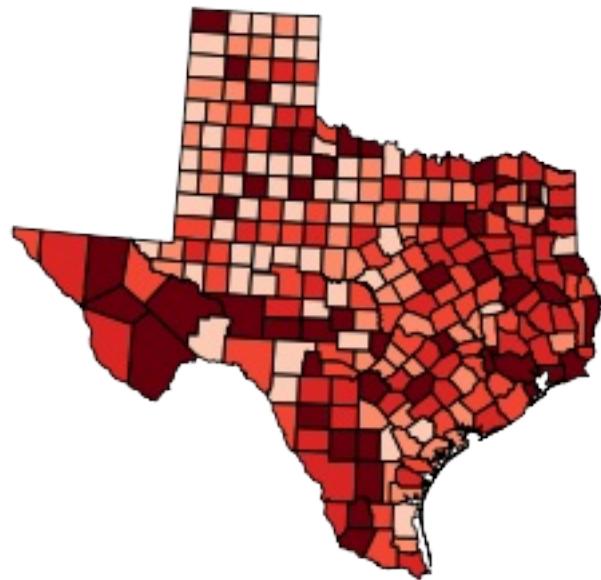
    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        #facecolor = 'red'
        edgecolor = 'black'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)

    leg = [mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors]
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1), fancybox=True)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
    plt.show()
```

```
HR90 = values
```

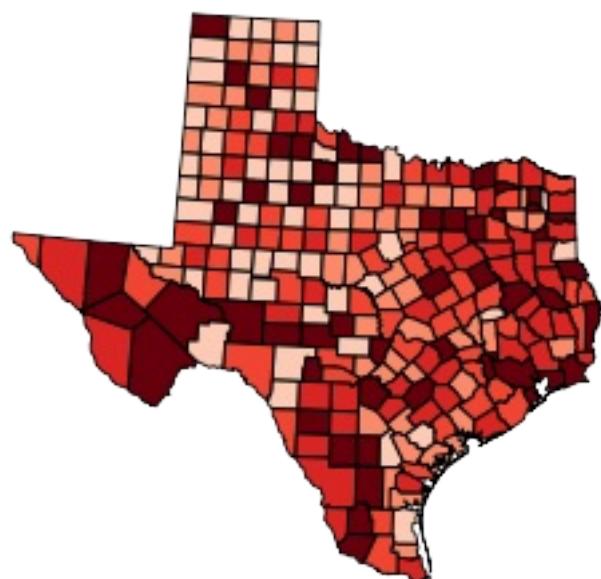
```
bins_q5 = ps.Quantiles(HR90, k=5)
```

```
bwr = plt.cm.get_cmap('Reds')
bwr(.76)
c5 = [bwr(c) for c in [0.2, 0.4, 0.6, 0.7, 1.0]]
classes = bins_q5.yb
choropleth(classes, c5, reader)
```



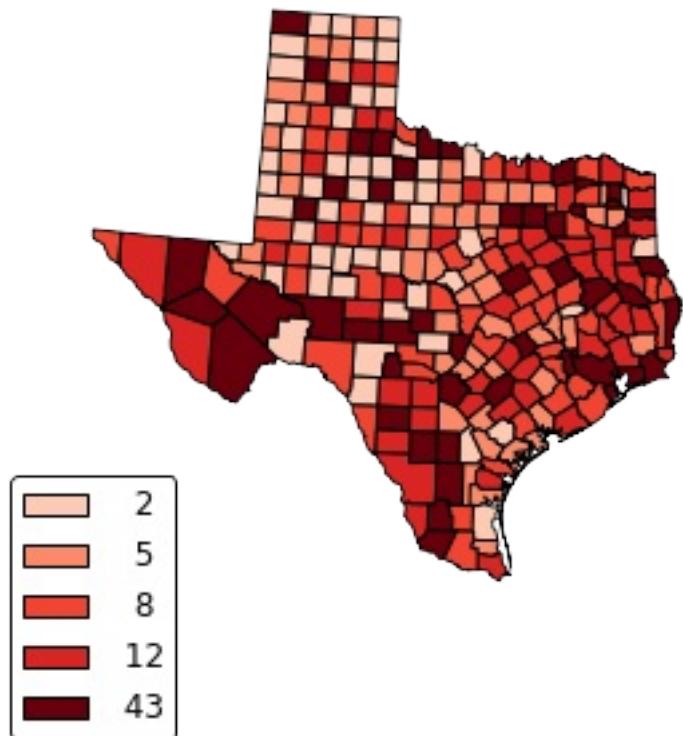
```
choropletch(classes, c5, reader, title="HR90 Quintiles")
```

HR90 Quintiles



```
legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```

HR90 Quintiles



```

def choropleth(classes, colors, reader, legend=None, title=None, fileName=None, dpi=600):
    f, ax = plt.subplots(1, figsize=(9,9))
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.axison=False

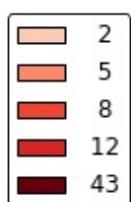
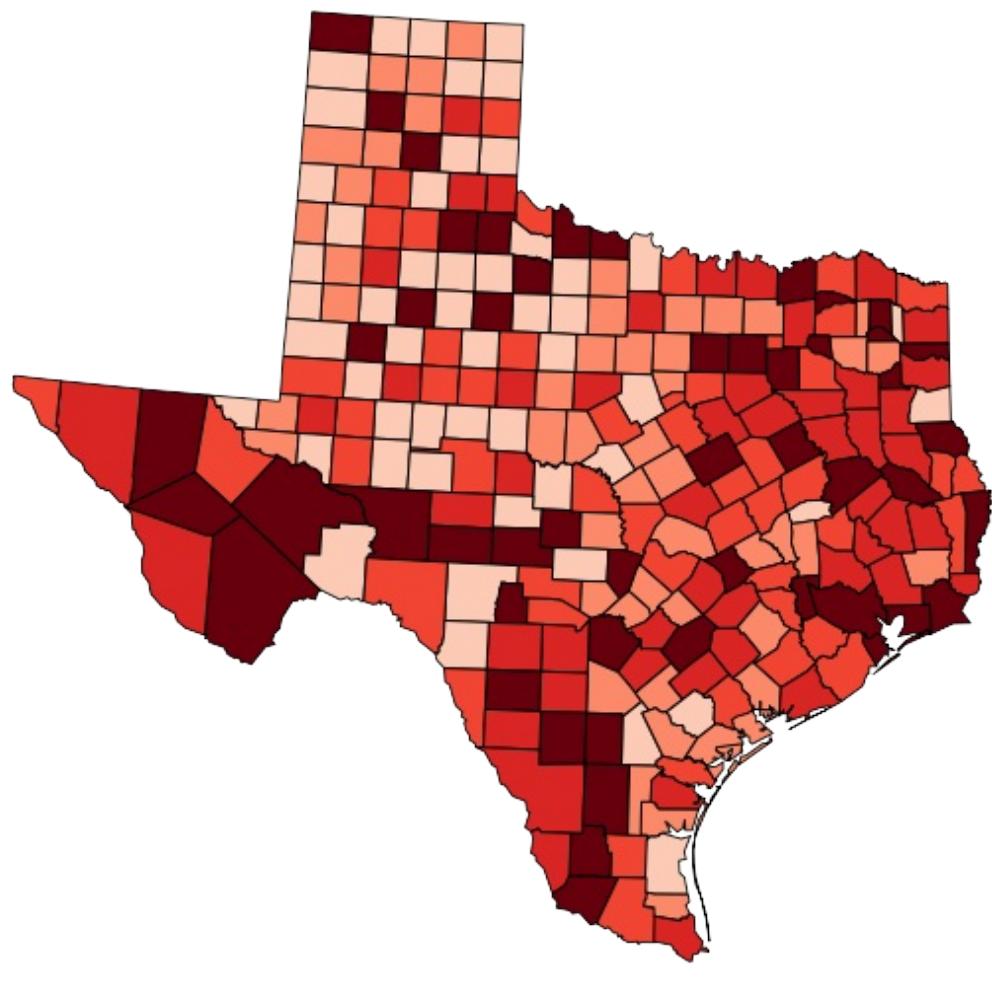
    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)
    if title:
        plt.title(title)
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)
    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        #facecolor = 'red'
        edgecolor = 'black'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)

    leg = [mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors]
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1), fancybox=True)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
        #ax.set_axis_off()
        plt.show()

legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")

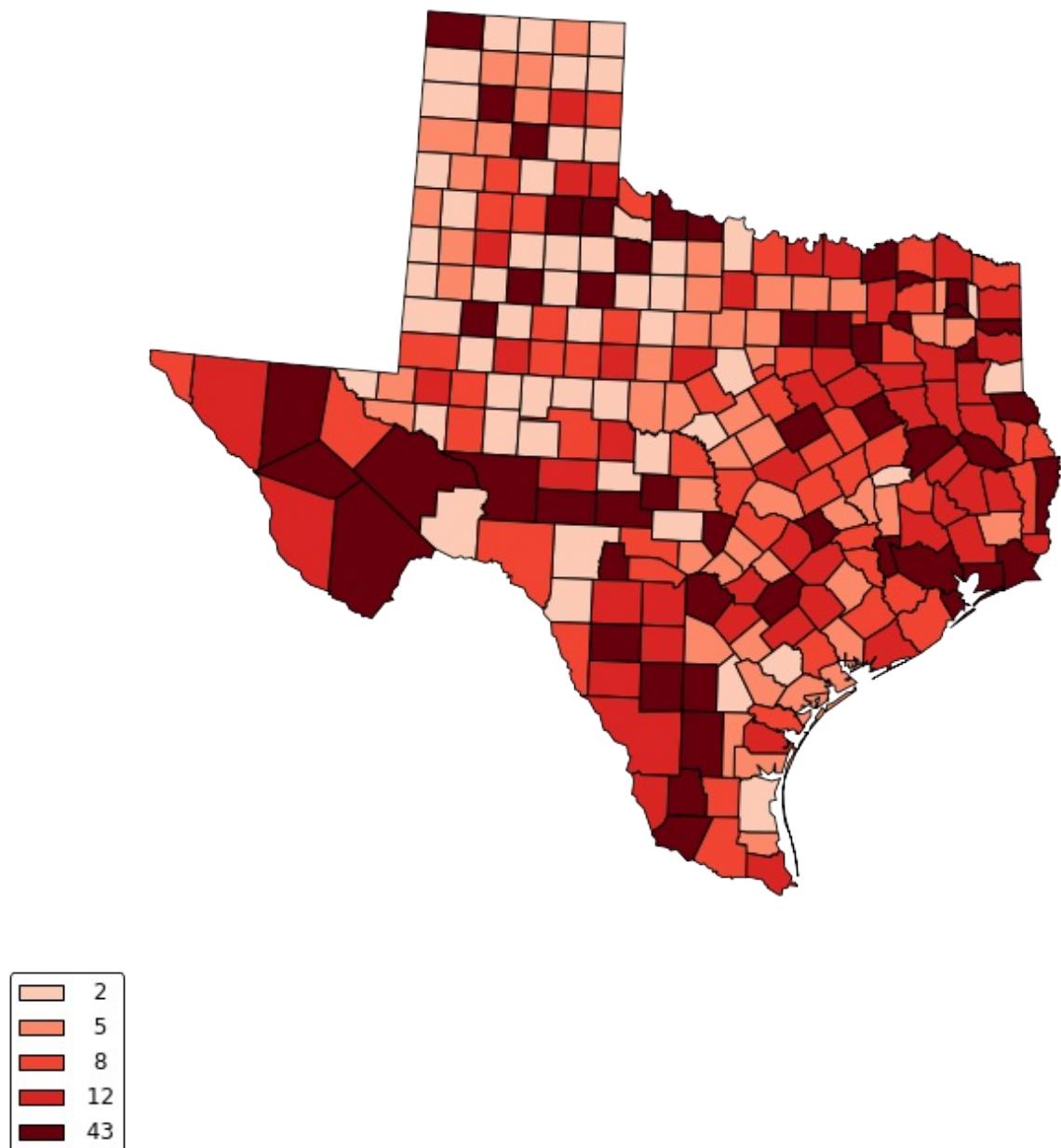
```

HR90 Quintiles



```
legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```

HR90 Quintiles



```
def choropleth(classes, colors, reader, legend=None, title=None, fileName=None, dpi=600):
    f, ax = plt.subplots(1, figsize=(9,9), frameon=False)
    ax.get_xaxis().set_visible(False)
    ax.get_yaxis().set_visible(False)
    ax.axison=False

    ax = plt.axes([0,0,1,1], projection=ccrs.LambertConformal())
    ax.set_extent([-108, -93, 38, 24], ccrs.Geodetic())
    ax.background_patch.set_visible(False)
    ax.outline_patch.set_visible(False)

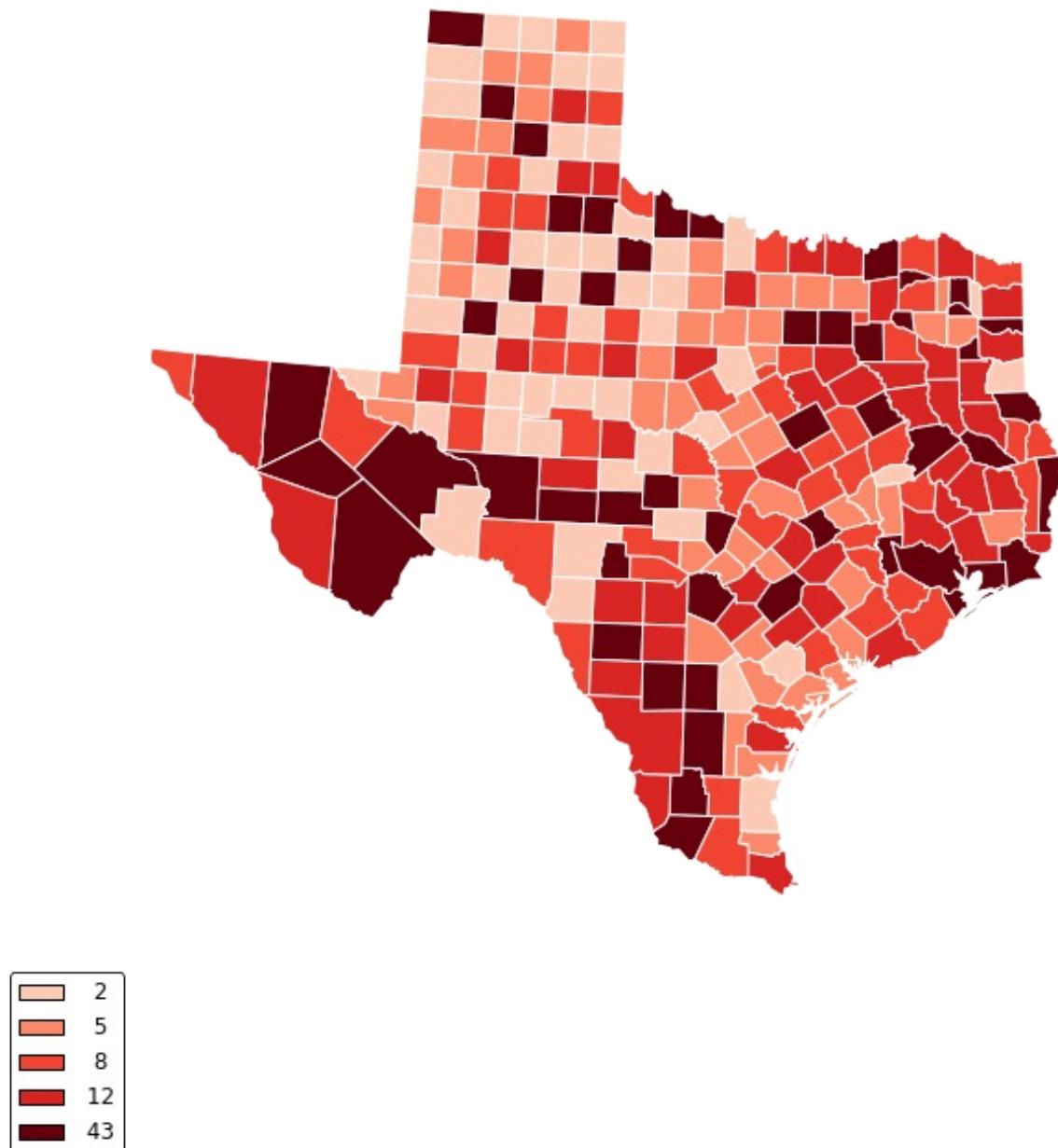
    if title:
        plt.title(title)

    for i,state in enumerate(reader.geometries()):
        facecolor = colors[classes[i]]
        edgecolor = 'white'
        ax.add_geometries([state], ccrs.PlateCarree(),
                          facecolor=facecolor, edgecolor=edgecolor)

    leg = [mpatches.Rectangle((0,0),1,1, facecolor=color) for color in colors]
    if legend:
        plt.legend(leg, legend, loc='lower left', bbox_to_anchor=(0.025, -0.1), fancybox=True)
    if fileName:
        plt.savefig(fileName, dpi=dpi)
    plt.show()

legend =[ "%3d"%ub for ub in bins_q5.bins]
choropleth(classes, c5, reader, legend, title="HR90 Quintiles")
```

HR90 Quintiles



For an example publication and code where Cartopy was used for the mapping see: [Rey \(2016\)](#).

Bokeh

[website](#)

```
from collections import OrderedDict

#from bokeh.sampledata import us_counties, unemployment
from bokeh.plotting import figure, show, output_notebook, ColumnDataSource
from bokeh.models import HoverTool

from bokeh.charts import Scatter, output_file, show

def gpd_bokeh(df):
    """Convert geometries from geopandas to bokeh format"""
    nan = float('nan')
    lons = []
    lats = []
    for i,shape in enumerate(df.geometry.values):
        if shape.geom_type == 'MultiPolygon':
            gx = []
            gy = []
            ng = len(shape.geoms) - 1
            for j,member in enumerate(shape.geoms):
                xy = np.array(list(member.exterior.coords))
                xs = xy[:,0].tolist()
                ys = xy[:,1].tolist()
                gx.extend(xs)
                gy.extend(ys)
                if j < ng:
                    gx.append(nan)
                    gy.append(nan)
            lons.append(gx)
            lats.append(gy)

        else:
            xy = np.array(list(shape.exterior.coords))
            xs = xy[:,0].tolist()
            ys = xy[:,1].tolist()
            lons.append(xs)
            lats.append(ys)

    return lons,lats
```

```
lons, lats = gpd_bokeh(tx)
```

```
p = figure(title="Texas", toolbar_location='left',
           plot_width=1100, plot_height=700)
p.patches(lons, lats, fill_alpha=0.7, #fill_color=state_colors,
           line_color="#884444", line_width=2, line_alpha=0.3)
output_file('choropleth.html', title="choropleth.py example")
show(p)
```

```
bwr = plt.cm.get_cmap('Reds')
bwr(.76)
c5 = [bwr(c) for c in [0.2, 0.4, 0.6, 0.7, 1.0]]
classes = bins_q5.yb
colors = [c5[i] for i in classes]
```

```
colors5 = ["#F1EEF6", "#D4B9DA", "#C994C7", "#DF65B0", "#DD1C77"]
colors = [colors5[i] for i in classes]

p = figure(title="Texas HR90 Quintiles", toolbar_location='left',
           plot_width=1100, plot_height=700)
p.patches(lons, lats, fill_alpha=0.7, fill_color=colors,
           line_color="#884444", line_width=2, line_alpha=0.3)
output_file('choropleth.html', title="choropleth.py example")
show(p)
```

Hover

```
from bokeh.models import HoverTool
from bokeh.plotting import figure, show, output_file, ColumnDataSource
```

```
source = ColumnDataSource(data=dict(
    x=lons,
    y=lats,
    color=colors,
    name=tx.NAME,
    rate=HR90
))

TOOLS = "pan, wheel_zoom, box_zoom, reset, hover, save"
p = figure(title="Texas Homicide 1990 (Quintiles)", tools=TOOLS,
           plot_width=900, plot_height=900)

p.patches('x', 'y', source=source,
          fill_color='color', fill_alpha=0.7,
          line_color='white', line_width=0.5)

hover = p.select_one(HoverTool)
hover.point_policy = 'follow_mouse'
hover.tooltips = [
    ("Name", "@name"),
    ("Homicide rate", "@rate"),
    ("(Long, Lat)", "($x, $y)"),
]
]

output_file("hr90.html", title="hr90.py example")
show(p)
```

Exercises

1. Using Bokeh, use PySALs Fisher Jenks classifier with k=10 to generate a choropleth map of the homicide rates in 1990 for Texas counties. Modify the hover tooltips so that in addition to showing the Homicide rate, the rank of that rate is also shown.
2. Explore `ps.esda.mapclassify`. (hint: use tab completion) to select a new classifier (different from the ones in this notebook). Using the same data as in exercise 1, apply this classifier and create a choropleth using Bokeh.

Spatial Weights

IPYNB

Spatial weights are mathematical structures used to represent spatial relationships. Many spatial analytics, such as spatial autocorrelation statistics and regionalization algorithms rely on spatial weights. Generally speaking, a spatial weight $w_{i,j}$ expresses the notion of a geographical relationship between locations i and j . These relationships can be based on a number of criteria including contiguity, geospatial distance and general distances.

PySAL offers functionality for the construction, manipulation, analysis, and conversion of a wide array of spatial weights.

We begin with construction of weights from common spatial data formats.

```
import pysal as ps
import numpy as np
```

There are functions to construct weights directly from a file path.

```
shp_path = "../data/texas.shp"
```

Weight Types

Contiguity:

Queen Weights

A commonly-used type of weight is a queen contiguity weight, which reflects adjacency relationships as a binary indicator variable denoting whether or not a polygon shares an edge or a vertex with another polygon. These weights are symmetric, in that when polygon A neighbors polygon B , both $w_{AB} = 1$ and $w_{BA} = 1$.

To construct queen weights from a shapefile, use the `queen_from_shapefile` function:

```
qw = ps.queen_from_shapefile(shp_path)
dataframe = ps.pdio.read_files(shp_path)
```

```
qw
```

```
<pysal.weights.weights.W at 0x7fb96bce8c18>
```

All weights objects have a few traits that you can use to work with the weights object, as well as to get information about the weights object.

To get the neighbors & weights around an observation, use the observation's index on the weights object, like a dictionary:

```
qw[4] #neighbors & weights of the 5th observation (0-index remember)
```

```
{0: 1.0, 3: 1.0, 5: 1.0, 6: 1.0, 7: 1.0}
```

By default, the weights and the pandas dataframe will use the same index. So, we can view the observation and its neighbors in the dataframe by putting the observation's index and its neighbors' indexes together in one list:

```
self_and_neighbors = [4]
self_and_neighbors.extend(qw.neighbors[4])
print(self_and_neighbors)
```

```
[4, 0, 3, 5, 6, 7]
```

and grabbing those elements from the dataframe:

```
dataframe.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
4	Ochiltree	Texas	48	357	48357
0	Lipscomb	Texas	48	295	48295
3	Hansford	Texas	48	195	48195
5	Roberts	Texas	48	393	48393
6	Hemphill	Texas	48	211	48211
7	Hutchinson	Texas	48	233	48233

6 rows × 70 columns

A full, dense matrix describing all of the pairwise relationships is constructed using the `.full` method, or when `pysal.full` is called on a weights object:

```
Wmatrix, ids = qW.full()
#Wmatrix, ids = ps.full(qW)
```

`Wmatrix`

```
array([[ 0.,  0.,  0., ...,  0.,  0.,  0.],
       [ 0.,  0.,  1., ...,  0.,  0.,  0.],
       [ 0.,  1.,  0., ...,  0.,  0.,  0.],
       ...,
       [ 0.,  0.,  0., ...,  0.,  1.,  1.],
       [ 0.,  0.,  0., ...,  1.,  0.,  1.],
       [ 0.,  0.,  0., ...,  1.,  1.,  0.]])
```

```
n_neighbors = Wmatrix.sum(axis=1) # how many neighbors each region has
```

`n_neighbors[4]`

```
5.0
```

```
qw.cardinalities[4]
```

```
5
```

Note that this matrix is binary, in that its elements are either zero or one, since an observation is either a neighbor or it is not a neighbor.

However, many common use cases of spatial weights require that the matrix is row-standardized. This is done simply in PySAL using the `.transform` attribute

```
qw.transform = 'r'
```

Now, if we build a new full matrix, its rows should sum to one:

```
Wmatrix, ids = qw.full()
```

```
Wmatrix.sum(axis=1) #numpy axes are 0:column, 1:row, 2:facet, into higher dimensions
```

Since weight matrices are typically very sparse, there is also a sparse weights matrix constructor:

qw.sparse

```
<254x254 sparse matrix of type '<class 'numpy.float64'>'  
with 1460 stored elements in Compressed Sparse Row format>
```

```
qw.pct_nonzero #Percentage of nonzero neighbor counts
```

2.263004526009052

By default, PySAL assigns each observation an index according to the order in which the observation was read in. This means that, by default, all of the observations in the weights object are indexed by table order. If you have an alternative ID variable, you can pass that into the weights constructor.

For example, the `texas.shp` dataset has a possible alternative ID Variable, a `FIPS` code.

```
dataframe.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
0	Lipscomb	Texas	48	295	48295
1	Sherman	Texas	48	421	48421
2	Dallam	Texas	48	111	48111
3	Hansford	Texas	48	195	48195
4	Ochiltree	Texas	48	357	48357

5 rows × 70 columns

The observation we were discussing above is in the fifth row: Ochiltree county, Texas. Note that its FIPS code is 48357.

Then, instead of indexing the weights and the dataframe just based on read-order, use the `FIPS` code as an index:

```
qw = ps.queen_from_shapefile(shp_path, idVariable='FIPS')
```

```
qw[4] #fails, since no FIPS is 4.
```

```

-----
KeyError                                     Traceback (most recent call last)

<ipython-input-20-1d8a3009bc1e> in <module>()
----> 1 qw[4] #fails, since no FIPS is 4.

/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/pysal/weights/w
eights.py in __getitem__(self, key)
    504         {1: 1.0, 4: 1.0, 101: 1.0, 85: 1.0, 5: 1.0}
    505         """
--> 506         return dict(list(zip(self.neighbors[key], self.weights[key])))
    507
    508     def __iter__(self):
KeyError: 4

```

Note that a `KeyError` in Python usually means that some index, here `4`, was not found in the collection being searched, the IDs in the queen weights object. This makes sense, since we explicitly passed an `idVariable` argument, and nothing has a `FIPS` code of 4.

Instead, if we use the observation's `FIPS` code:

```
qw['48357']
```

```
{'48195': 1.0, '48211': 1.0, '48233': 1.0, '48295': 1.0, '48393': 1.0}
```

We get what we need.

In addition, we have to now query the dataframe using the `FIPS` code to find our neighbors. But, this is relatively easy to do, since pandas will parse the query by looking into python objects, if told to.

First, let us store the neighbors of our target county:

```
self_and_neighbors = ['48357']
self_and_neighbors.extend(qw.neighbors['48357'])
```

Then, we can use this list in `.query`:

```
dataframe.query('FIPS in @self_and_neighbors')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
0	Lipscomb	Texas	48	295	48295
3	Hansford	Texas	48	195	48195
4	Ochiltree	Texas	48	357	48357
5	Roberts	Texas	48	393	48393
6	Hemphill	Texas	48	211	48211
7	Hutchinson	Texas	48	233	48233

6 rows × 70 columns

Note that we have to use `@` before the name in order to show that we're referring to a python object and not a column in the dataframe.

```
#dataframe.query('FIPS in self_and_neighbors') will fail because there is no column called 'self_and_neighbors'
```

Of course, we could also reindex the dataframe to use the same index as our weights:

```
fips_frame = dataframe.set_index(dataframe.FIPS)
fips_frame.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
FIPS					
48295	Lipscomb	Texas	48	295	4829
48421	Sherman	Texas	48	421	4842
48111	Dallam	Texas	48	111	4811
48195	Hansford	Texas	48	195	4819
48357	Ochiltree	Texas	48	357	4835

5 rows × 70 columns

Now that both are using the same weights, we can use the `.loc` indexer again:

```
fips_frame.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
FIPS					
48357	Ochiltree	Texas	48	357	483
48295	Lipscomb	Texas	48	295	482
48195	Hansford	Texas	48	195	481
48393	Roberts	Texas	48	393	483
48211	Hemphill	Texas	48	211	482
48233	Hutchinson	Texas	48	233	482

6 rows × 70 columns

Rook Weights

Rook weights are another type of contiguity weight, but consider observations as neighboring only when they share an edge. The rook neighbors of an observation may be different than its queen neighbors, depending on how the observation and its nearby polygons are configured.

We can construct this in the same way as the queen weights, using the special `rook_from_shapefile` function:

```
rw = ps.rook_from_shapefile(shp_path, idVariable='FIPS')
```

```
rw['48357']
```

```
{'48195': 1.0, '48295': 1.0, '48393': 1.0}
```

These weights function exactly like the Queen weights, and are only distinguished by what they consider "neighbors."

```
self_and_neighbors = ['48357']
self_and_neighbors.extend(rw.neighbors['48357'])
fips_frame.loc[self_and_neighbors]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
FIPS					
48357	Ochiltree	Texas	48	357	4835
48295	Lipscomb	Texas	48	295	4829
48195	Hansford	Texas	48	195	4819
48393	Roberts	Texas	48	393	4839

4 rows × 70 columns

Bishop Weights

In theory, a "Bishop" weighting scheme is one that arises when only polygons that share vertexes are considered to be neighboring. But, since Queen contiguity requires either an edge or a vertex and Rook contiguity requires only shared edges, the following relationship is true:

$$\mathcal{Q} = \mathcal{R} \cup \mathcal{B}$$

where \mathcal{Q} is the set of neighbor pairs *via* queen contiguity, \mathcal{R} is the set of neighbor pairs *via* Rook contiguity, and \mathcal{B} *via* Bishop contiguity. Thus:

$$\mathcal{Q} \setminus \mathcal{R} = \mathcal{B}$$

Bishop weights entail all Queen neighbor pairs that are not also Rook neighbors.

PySAL does not have a dedicated bishop weights constructor, but you can construct very easily using the `w_difference` function. This function is one of a family of tools to work with weights, all defined in `ps.weights`, that conduct these types of set operations between weight objects.

```
bw = ps.w_difference(qW, rW, constrained=False, silent_island_warning=True) #silence because there will be a lot of warnings
```

```
bw.histogram
```

```
[(0, 161), (1, 48), (2, 33), (3, 8), (4, 4)]
```

Thus, the vast majority of counties have no bishop neighbors. But, a few do. A simple way to see these observations in the dataframe is to find all elements of the dataframe that are not "islands," the term for an observation with no neighbors:

```
islands = bw.islands
```

```
dataframe.query('FIPS not in @islands')
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FI
0	Lipscomb	Texas	48	295	482
1	Sherman	Texas	48	421	484
2	Dallam	Texas	48	111	481

3	Hansford	Texas	48	195	481
4	Ochiltree	Texas	48	357	482
5	Roberts	Texas	48	393	483
6	Hemphill	Texas	48	211	482
7	Hutchinson	Texas	48	233	482
8	Hartley	Texas	48	205	482
9	Moore	Texas	48	341	483
10	Potter	Texas	48	375	483
11	Carson	Texas	48	065	480
12	Gray	Texas	48	179	481
13	Wheeler	Texas	48	483	484
14	Oldham	Texas	48	359	483
15	Randall	Texas	48	381	483
16	Armstrong	Texas	48	011	480
17	Collingsworth	Texas	48	087	480
18	Donley	Texas	48	129	481
27	Wilbarger	Texas	48	487	484
28	Motley	Texas	48	345	483

Spatial weights in PySAL

28	Motley	Texas	48	345	481
29	Cottle	Texas	48	101	481
30	Floyd	Texas	48	153	481
31	Hale	Texas	48	189	481
32	Lamb	Texas	48	279	482
33	Bailey	Texas	48	017	480
35	Wichita	Texas	48	485	484
38	Red River	Texas	48	387	481
41	Lamar	Texas	48	277	482
43	King	Texas	48	269	482
...
119	Midland	Texas	48	329	481
126	Anderson	Texas	48	001	480
128	Freestone	Texas	48	161	481
132	Hudspeth	Texas	48	229	482
140	Tom Green	Texas	48	451	484
144	Upton	Texas	48	461	484
145	Reagan	Texas	48	383	481

Spatial weights in PySAL

146	Leon	Texas	48	289	482
149	Concho	Texas	48	095	480
158	Robertson	Texas	48	395	481
165	Madison	Texas	48	313	483
166	Schleicher	Texas	48	413	484
167	Menard	Texas	48	327	483
172	Brazos	Texas	48	041	480
183	Presidio	Texas	48	377	483
191	Bastrop	Texas	48	021	480
200	Fayette	Texas	48	149	481
205	Caldwell	Texas	48	055	480
212	Gonzales	Texas	48	177	481
214	Medina	Texas	48	325	483
217	Uvalde	Texas	48	463	484
218	Kinney	Texas	48	271	482
224	Atascosa	Texas	48	013	480
228	Zavala	Texas	48	507	485
229	Frio	Texas	48	163	481

230	Maverick	Texas	48	323	483
234	La Salle	Texas	48	283	482
235	McMullen	Texas	48	311	483
239	Webb	Texas	48	479	484
242	Duval	Texas	48	131	481

93 rows × 70 columns

Distance

There are many other kinds of weighting functions in PySAL. Another separate type use a continuous measure of distance to define neighborhoods.

```
radius = ps.cg.sphere.RADIUS_EARTH_MILES
radius
```

```
3958.755865744055
```

```
ps.min_threshold_dist_from_shapefile?
```

```
threshold = ps.min_threshold_dist_from_shapefile('../data/texas.shp', radius) # now
in miles, maximum nearest neighbor distance between the n observations
```

```
threshold
```

```
60.47758554135752
```

knn defined weights

```
knn4_bad = ps.knnW_from_shapefile('../data/texas.shp', k=4) # ignore curvature of  
the earth
```

```
knn4_bad.histogram
```

```
[(4, 254)]
```

```
knn4 = ps.knnW_from_shapefile('../data/texas.shp', k=4, radius=radius)
```

```
knn4.histogram
```

```
[(4, 254)]
```

```
knn4[0]
```

```
{3: 1.0, 4: 1.0, 5: 1.0, 6: 1.0}
```

```
knn4_bad[0]
```

```
{4: 1.0, 5: 1.0, 6: 1.0, 13: 1.0}
```

Kernel W

Kernel Weights are continuous distance-based weights that use kernel densities to define the neighbor relationship. Typically, they estimate a `bandwidth`, which is a parameter governing how far out observations should be considered neighboring. Then, using this bandwidth, they evaluate a continuous kernel function to provide a weight between 0 and 1.

Many different choices of kernel functions are supported, and bandwidths can either be fixed (constant over all units) or adaptive in function of unit density.

For example, if we want to use adaptive bandwidths for the map and weight according to a gaussian kernel:

```
kernelWa = ps.adaptive_kernelW_from_shapefile('../data/texas.shp', radius=radius)
kernelWa
```

```
<pysal.weights.Distance.Kernel at 0x7fb96b8979b0>
```

```
dataframe.loc[kernelWa.neighbors[4] + [4]]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
4	Ochiltree	Texas	48	357	48357
5	Roberts	Texas	48	393	48393
3	Hansford	Texas	48	195	48195
4	Ochiltree	Texas	48	357	48357

4 rows × 70 columns

```
kernelWa.bandwidth[0:7]
```

```
array([[ 30.30546757],
       [ 30.05684855],
       [ 39.14876899],
       [ 29.96302462],
       [ 29.96302462],
       [ 30.21084447],
       [ 30.23619029]])
```

```
kernelWa[4]
```

```
{3: 9.99999900663795e-08, 4: 1.0, 5: 0.002299013803371608}
```

```
kernelWa[2]
```

```
{1: 9.9999900663795e-08, 2: 1.0, 8: 0.23409571720488287}
```

Distance Thresholds

```
ps.min_threshold_dist_from_shapefile?
```

```
# find the largest nearest neighbor distance between centroids
threshold = ps.min_threshold_dist_from_shapefile('../data/texas.shp', radius=radius) # decimal degrees
Wmind0 = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radius, threshold=threshold*.9)
```

```
WARNING: there are 2 disconnected observations
Island ids: [133, 181]
```

```
Wmind0.histogram
```

```
[(0, 2),
 (1, 3),
 (2, 5),
 (3, 4),
 (4, 10),
 (5, 26),
 (6, 16),
 (7, 31),
 (8, 70),
 (9, 32),
 (10, 29),
 (11, 12),
 (12, 5),
 (13, 2),
 (14, 5),
 (15, 2)]
```

```
Wmind = ps.threshold_binaryW_from_shapefile('../data/texas.shp', radius=radius, threshold=threshold)
```

```
Wmind.histogram
```

```
[(1, 2),  
 (2, 3),  
 (3, 4),  
 (4, 8),  
 (5, 5),  
 (6, 20),  
 (7, 26),  
 (8, 9),  
 (9, 32),  
 (10, 31),  
 (11, 37),  
 (12, 33),  
 (13, 23),  
 (14, 6),  
 (15, 7),  
 (16, 2),  
 (17, 4),  
 (18, 2)]
```

```
centroids = np.array([list(poly.centroid) for poly in dataframe.geometry])
```

```
centroids[0:10]
```

```
array([[[-100.27156111, 36.27508641],  
       [-101.8930971 , 36.27325425],  
       [-102.59590795, 36.27354996],  
       [-101.35351324, 36.27230422],  
       [-100.81561379, 36.27317803],  
       [-100.81482387, 35.8405153 ],  
       [-100.2694824 , 35.83996075],  
       [-101.35420366, 35.8408377 ],  
       [-102.59375964, 35.83958662],  
       [-101.89248229, 35.84058246]])
```

```
Wmind[0]
```

```
{3: 1, 4: 1, 5: 1, 6: 1, 13: 1}
```

```
knn4[0]
```

```
{3: 1.0, 4: 1.0, 5: 1.0, 6: 1.0}
```

Visualization

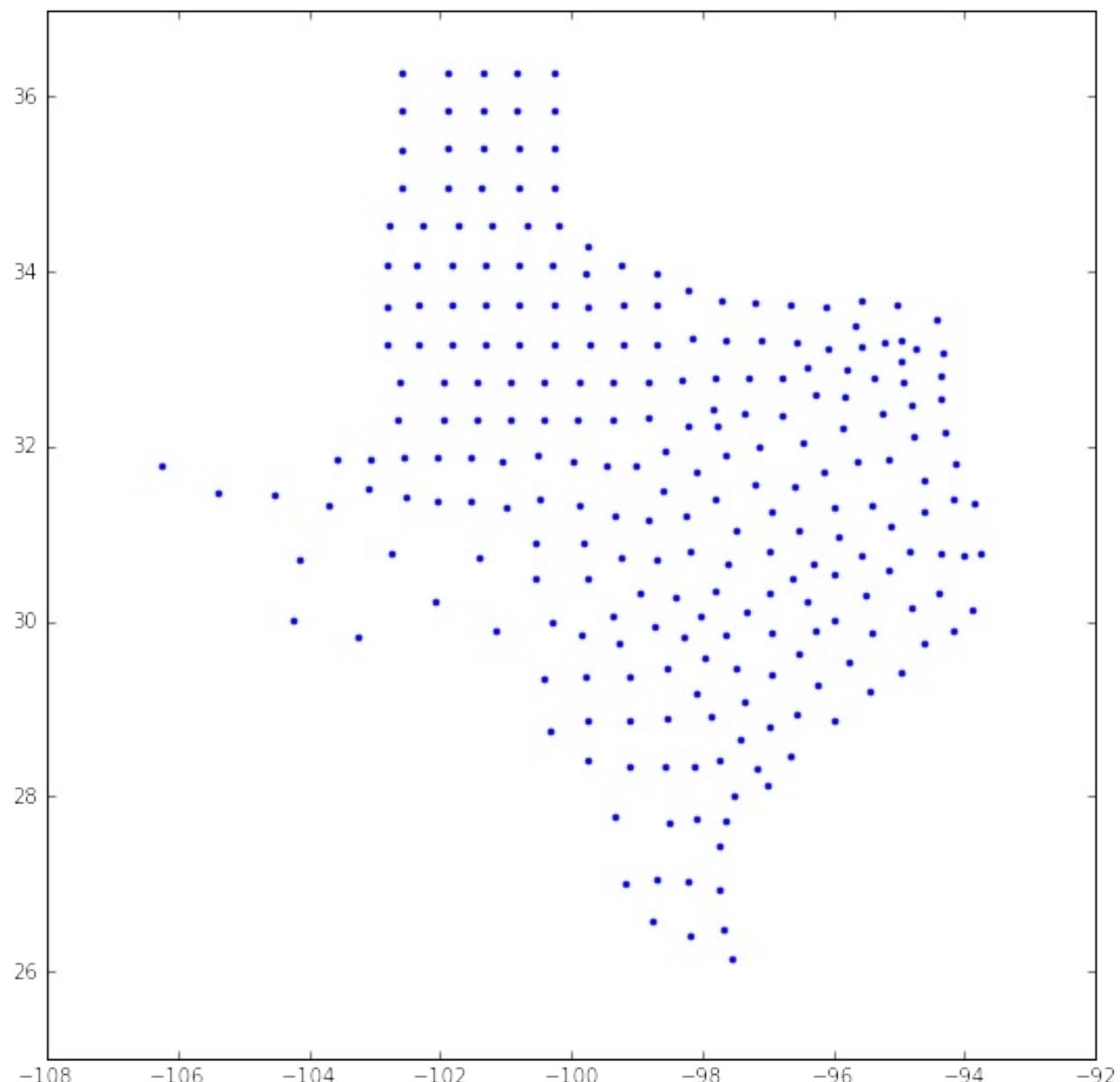
```
%matplotlib inline
import matplotlib.pyplot as plt
from pylab import figure, scatter, show
```

```
wq = ps.queen_from_shapefile('../data/texas.shp')
```

```
wq[0]
```

```
{4: 1.0, 5: 1.0, 6: 1.0}
```

```
fig = figure(figsize=(9,9))
plt.plot(centroids[:,0], centroids[:,1], '.')
plt.ylim([25,37])
show()
```



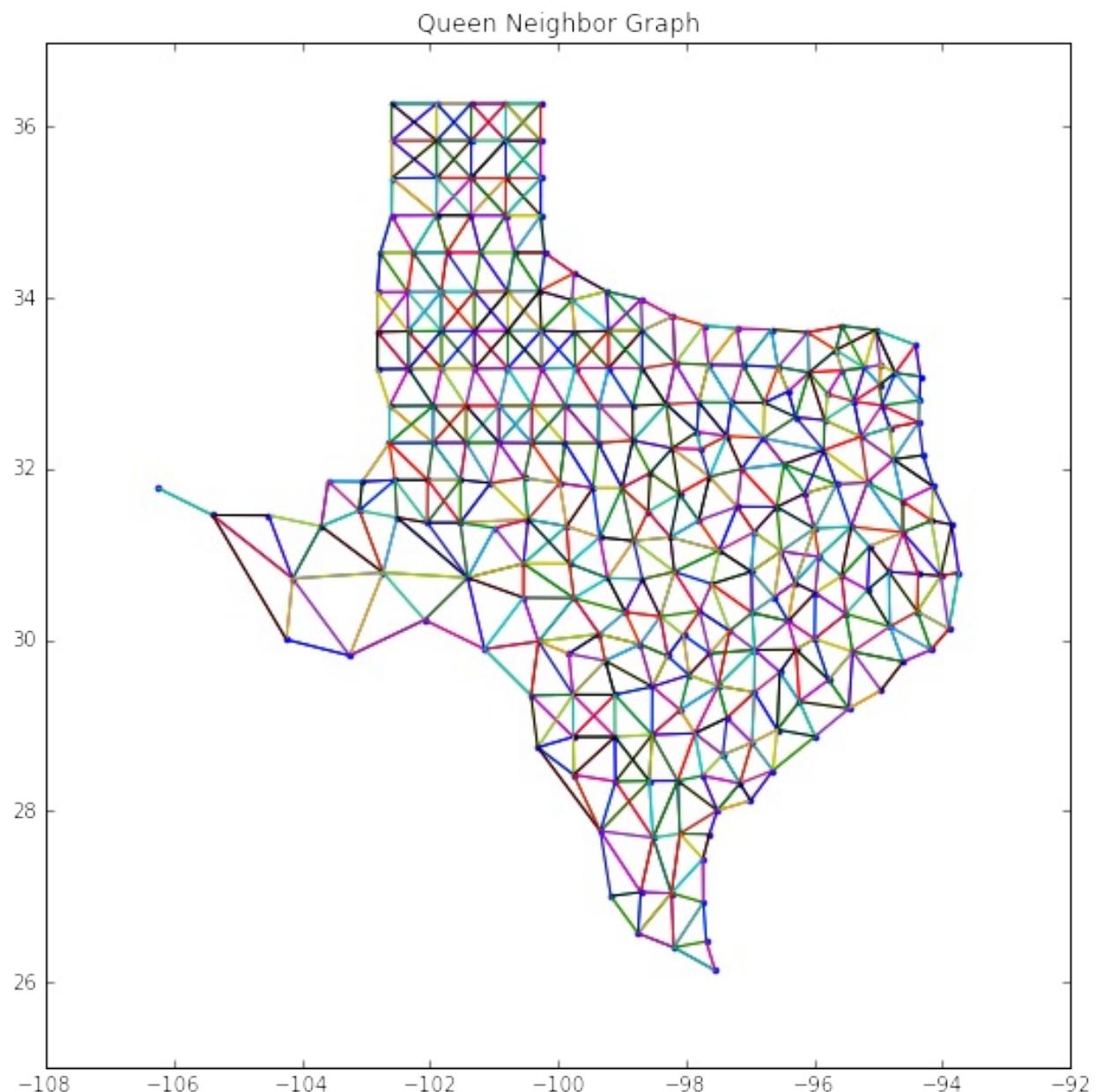
```
wq.neighbors[0]
```

```
[4, 5, 6]
```

```
from pylab import figure, scatter, show
fig = figure(figsize=(9,9))

plt.plot(centroids[:,0], centroids[:,1], '.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in wq.neighbors.items():
    #print(k,neighs)
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')

plt.title('Queen Neighbor Graph')
show()
```

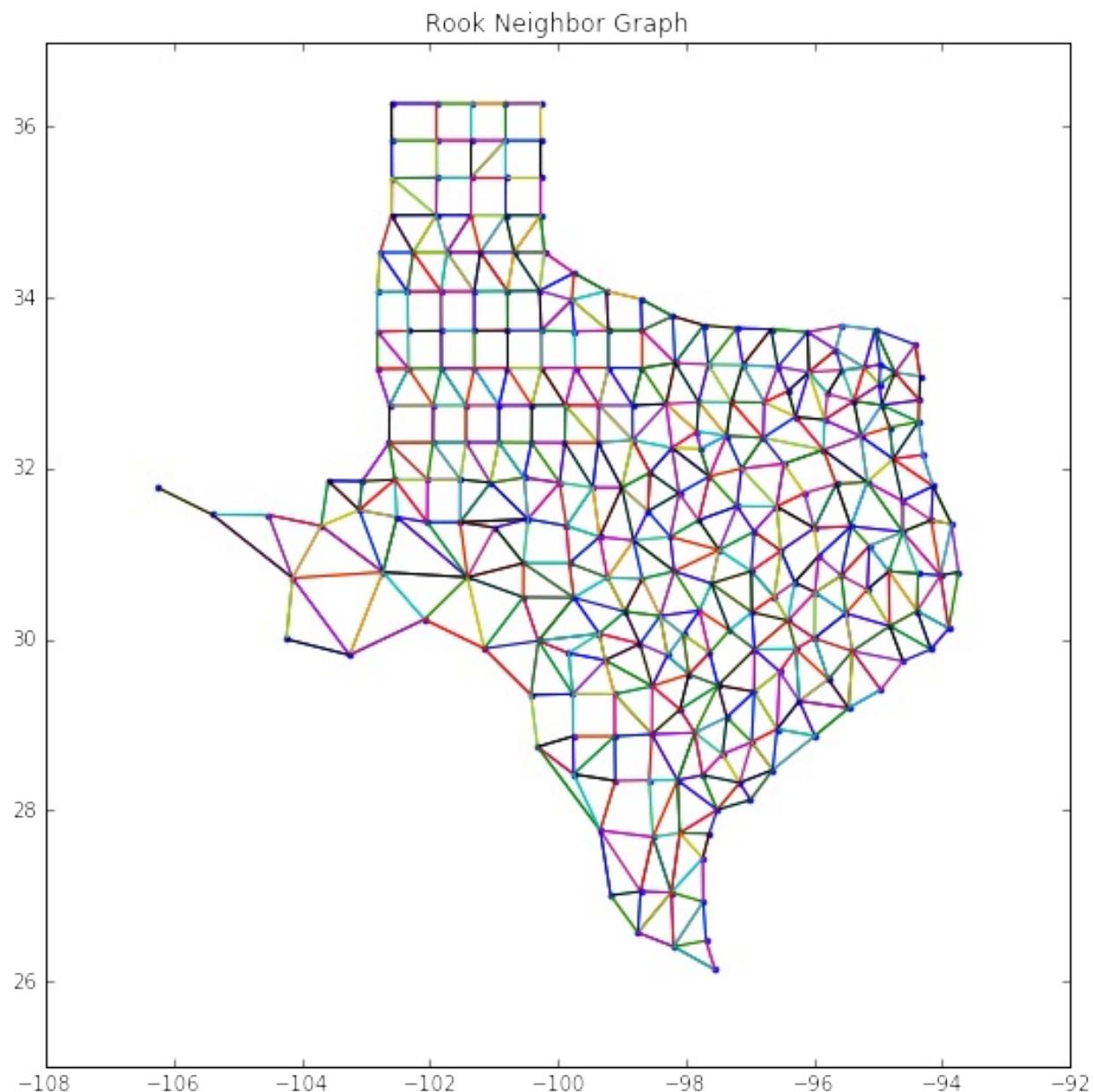


```
wr = ps.rook_from_shapefile('../data/texas.shp')
```

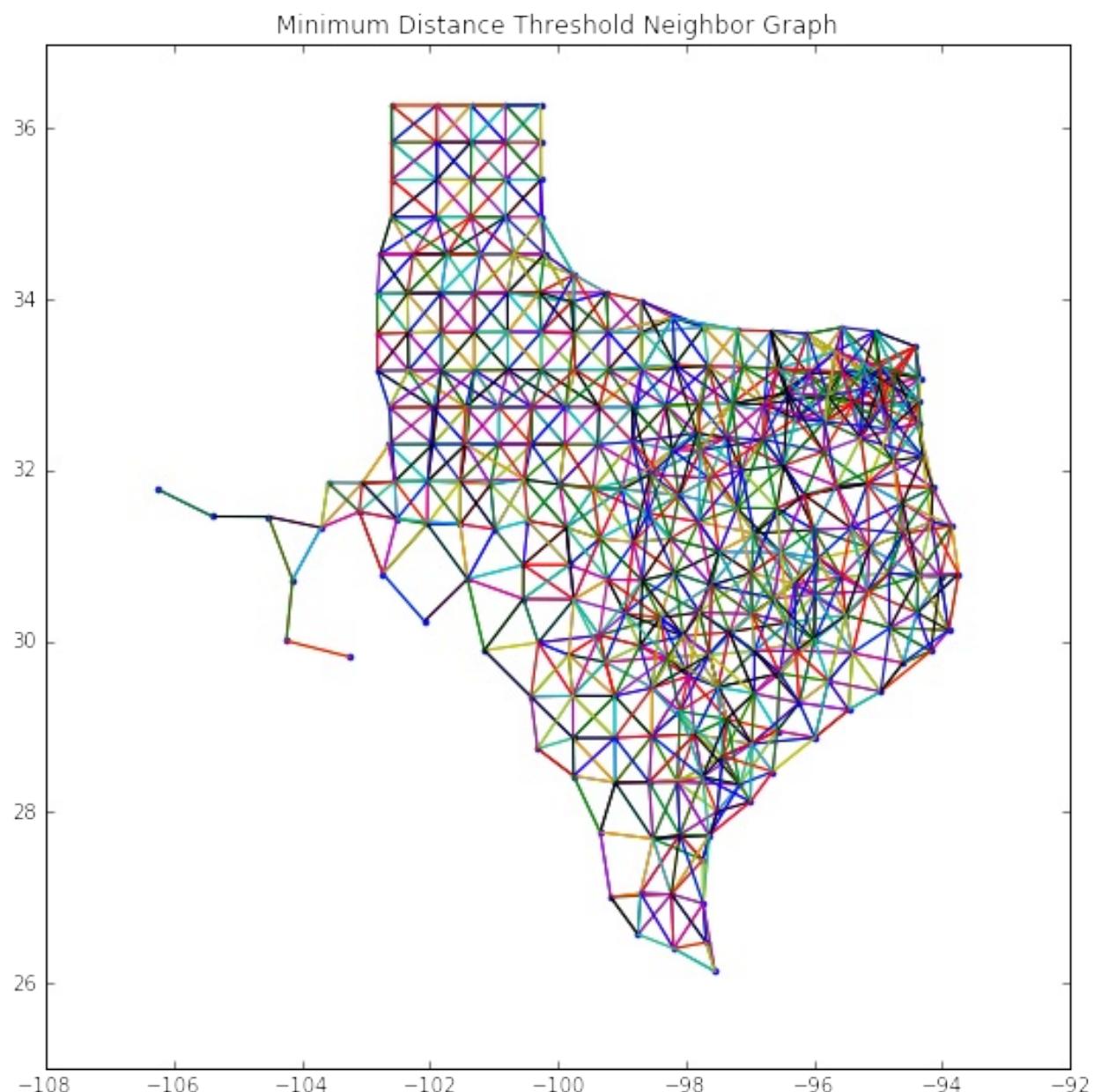
```
fig = figure(figsize=(9,9))

plt.plot(centroids[:,0], centroids[:,1], '.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in wr.neighbors.items():
    #print(k,neighs)
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')

plt.title('Rook Neighbor Graph')
show()
```



```
fig = figure(figsize=(9,9))
plt.plot(centroids[:,0], centroids[:,1], '.')
#plt.plot(s04[:,0], s04[:,1], '-')
plt.ylim([25,37])
for k,neighs in Wmind.neighbors.items():
    origin = centroids[k]
    for neigh in neighs:
        segment = centroids[[k,neigh]]
        plt.plot(segment[:,0], segment[:,1], '-')
plt.title('Minimum Distance Threshold Neighbor Graph')
show()
```



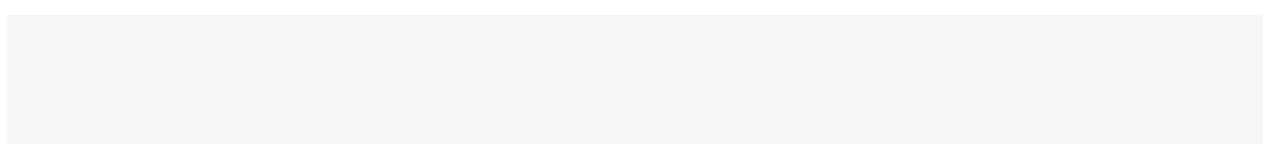
3.8378076756153514

wr.pct_nonzero

2.0243040486080974

wq.pct_nonzero

2.263004526009052



Exploratory Spatial Data Analysis (ESDA)

IPYNB

```
%matplotlib inline
import pysal as ps
import pandas as pd
import numpy as np
from pysal.contrib.viz import mapping as maps
```

A well-used functionality in PySAL is the use of PySAL to conduct exploratory spatial data analysis. This notebook will provide an overview of ways to conduct exploratory spatial analysis in Python.

First, let's read in some data:

```
data = ps.pdio.read_files("../data/texas.shp")
```

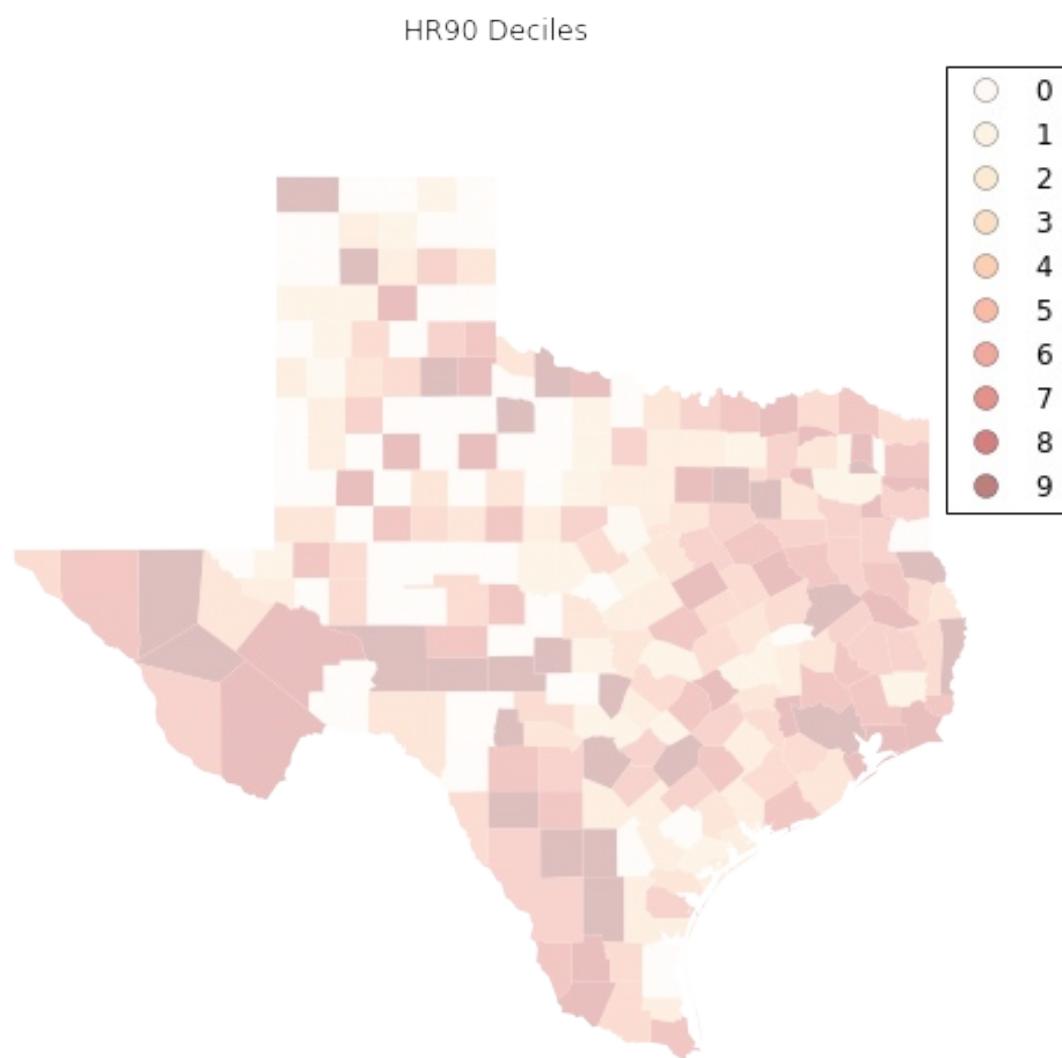
```
data.head()
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
0	Lipscomb	Texas	48	295	48295
1	Sherman	Texas	48	421	48421
2	Dallam	Texas	48	111	48111
3	Hansford	Texas	48	195	48195
4	Ochiltree	Texas	48	357	48357

5 rows × 70 columns

```
import matplotlib.pyplot as plt

import geopandas as gpd
shp_link = "../data/texas.shp"
tx = gpd.read_file(shp_link)
hr10 = ps.Quantiles(data.HR90, k=10)
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hr10.yb).plot(column='cl', categorical=True, \
    k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
    edgecolor='white', legend=True)
ax.set_axis_off()
plt.title("HR90 Deciles")
plt.show()
```



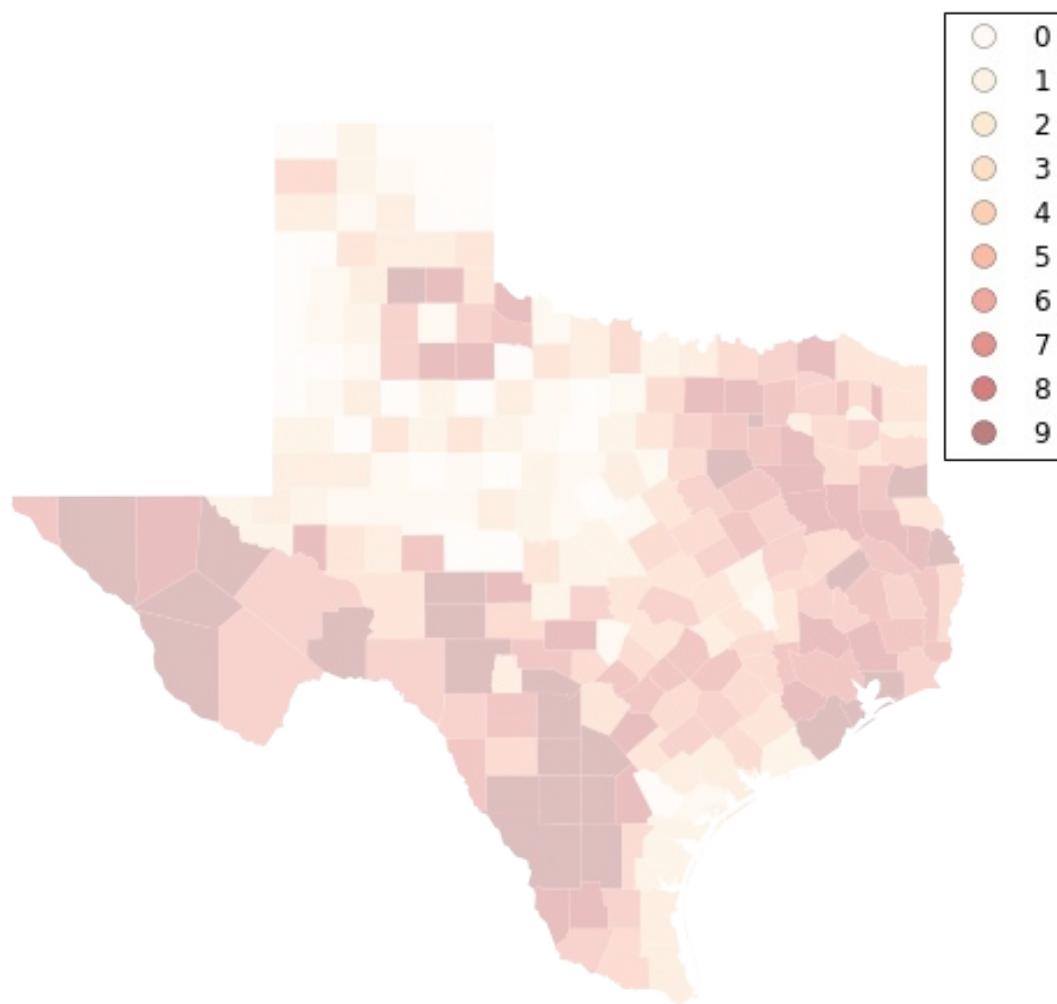
```
data = ps.pdio.read_files("../data/texas.shp")
W = ps.queen_from_shapefile("../data/texas.shp")
W.transform = 'r'
```

```
HR90Lag = ps.lag_spatial(W, data.HR90)
HR90LagQ10 = ps.Quantiles(HR90Lag, k=10)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=HR90LagQ10.yb).plot(column='cl', categorical=True, \
    k=10, cmap='OrRd', linewidth=0.1, ax=ax, \
    edgecolor='white', legend=True)
ax.set_axis_off()
plt.title("HR90 Spatial Lag Deciles")

plt.show()
```

HR90 Spatial Lag Deciles



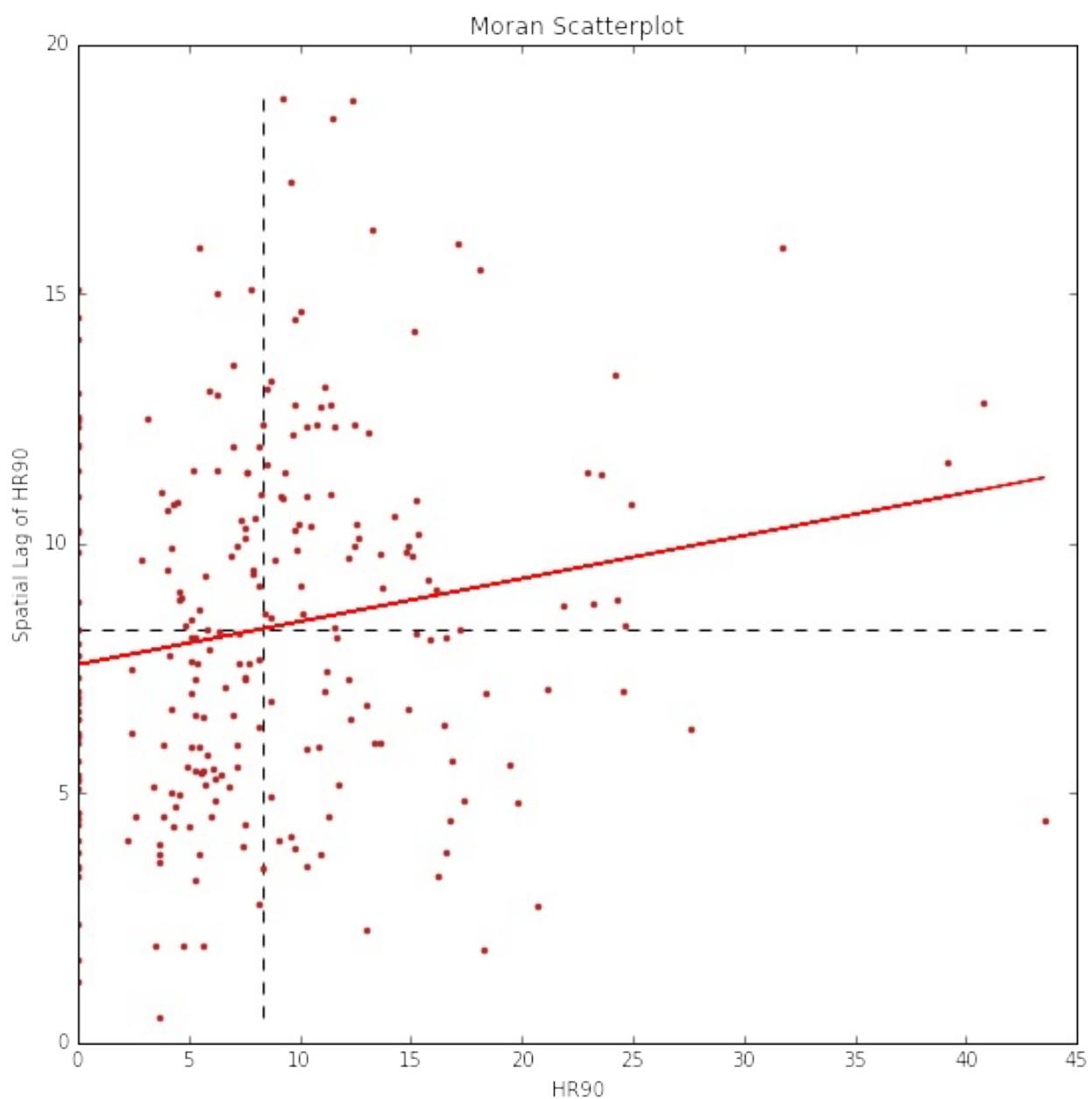
```
HR90 = data.HR90  
b,a = np.polyfit(HR90, HR90Lag, 1)
```

```
f, ax = plt.subplots(1, figsize=(9, 9))

plt.plot(HR90, HR90Lag, '.', color='firebrick')

# dashed vert at mean of the last year's PCI
plt.vlines(HR90.mean(), HR90Lag.min(), HR90Lag.max(), linestyle='--')
# dashed horizontal at mean of lagged PCI
plt.hlines(HR90Lag.mean(), HR90.min(), HR90.max(), linestyle='--')

# red line of best fit using global I as slope
plt.plot(HR90, a + b*HR90, 'r')
plt.title('Moran Scatterplot')
plt.ylabel('Spatial Lag of HR90')
plt.xlabel('HR90')
plt.show()
```



Global Spatial Autocorrelation

In PySAL, commonly-used analysis methods are very easy to access. For example, if we were interested in examining the spatial dependence in `HR90` we could quickly compute a Moran's $I\$$ statistic:

```
I_HR90 = ps.Moran(data.HR90.values, w)
```

```
I_HR90.I, I_HR90.p_sim
```

```
(0.085976640313889768, 0.007000000000000001)
```

Thus, the $I\$$ statistic is $0.859\$$ for this data, and has a very small $p\$$ value.

```
b # note I is same as the slope of the line in the scatterplot
```

```
0.085976640313889505
```

We can visualize the distribution of simulated $I\$$ statistics using the stored collection of simulated statistics:

```
I_HR90.sim[0:5]
```

```
array([ 0.02513765, -0.01225297,  0.0088749 , -0.04848781,  0.03966136])
```

A simple way to visualize this distribution is to make a KDEplot (like we've done before), and add a rug showing all of the simulated points, and a vertical line denoting the observed value of the statistic:

```
import matplotlib.pyplot as plt
import seaborn as sns
%matplotlib inline
```

```

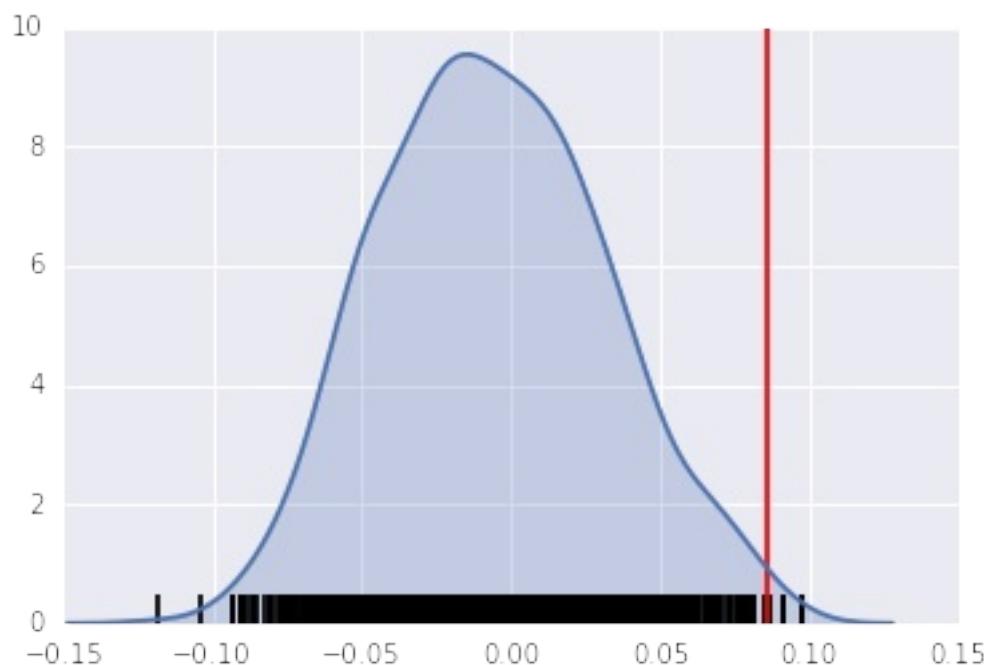
sns.kdeplot(I_HR90.sim, shade=True)
plt.vlines(I_HR90.sim, 0, 0.5)
plt.vlines(I_HR90.I, 0, 10, 'r')
plt.xlim([-0.15, 0.15])

```

/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i nstead of an integer will result in an error in the future

```
y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

```
(-0.15, 0.15)
```



Instead, if our $I\$$ statistic were close to our expected value, `I_HR90.EI`, our plot might look like this:

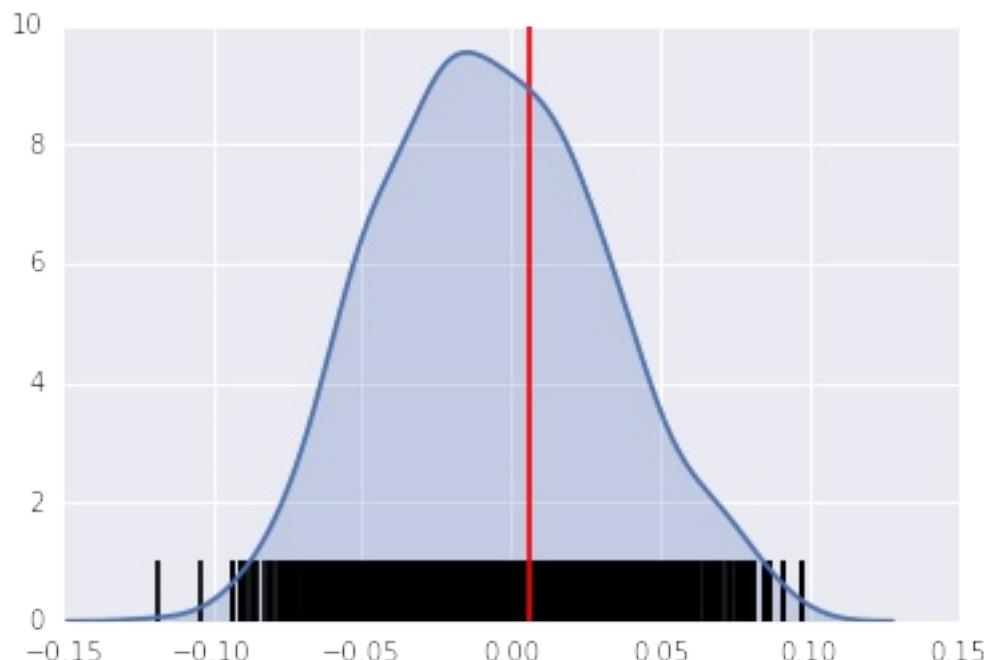
```

sns.kdeplot(I_HR90.sim, shade=True)
plt.vlines(I_HR90.sim, 0, 1)
plt.vlines(I_HR90.EI+.01, 0, 10, 'r')
plt.xlim([-0.15, 0.15])

```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non
parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i
nstead of an integer will result in an error in the future
y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

(-0.15, 0.15)



In addition, we can compute a global Bivariate Moran statistic, which relates an observation to the spatial lag of another observation:

```
bv_HRBLK = ps.Moran_BV(data.HR90.values, data.BLK90.values, W)
```

```
bv_HRBLK.I, bv_HRBLK.p_sim
```

(0.080041444794258343, 0.007000000000000001)

Local Autocorrelation Statistics

In addition to the Global autocorrelation statistics, PySAL has many local autocorrelation statistics. Let's compute a local Moran statistic for the same data shown above:

```
LMo_HR90 = ps.Moran_Local(data.HR90.values, w)
```

Now, instead of a single \$I\$ statistic, we have an *array* of local \$I_i\$ statistics, stored in the `.Is` attribute, and p-values from the simulation are in `p_sim`.

```
LMo_HR90.Is, LMo_HR90.p_sim
```

```
(array([
  1.12087323e+00, 4.74852232e-01, -1.22758423e+00,
  9.38686608e-01, 6.89742960e-01, 7.85031726e-01,
  7.10475151e-01, 4.10606860e-01, 7.40368367e-03,
  1.48663520e-01, -5.99072474e-01, 8.11752611e-02,
 -1.20482997e-01, 3.18800252e-01, 3.47276792e-01,
  4.39609038e-02, -3.04514513e-01, 2.01610062e-01,
  2.85061414e-01, 5.84462378e-01, 7.56336432e-01,
  3.83077442e-01, 3.99898754e-03, -9.16855756e-01,
 -4.87000938e-02, 1.46381951e-02, -1.20220927e-01,
 -7.65176192e-01, -2.57853261e+00, 2.03465774e-01,
 -1.00025313e-02, 1.51706786e-01, 4.93433817e-01,
  4.35124305e-01, -3.12259940e-01, -2.32723324e-01,
  2.82455741e-03, 1.22002377e-01, -1.27741850e-02,
 -3.54088100e-04, -9.99830522e-03, -4.29411693e-02,
  1.14955949e-01, -5.76707109e-01, -5.79367567e-01,
 -2.44357064e-01, -2.45937883e-02, 1.28248005e-01,
  2.40904907e-01, -1.30657056e+00, 8.75684587e-02,
  7.60257313e-01, 7.46769544e-03, 8.46525604e-02,
 -9.91242446e-02, -3.63476666e-03, -1.84994288e-01,
  6.10806717e-02, -4.10962997e-01, -1.55435599e-02,
  3.34932522e-01, 5.84998647e-01, 3.97457030e-01,
  4.84549106e-01, -7.05237462e-01, -1.89267505e-01,
 -5.36300770e-01, 2.18228666e-01, 2.45387767e-01,
  1.04747657e+00, -5.03740235e-02, -4.99501600e-01,
 -6.79039774e-02, -4.28204631e-01, -5.19255631e-02,
  1.62667971e-01, 3.96182300e-02, 1.39001221e-01,
  7.85033384e-01, -6.34873174e-02, -2.05239166e-01,
  6.70092095e-01, 1.19432435e-01, 8.33565709e-02,
  6.08109031e-02, 4.71129836e-02, -7.53103370e-01,
  4.12179684e-01, 4.18690212e-01, -1.24823445e-01,
 -2.20055897e-01, 8.71366261e-01, -8.79710509e-02,
  1.68820593e-03, 1.07166595e-02, -3.54370029e-02,
  1.20592772e-03, 2.34359115e-01, 1.74921357e-01,
 -1.82542919e-01, -2.17921039e-01, -1.29920391e-03,
 -5.76690555e-02, 1.45723641e-01, 3.62521949e-01,
  4.39212267e-02, 1.62791100e-01, 2.34754317e-01,
  4.24252755e-02, -7.44382557e-01, 1.01380671e-01,
```

```

 1.68446607e-01, 1.36810620e-01, 4.56977527e-02,
 6.92868233e-02, 3.82546883e-02, 1.23206557e-01,
 5.81014061e-01, 6.19044577e-01, 5.97840100e-02,
 6.02552478e-01, 1.98331508e-01, -2.08124490e-01,
 2.89307856e-01, 2.94694807e-01, 3.17901639e-01,
 3.26156089e-01, 3.01034558e-01, 1.49504893e-01,
 1.95148623e+00, -2.54303828e-01, 5.08956169e-01,
 6.08264082e-01, -3.81888449e-02, -3.00282633e-01,
 -2.65739969e-02, 2.24230408e-01, -1.44008615e-02,
 4.66594113e-01, -4.31489357e-02, 1.87061533e-02,
 1.45607307e-01, 2.47182058e-03, -6.64057756e-01,
 1.54192078e-02, 2.36636263e-01, -8.45056351e-03,
 -1.34851321e-01, 1.18537899e-01, -2.27964598e-01,
 -4.18648686e-01, 3.39507930e-01, -2.63719989e-02,
 1.55681623e-01, 1.22301226e-01, 1.77438253e-01,
 4.10677287e-02, 1.49101636e-01, 1.04297773e-01,
 -7.67634652e-02, -5.64248729e-03, -3.05586580e-02,
 5.53503759e-02, 1.37140043e-02, 1.29212068e+00,
 -8.56707799e-02, 2.62786886e-01, -6.36623531e-01,
 -3.90746796e-01, 2.74114791e-02, -3.42998234e-02,
 1.96470454e-02, 2.66085647e-01, -5.86103398e-01,
 -1.40050219e-01, -9.58183688e-02, 1.06002105e-01,
 5.78305864e-02, 9.12034797e-02, 1.75316202e-01,
 1.53146979e+00, 2.55223569e-01, -1.07521719e+00,
 2.10421553e-01, 1.79559685e-03, 2.46623711e-01,
 -1.37221563e-01, -1.93104397e-01, -6.70944099e-01,
 -6.21994147e-01, 2.62134357e-01, 8.44258361e-02,
 2.15034457e-03, -1.13695914e-02, -9.85881978e-01,
 -3.88051312e-02, -3.94113412e-02, 2.54665339e-01,
 1.03033517e-01, 1.87014888e-01, 6.57921237e-02,
 8.93885114e-01, -4.12463002e-02, 5.93523798e-04,
 -7.48641130e-01, -2.33977229e-01, -1.81364976e-01,
 -2.12719096e-02, -1.77624803e-01, 7.56458111e-01,
 4.65743492e-02, -1.17037830e-02, 1.42744115e-01,
 -2.52132245e-01, 2.20273885e-02, 1.32249407e-01,
 1.50840035e-02, 1.58872364e-01, -3.07330315e-01,
 -2.15826555e-01, 7.78727321e-01, 1.53798953e-01,
 1.39830232e-02, 1.14430614e-02, -4.13603775e-01,
 -1.13601903e-01, 3.98036147e-02, 5.00495648e-02,
 1.85345062e-02, 8.20513963e-01, -2.85154006e-03,
 3.37194545e-01, -6.54881438e-01, 2.87571205e-01,
 3.40025967e+00, 2.79223250e+00, 2.18713536e-01,
 2.57852929e-01, 1.00405907e-01, 1.92499398e-01,
 9.07540052e-02, 1.49438551e-01, 1.33219486e+00,
 9.58722126e-03, -9.26242513e-02, 1.26208606e-01,
 3.53039195e-01, 1.91842139e-01, -2.84025227e-03,
 2.62219645e-01, 1.87256013e-01, 2.39837728e-03,
 8.94400168e-02, -1.39329420e-01]),
array([ 0.009,  0.149,  0.031,  0.016,  0.001,  0.013,  0.06 ,  0.053,
       0.488,  0.104,  0.079,  0.357,  0.033,  0.01 ,  0.224,  0.467,
       0.238,  0.36 ,  0.268,  0.016,  0.065,  0.056,  0.254,  0.047,
       0.457,  0.142,  0.184,  0.136,  0.057,  0.229,  0.293,  0.111,

```

```

0.032, 0.015, 0.216, 0.276, 0.476, 0.233, 0.487, 0.46 ,
0.376, 0.177, 0.358, 0.091, 0.073, 0.249, 0.075, 0.281,
0.027, 0.011, 0.445, 0.029, 0.466, 0.347, 0.046, 0.446,
0.125, 0.109, 0.087, 0.446, 0.218, 0.067, 0.037, 0.126,
0.022, 0.18 , 0.003, 0.313, 0.055, 0.024, 0.323, 0.15 ,
0.443, 0.213, 0.377, 0.144, 0.462, 0.358, 0.184, 0.054,
0.369, 0.039, 0.131, 0.458, 0.142, 0.488, 0.016, 0.041,
0.18 , 0.251, 0.291, 0.146, 0.117, 0.462, 0.399, 0.498,
0.467, 0.141, 0.023, 0.025, 0.066, 0.023, 0.026, 0.154,
0.191, 0.264, 0.078, 0.242, 0.148, 0.122, 0.081, 0.194,
0.171, 0.25 , 0.064, 0.366, 0.066, 0.087, 0.069, 0.057,
0.108, 0.273, 0.127, 0.29 , 0.122, 0.071, 0.085, 0.065,
0.26 , 0.161, 0.032, 0.223, 0.01 , 0.216, 0.382, 0.473,
0.093, 0.216, 0.155, 0.355, 0.005, 0.21 , 0.122, 0.102,
0.381, 0.28 , 0.325, 0.075, 0.397, 0.039, 0.181, 0.133,
0.265, 0.431, 0.13 , 0.162, 0.099, 0.258, 0.125, 0.395,
0.426, 0.216, 0.222, 0.479, 0.017, 0.367, 0.063, 0.094,
0.325, 0.344, 0.235, 0.467, 0.108, 0.19 , 0.292, 0.304,
0.288, 0.418, 0.447, 0.379, 0.075, 0.254, 0.055, 0.06 ,
0.436, 0.094, 0.337, 0.198, 0.085, 0.057, 0.059, 0.233,
0.445, 0.413, 0.023, 0.322, 0.204, 0.205, 0.2 , 0.301,
0.168, 0.127, 0.403, 0.429, 0.325, 0.17 , 0.191, 0.397,
0.068, 0.026, 0.159, 0.123, 0.367, 0.33 , 0.08 , 0.256,
0.409, 0.141, 0.244, 0.084, 0.089, 0.125, 0.297, 0.48 ,
0.004, 0.259, 0.466, 0.211, 0.404, 0.001, 0.205, 0.238,
0.069, 0.042, 0.013, 0.058, 0.017, 0.104, 0.331, 0.002,
0.132, 0.236, 0.018, 0.48 , 0.313, 0.259, 0.108, 0.129,
0.331, 0.338, 0.29 , 0.473, 0.382, 0.403]))
```

We can adjust the number of permutations used to derive every *pseudo-\$p\$* value by passing a different `permutations` argument:

```
LMo_HR90 = ps.Moran_Local(data.HR90.values, w, permutations=9999)
```

In addition to the typical clustermap, a helpful visualization for LISA statistics is a Moran scatterplot with statistically significant LISA values highlighted.

This is very simple, if we use the same strategy we used before:

First, construct the spatial lag of the covariate:

```
Lag_HR90 = ps.lag_spatial(w, data.HR90.values)
HR90 = data.HR90.values
```

Then, we want to plot the statistically-significant LISA values in a different color than the others. To do this, first find all of the statistically significant LISAs. Since the `p`-values are in the same order as the `I_i` statistics, we can do this in the following way

```
sigs = HR90[LMo_HR90.p_sim <= .001]
W_sigs = Lag_HR90[LMo_HR90.p_sim <= .001]
insigs = HR90[LMo_HR90.p_sim > .001]
W_insigs = Lag_HR90[LMo_HR90.p_sim > .001]
```

Then, since we have a lot of points, we can plot the points with a statistically insignificant LISA value lighter using the `alpha` keyword. In addition, we would like to plot the statistically significant points in a dark red color.

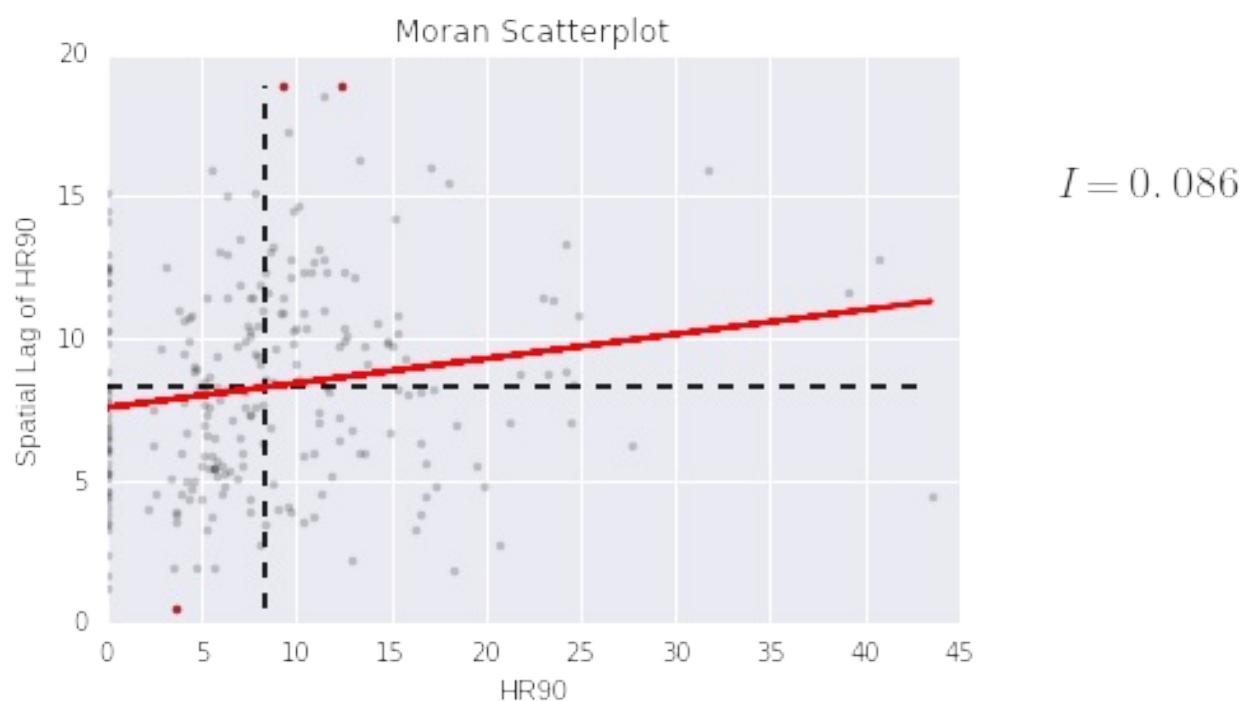
```
b,a = np.polyfit(HR90, Lag_HR90, 1)
```

Matplotlib has a list of [named colors](#) and will interpret colors that are provided in hexadecimal strings:

```
plt.plot(sigs, W_sigs, '.', color='firebrick')
plt.plot(insigs, W_insigs, '.k', alpha=.2)
# dashed vert at mean of the last year's PCI
plt.vlines(HR90.mean(), Lag_HR90.min(), Lag_HR90.max(), linestyle='--')
# dashed horizontal at mean of lagged PCI
plt.hlines(Lag_HR90.mean(), HR90.min(), HR90.max(), linestyle='--')

# red line of best fit using global I as slope
plt.plot(HR90, a + b*HR90, 'r')
plt.text(s='I = %.3f' % I_HR90.I, x=50, y=15, fontsize=18)
plt.title('Moran Scatterplot')
plt.ylabel('Spatial Lag of HR90')
plt.xlabel('HR90')
```

```
<matplotlib.text.Text at 0x7f6e70b434e0>
```



We can also make a LISA map of the data.

```
sig = LMo_HR90.p_sim < 0.05
```

```
sig.sum()
```

```
43
```

```
hotspots = LMo_HR90.q==1 * sig
```

```
hotspots.sum()
```

```
9
```

```
coldspots = LMo_HR90.q==3 * sig
```

```
coldspots.sum()
```

17

```
data.HR90[hotspots]
```

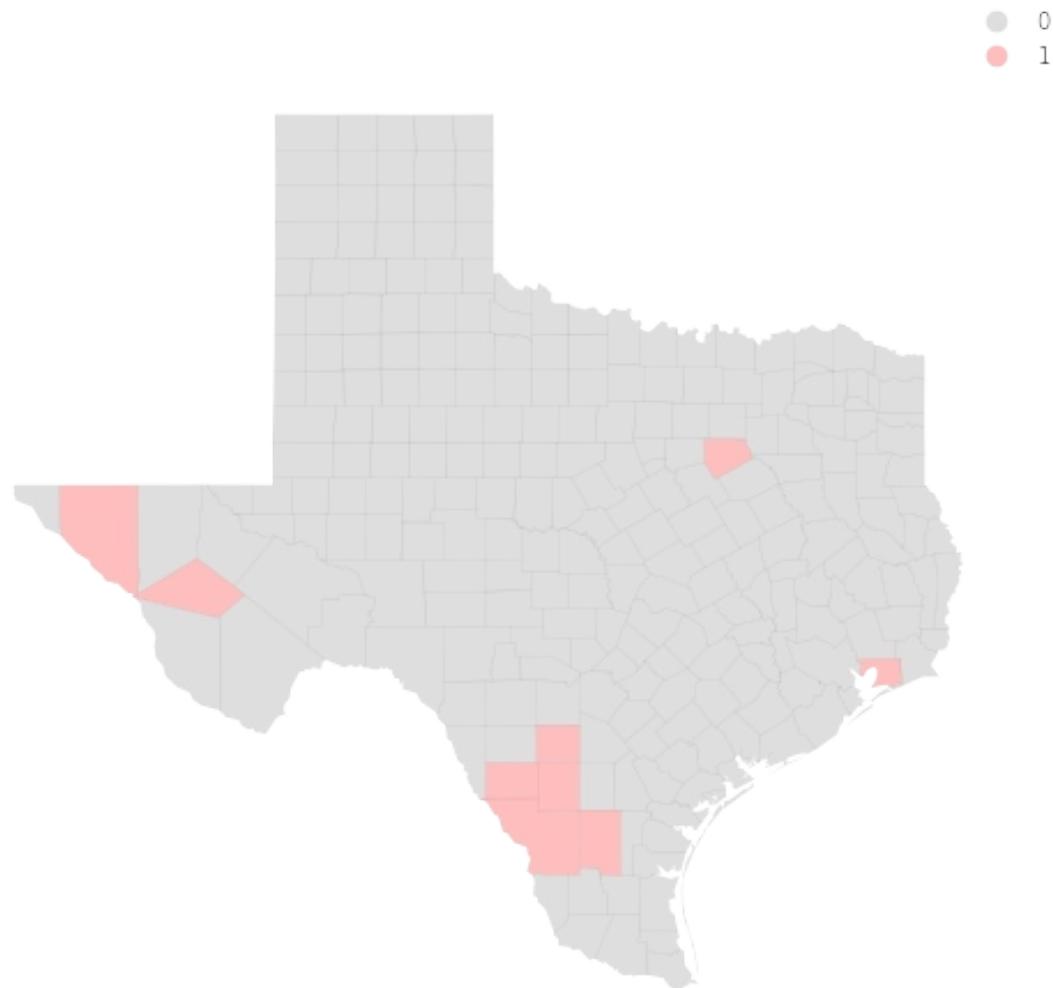
```
98      9.784698
132     11.435106
164     17.129154
209     13.274924
229     12.371338
234     31.721863
236     9.584971
239     9.256549
242     18.062652
Name: HR90, dtype: float64
```

```
data[hotspots]
```

	NAME	STATE_NAME	STATE_FIPS	CNTY_FIPS	FIPS
98	Ellis	Texas	48	139	48139
132	Hudspeth	Texas	48	229	48229
164	Jeff Davis	Texas	48	243	48243
209	Chambers	Texas	48	071	48071
229	Frio	Texas	48	163	48163
234	La Salle	Texas	48	283	48283
236	Dimmit	Texas	48	127	48127
239	Webb	Texas	48	479	48479
242	Duval	Texas	48	131	48131

9 rows × 70 columns

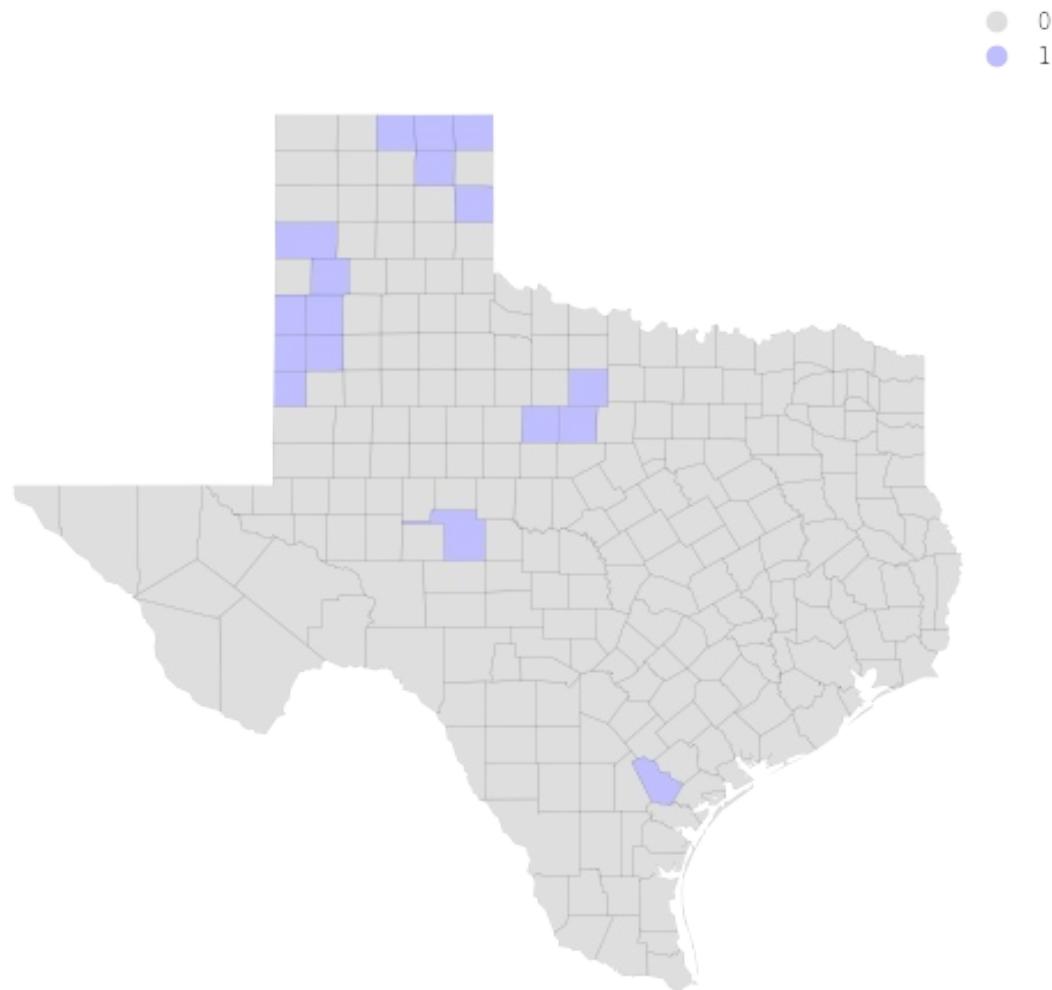
```
from matplotlib import colors
hmap = colors.ListedColormap(['grey', 'red'])
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hotspots*1).plot(column='cl', categorical=True, \
    k=2, cmap=hmap, linewidth=0.1, ax=ax, \
    edgecolor='grey', legend=True)
ax.set_axis_off()
plt.show()
```



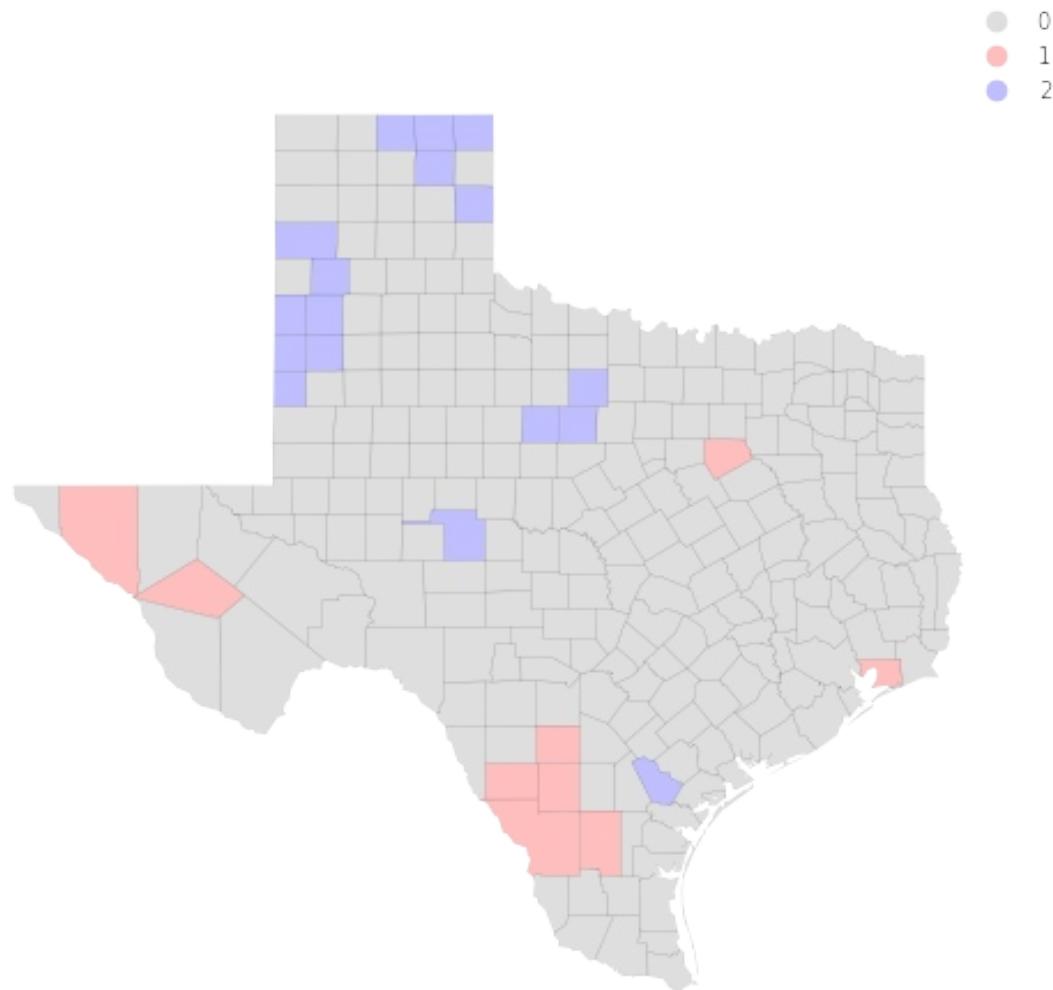
```
data.HR90[coldspots]
```

```
0      0.000000
3      0.000000
4      3.651767
5      0.000000
13     5.669899
19     3.480743
21     3.675119
32     2.211607
33     4.718762
48     5.509870
51     0.000000
62     3.677958
69     0.000000
81     0.000000
87     3.699593
140    8.125292
233    5.304688
Name: HR90, dtype: float64
```

```
cmap = colors.ListedColormap(['grey', 'blue'])
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=coldspots*1).plot(column='cl', categorical=True, \
    k=2, cmap=cmap, linewidth=0.1, ax=ax, \
    edgecolor='black', legend=True)
ax.set_axis_off()
plt.show()
```



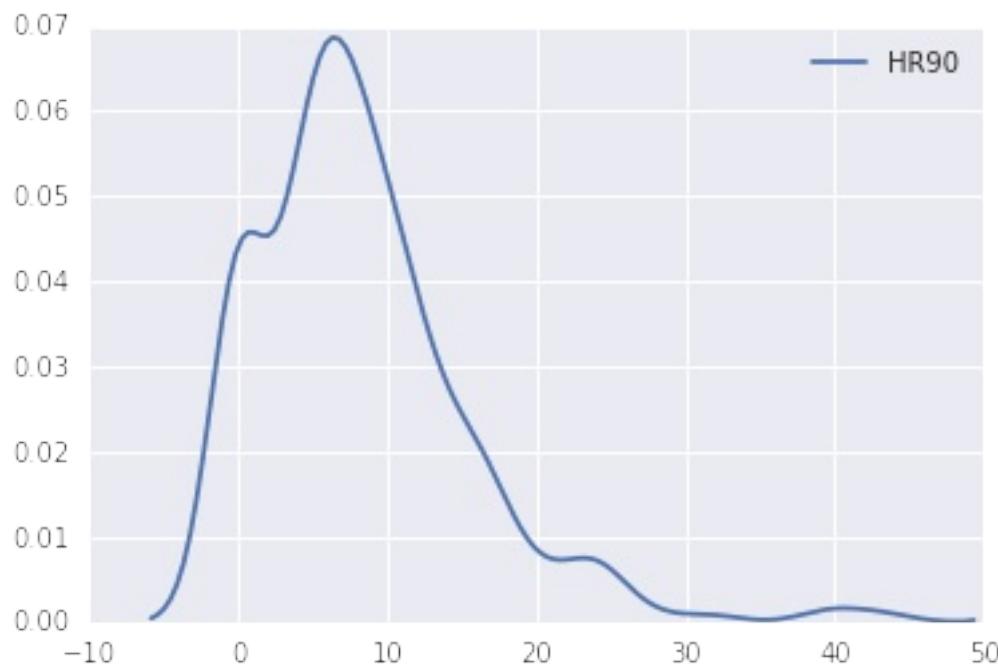
```
from matplotlib import colors
hcmap = colors.ListedColormap(['grey', 'red','blue'])
hotcold = hotspots*1 + coldspots*2
f, ax = plt.subplots(1, figsize=(9, 9))
tx.assign(cl=hotcold).plot(column='cl', categorical=True, \
    k=2, cmap=hcmap, linewidth=0.1, ax=ax, \
    edgecolor='black', legend=True)
ax.set_axis_off()
plt.show()
```



```
sns.kdeplot(data.HR90)
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non
parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i
nstead of an integer will result in an error in the future
y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

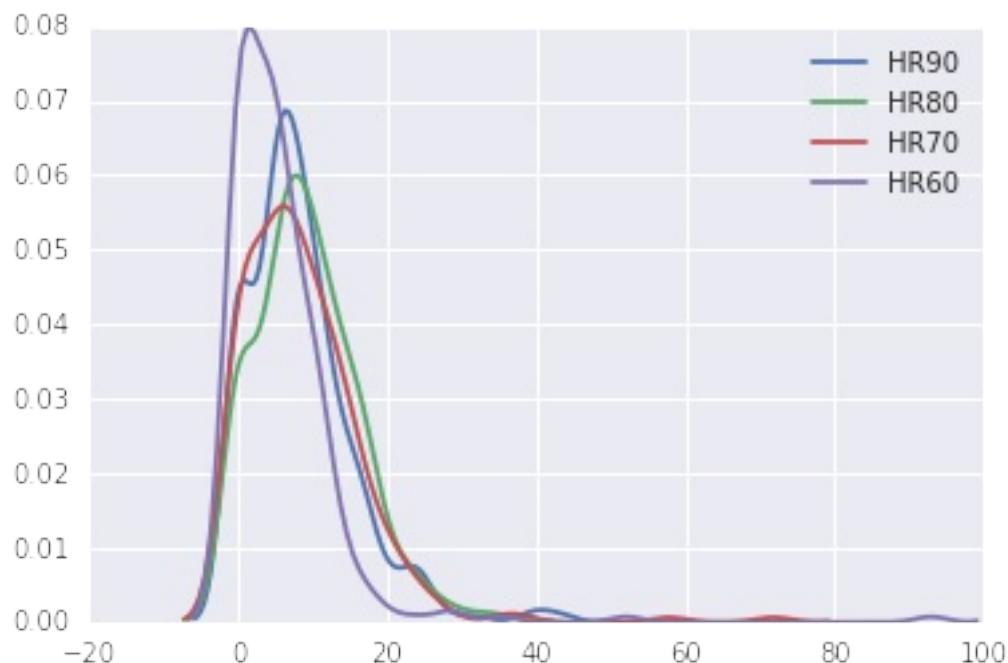
```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6e6e147550>
```



```
sns.kdeplot(data.HR90)
sns.kdeplot(data.HR80)
sns.kdeplot(data.HR70)
sns.kdeplot(data.HR60)
```

```
/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non
parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i
nstead of an integer will result in an error in the future
y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j
```

```
<matplotlib.axes._subplots.AxesSubplot at 0x7f6e6e5ccdd8>
```



```
data.HR90.mean()
```

```
8.302494460285041
```

```
data.HR90.median()
```

```
7.23234613355
```


Exploratory Spatial and Temporal Data Analysis (ESTDA)

IPYNB

```
import matplotlib
import numpy as np
import pysal as ps
import matplotlib.pyplot as plt
%matplotlib inline
```

```
f = ps.open(ps.examples.get_path('usjoin.csv'), 'r')
```

To determine what is in the file, check the `header` attribute on the file object:

```
f.header
```

```
[ 'Name',
  'STATE_FIPS',
  '1929',
  '1930',
  '1931',
  '1932',
  '1933',
  '1934',
  '1935',
  '1936',
  '1937',
  '1938',
  '1939',
  '1940',
  '1941',
  '1942',
  '1943',
  '1944',
  '1945',
  '1946',
  '1947',
  '1948',
  '1949',
  '1950',
  '1951',
```

```
'1952',
'1953',
'1954',
'1955',
'1956',
'1957',
'1958',
'1959',
'1960',
'1961',
'1962',
'1963',
'1964',
'1965',
'1966',
'1967',
'1968',
'1969',
'1970',
'1971',
'1972',
'1973',
'1974',
'1975',
'1976',
'1977',
'1978',
'1979',
'1980',
'1981',
'1982',
'1983',
'1984',
'1985',
'1986',
'1987',
'1988',
'1989',
'1990',
'1991',
'1992',
'1993',
'1994',
'1995',
'1996',
'1997',
'1998',
'1999',
'2000',
'2001',
'2002',
'2003',
```

```
'2004',
'2005',
'2006',
'2007',
'2008',
'2009']
```

Ok, lets pull in the `name` variable to see what we have.

```
name = f.by_col('Name')
```

```
name
```

```
[ 'Alabama',
  'Arizona',
  'Arkansas',
  'California',
  'Colorado',
  'Connecticut',
  'Delaware',
  'Florida',
  'Georgia',
  'Idaho',
  'Illinois',
  'Indiana',
  'Iowa',
  'Kansas',
  'Kentucky',
  'Louisiana',
  'Maine',
  'Maryland',
  'Massachusetts',
  'Michigan',
  'Minnesota',
  'Mississippi',
  'Missouri',
  'Montana',
  'Nebraska',
  'Nevada',
  'New Hampshire',
  'New Jersey',
  'New Mexico',
  'New York',
  'North Carolina',
  'North Dakota',
  'Ohio',
  'Oklahoma',
  'Oregon',
  'Pennsylvania',
  'Rhode Island',
  'South Carolina',
  'South Dakota',
  'Tennessee',
  'Texas',
  'Utah',
  'Vermont',
  'Virginia',
  'Washington',
  'West Virginia',
  'Wisconsin',
  'Wyoming']
```

Now obtain per capital incomes in 1929 which is in the column associated with 1929 .

```
y1929 = f.by_col('1929')
```

```
y1929
```

```
[323,  
 600,  
 310,  
 991,  
 634,  
 1024,  
 1032,  
 518,  
 347,  
 507,  
 948,  
 607,  
 581,  
 532,  
 393,  
 414,  
 601,  
 768,  
 906,  
 790,  
 599,  
 286,  
 621,  
 592,  
 596,  
 868,  
 686,  
 918,  
 410,  
 1152,  
 332,  
 382,  
 771,  
 455,  
 668,  
 772,  
 874,  
 271,  
 426,  
 378,  
 479,  
 551,  
 634,  
 434,  
 741,  
 460,  
 673,  
 675]
```

And now 2009

```
y2009 = f.by_col("2009")
```

```
y2009
```

```
[32274,  
 32077,  
 31493,  
 40902,  
 40093,  
 52736,  
 40135,  
 36565,  
 33086,  
 30987,  
 40933,  
 33174,  
 35983,  
 37036,  
 31250,  
 35151,  
 35268,  
 47159,  
 49590,  
 34280,  
 40920,  
 29318,  
 35106,  
 32699,  
 37057,  
 38009,  
 41882,  
 48123,  
 32197,  
 46844,  
 33564,  
 38672,  
 35018,  
 33708,  
 35210,  
 38827,  
 41283,  
 30835,  
 36499,  
 33512,  
 35674,  
 30107,  
 36752,  
 43211,  
 40619,  
 31843,  
 35676,  
 42504]
```

These are read into regular Python lists which are not particularly well suited to efficient data analysis. So let's convert them to numpy arrays.

```
y2009 = np.array(y2009)
```

```
y2009
```

```
array([32274, 32077, 31493, 40902, 40093, 52736, 40135, 36565, 33086,
       30987, 40933, 33174, 35983, 37036, 31250, 35151, 35268, 47159,
       49590, 34280, 40920, 29318, 35106, 32699, 37057, 38009, 41882,
       48123, 32197, 46844, 33564, 38672, 35018, 33708, 35210, 38827,
       41283, 30835, 36499, 33512, 35674, 30107, 36752, 43211, 40619,
       31843, 35676, 42504])
```

Much better. But pulling these in and converting them a column at a time is tedious and error prone. So we will do all of this in a list comprehension.

```
Y = np.array( [ f.by_col(str(year)) for year in range(1929,2010) ] ) * 1.0
```

```
Y.shape
```

```
(81, 48)
```

```
Y = Y.transpose()
```

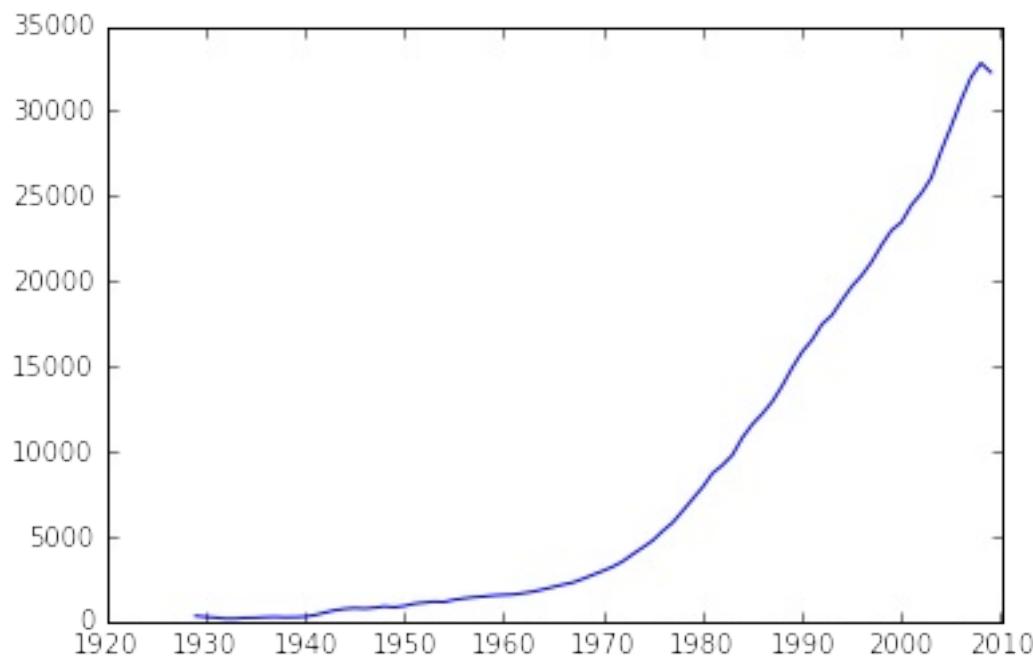
```
Y.shape
```

```
(48, 81)
```

```
years = np.arange(1929,2010)
```

```
plt.plot(years,Y[0])
```

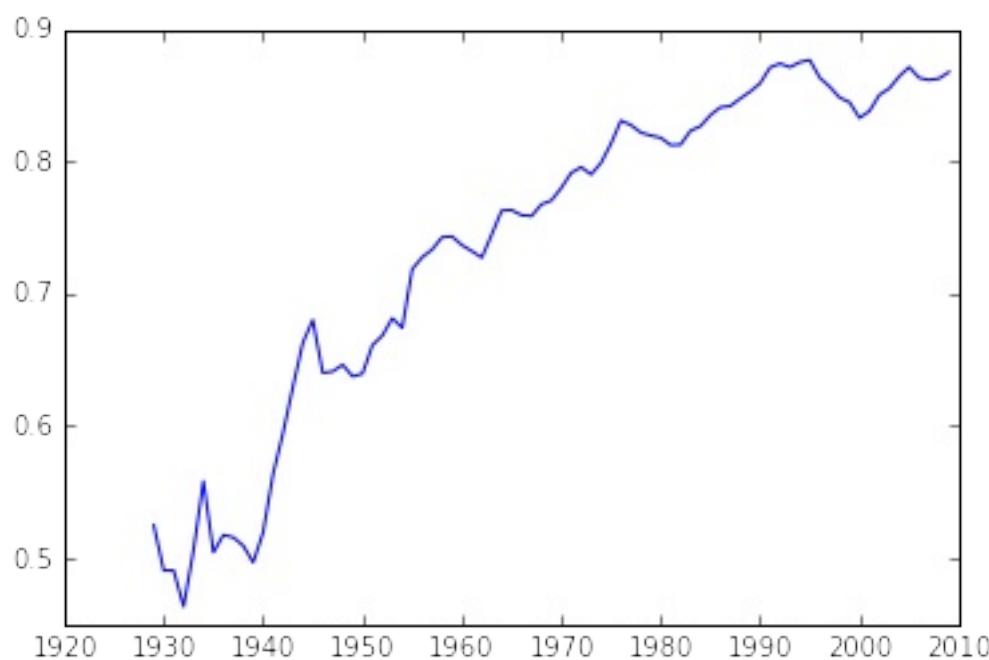
```
[<matplotlib.lines.Line2D at 0x7fc12c98b7b8>]
```



```
RY = Y / Y.mean(axis=0)
```

```
plt.plot(years, RY[0])
```

```
[<matplotlib.lines.Line2D at 0x7fc12c8f0c18>]
```



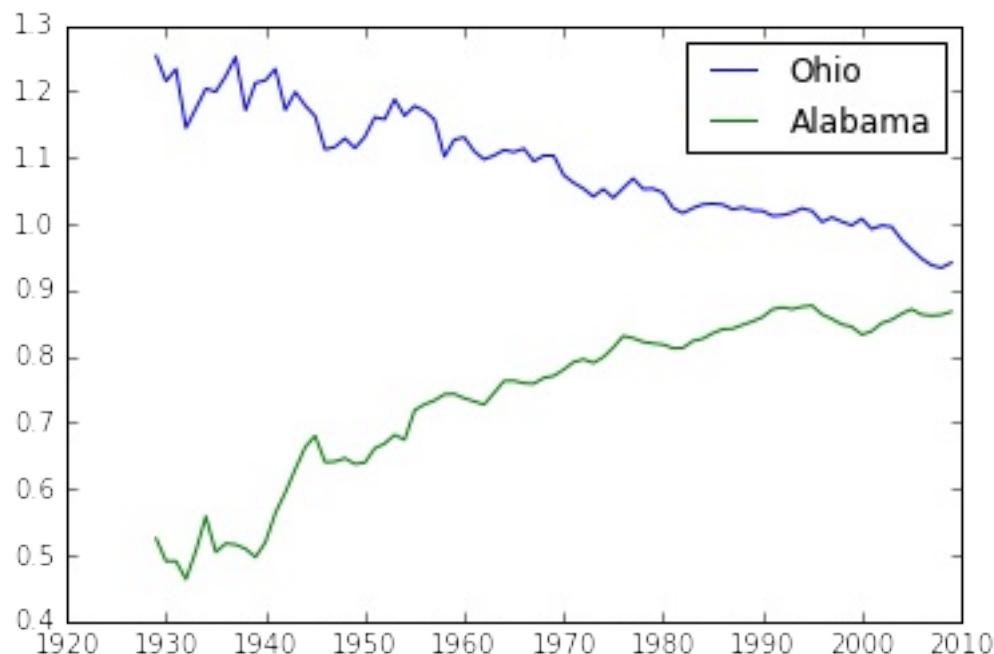
```
name = np.array(name)
```

```
np.nonzero(name=='Ohio')
```

```
(array([32]),)
```

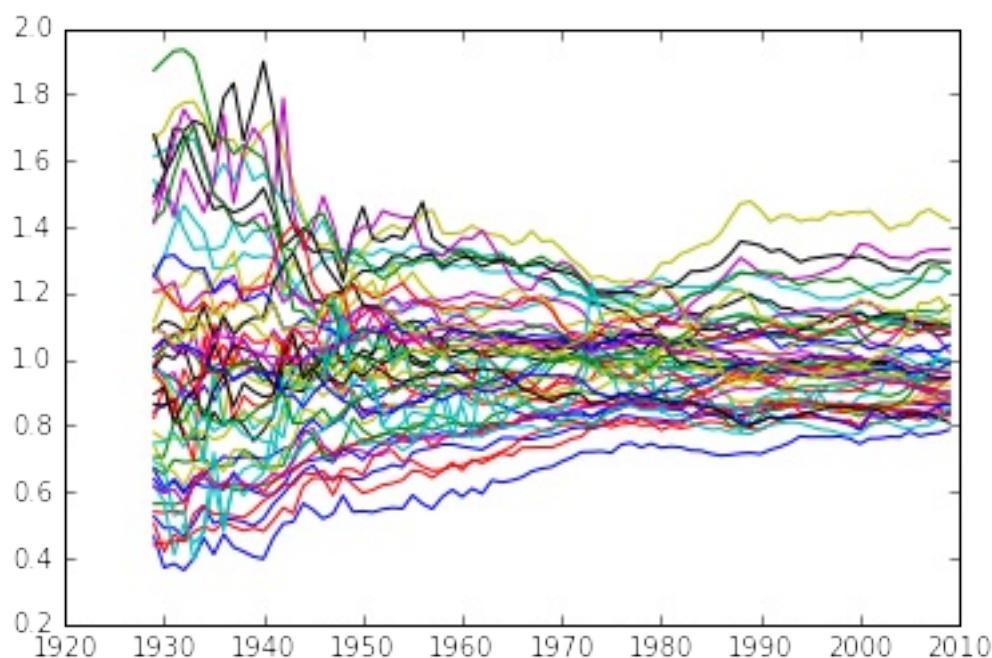
```
plt.plot(years, RY[32], label='Ohio')
plt.plot(years, RY[0], label='Alabama')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc12ad18518>
```



Spaghetti Plot

```
for row in RY:
    plt.plot(years, row)
```



Kernel Density (univariate, aspatial)

```
from scipy.stats.kde import gaussian_kde
```

```
density = gaussian_kde(Y[:,0])
```

```
Y[:,0]
```

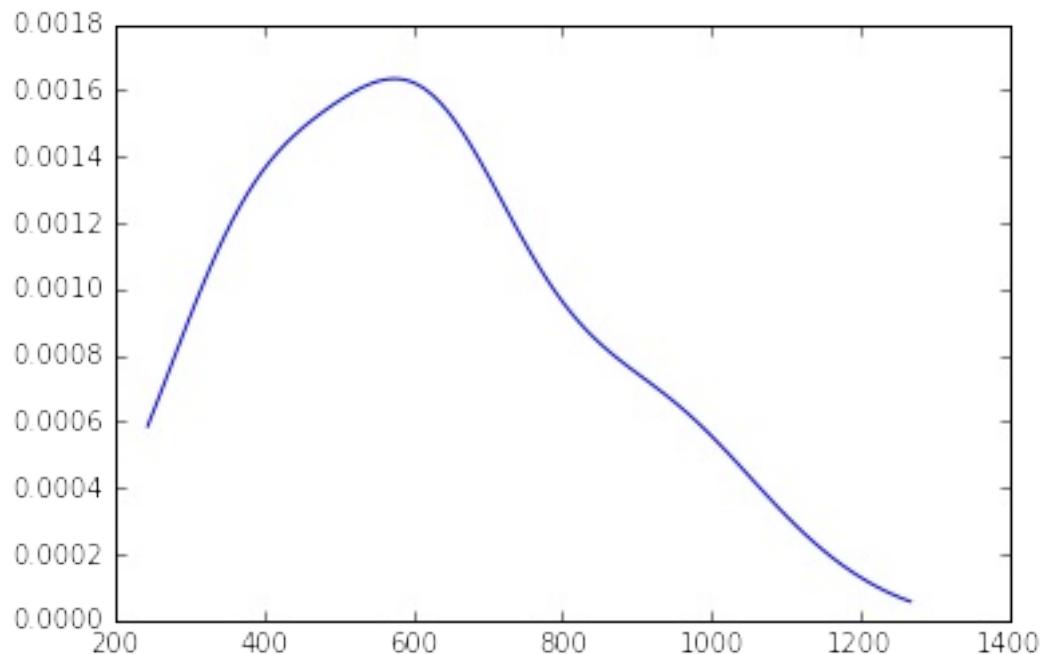
```
array([ 323.,  600.,  310.,  991.,  634.,  1024.,  1032.,  518.,
       347.,  507.,  948.,  607.,  581.,  532.,  393.,  414.,
       601.,  768.,  906.,  790.,  599.,  286.,  621.,  592.,
       596.,  868.,  686.,  918.,  410.,  1152.,  332.,  382.,
       771.,  455.,  668.,  772.,  874.,  271.,  426.,  378.,
       479.,  551.,  634.,  434.,  741.,  460.,  673.,  675.])
```

```
density = gaussian_kde(Y[:,0])
```

```
minY0 = Y[:,0].min()*0.90
maxY0 = Y[:,0].max()*1.10
x = np.linspace(minY0, maxY0, 100)
```

```
plt.plot(x,density(x))
```

```
[<matplotlib.lines.Line2D at 0x7fc12c86a2e8>]
```

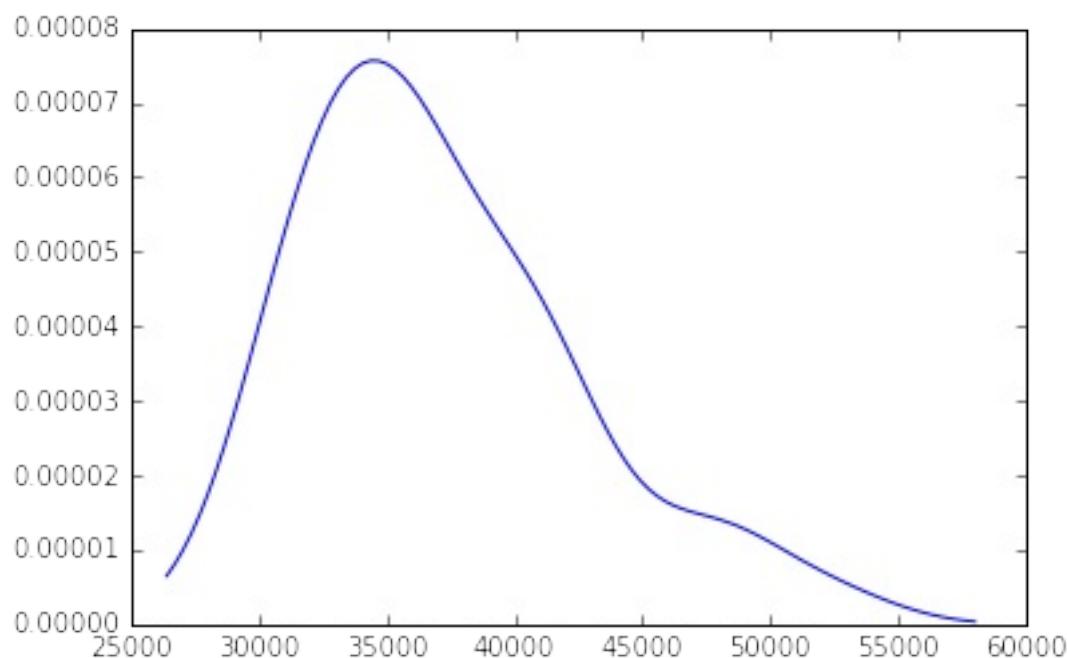


```
d2009 = gaussian_kde(Y[:, -1])
```

```
minY0 = Y[:, -1].min()*.90
maxY0 = Y[:, -1].max()*1.10
x = np.linspace(minY0, maxY0, 100)
```

```
plt.plot(x,d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x7fc12c9115f8>]
```



```
minR0 = RY.min()
```

```
maxR0 = RY.max()
```

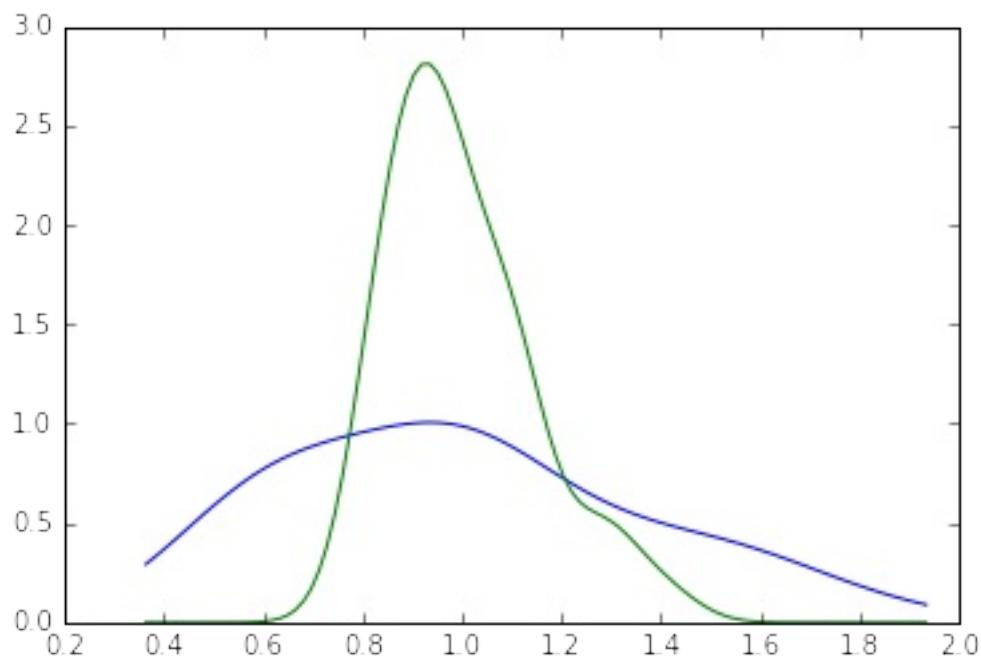
```
x = np.linspace(minR0, maxR0, 100)
```

```
d1929 = gaussian_kde(RY[:, 0])
```

```
d2009 = gaussian_kde(RY[:, -1])
```

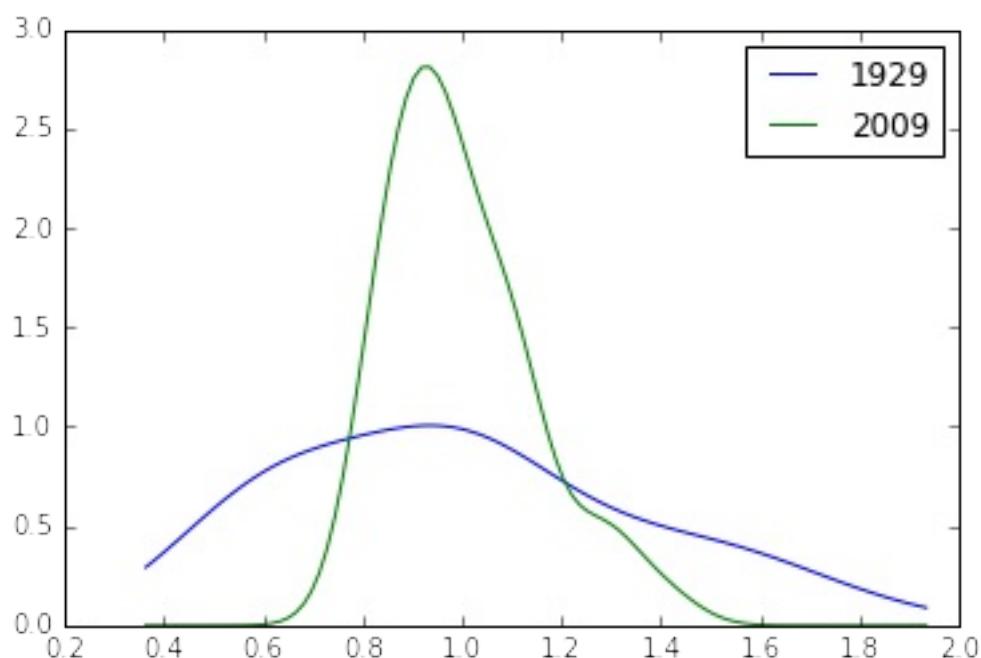
```
plt.plot(x, d1929(x))
plt.plot(x, d2009(x))
```

```
[<matplotlib.lines.Line2D at 0x7fc12c8aabe0>]
```



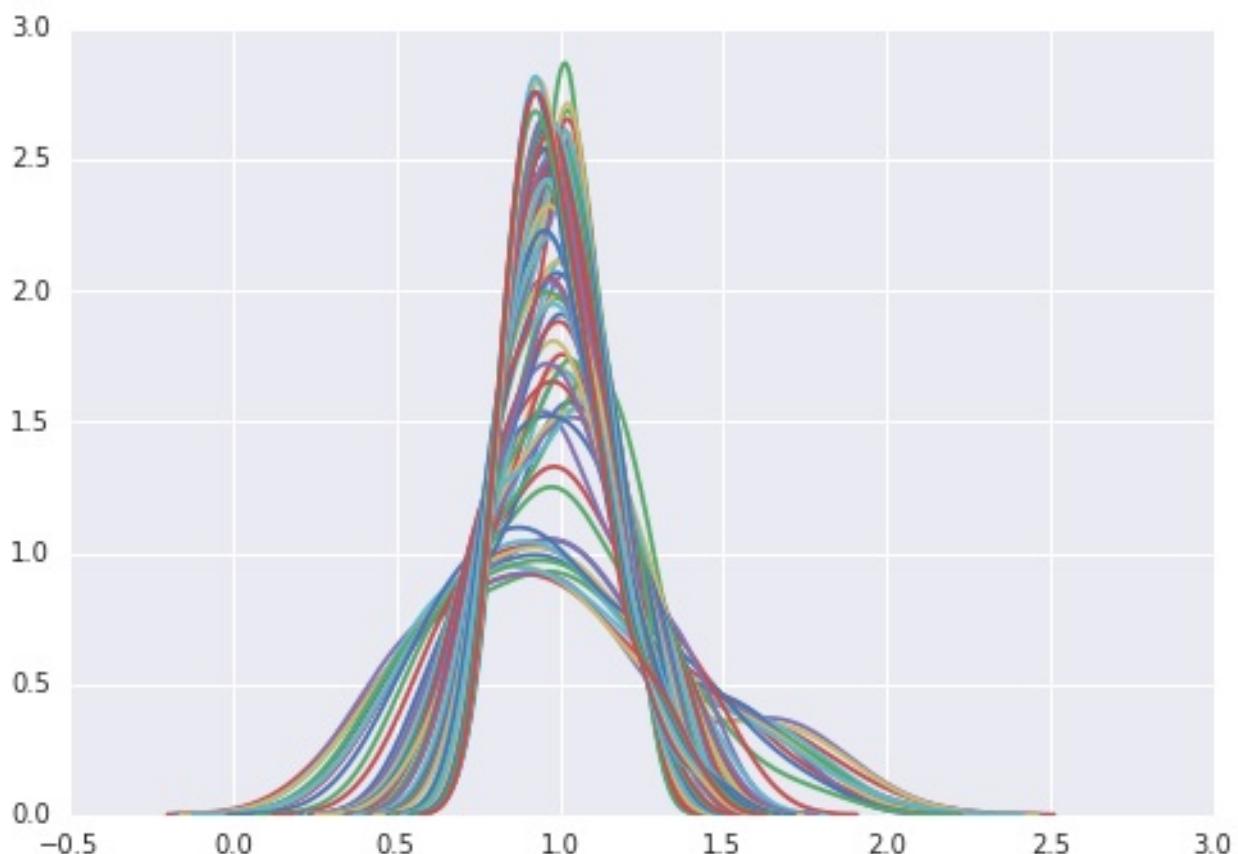
```
plt.plot(x, d1929(x), label='1929')
plt.plot(x, d2009(x), label='2009')
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc12c72b240>
```



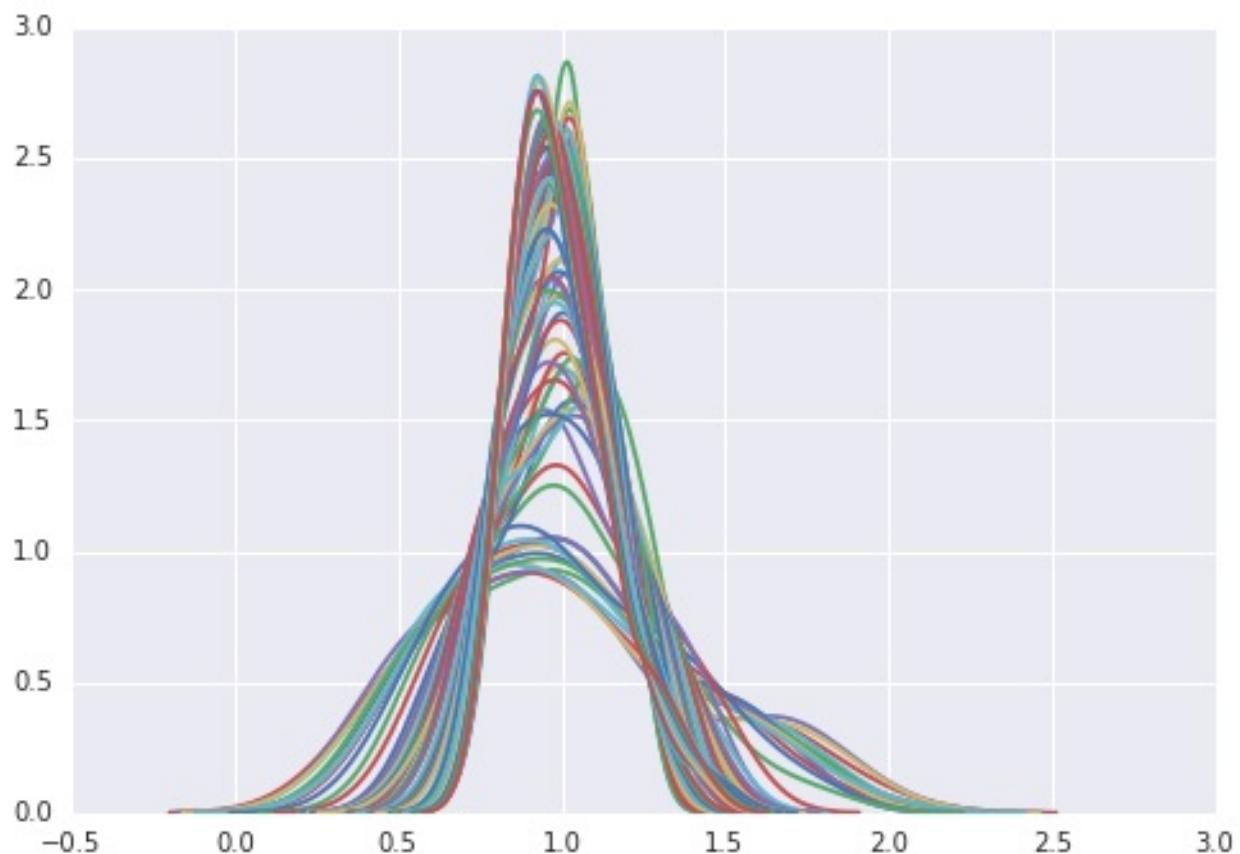
```
import seaborn as sns
for y in range(2010-1929):
    sns.kdeplot(RY[:,y])
#sns.kdeplot(data.HR80)
#sns.kdeplot(data.HR70)
#sns.kdeplot(data.HR60)
```

/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i nstead of an integer will result in an error in the future
 $y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j$

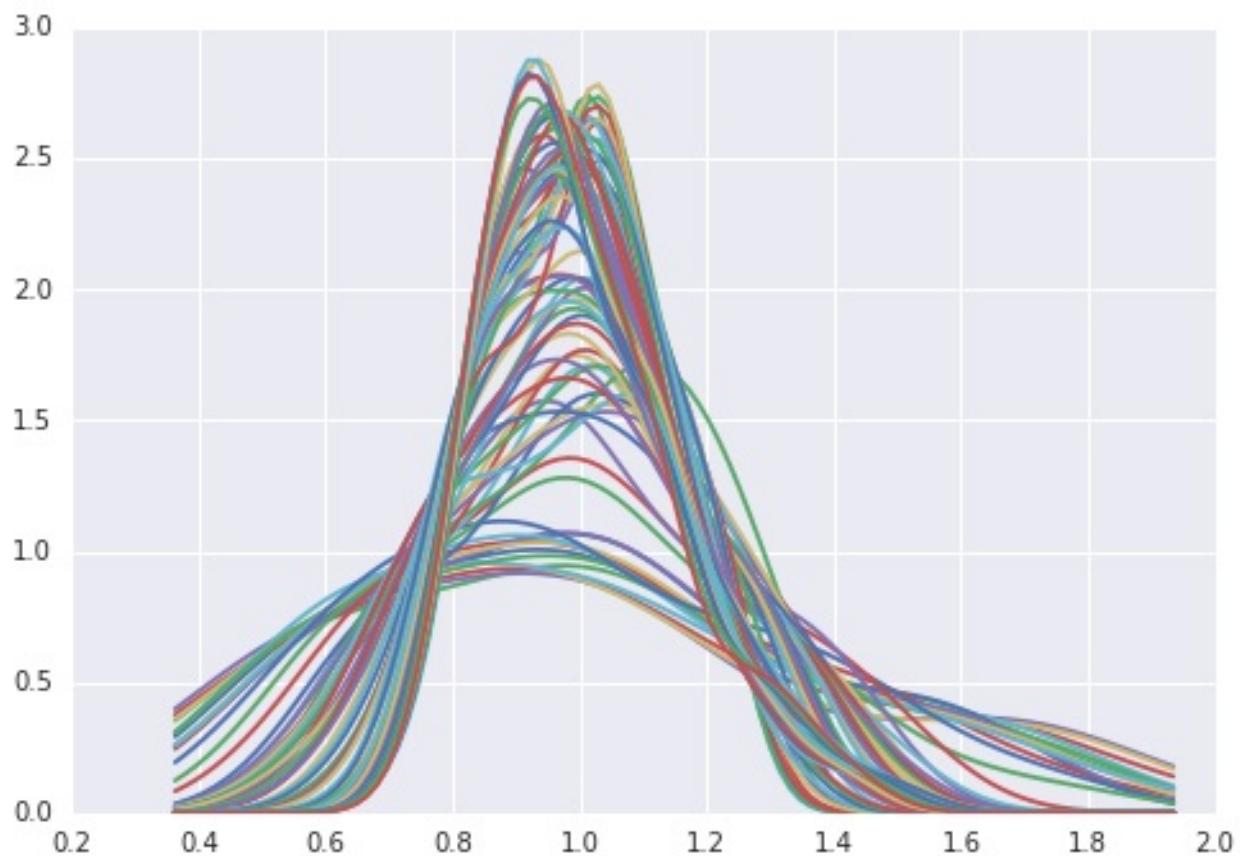


```
import seaborn as sns
for y in range(2010-1929):
    sns.kdeplot(RY[:,y])
```

/home/serge/anaconda2/envs/gds-scipy16/lib/python3.5/site-packages/statsmodels/non parametric/kdetools.py:20: VisibleDeprecationWarning: using a non-integer number i nstead of an integer will result in an error in the future
 $y = X[:m/2+1] + np.r_[0,X[m/2+1:],0]*1j$



```
for cs in RY.T: # take cross sections
    plt.plot(x, gaussian_kde(cs)(x))
```

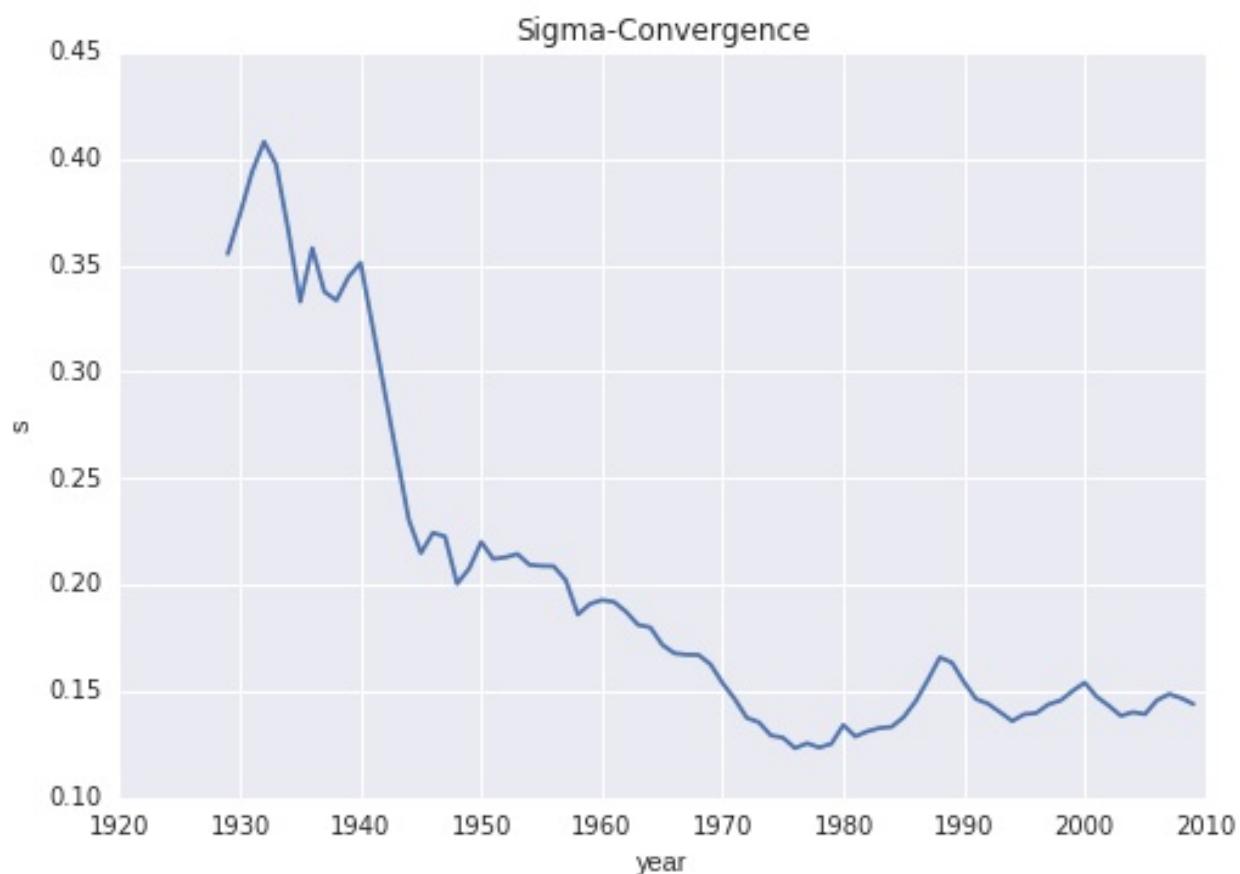


```
cs[0]
```

```
0.86746356478544273
```

```
sigma = RY.std(axis=0)
plt.plot(years, sigma)
plt.ylabel('s')
plt.xlabel('year')
plt.title("Sigma-Convergence")
```

```
<matplotlib.text.Text at 0x7fc129c01358>
```



So the distribution is becoming less dispersed over time.

But what about internal mixing? Do poor (rich) states remain poor (rich), or is there movement within the distribution over time?

Markov Chains

```
c = np.array([
    ['b', 'a', 'c'],
    ['c', 'c', 'a'],
    ['c', 'b', 'c'],
    ['a', 'a', 'b'],
    ['a', 'b', 'c']])
```

```
c
```

```
array([['b', 'a', 'c'],
      ['c', 'c', 'a'],
      ['c', 'b', 'c'],
      ['a', 'a', 'b'],
      ['a', 'b', 'c']],
      dtype='<U1')
```

```
m = ps.Markov(c)
```

```
m.classes
```

```
array(['a', 'b', 'c'],
      dtype='<U1')
```

```
m.transitions
```

```
array([[ 1.,  2.,  1.],
      [ 1.,  0.,  2.],
      [ 1.,  1.,  1.]])
```

```
m.p
```

```
matrix([[ 0.25        ,  0.5        ,  0.25        ],
       [ 0.33333333,  0.          ,  0.66666667],
       [ 0.33333333,  0.33333333,  0.33333333]])
```

State Per Capita Incomes

```
ps.examples.explain('us_income')
```

```
{'description': 'Per-capita income for the lower 47 US states 1929-2010',
 'explanation': [' * us48.shp: shapefile ',
                  ' * us48.dbf: dbf for shapefile',
                  ' * us48.shx: index for shapefile',
                  ' * usjoin.csv: attribute data (comma delimited file)'],
 'name': 'us_income'}
```

```
data = ps.pdio.read_files(ps.examples.get_path("us48.dbf"))
W = ps.queen_from_shapefile(ps.examples.get_path("us48.shp"))
W.transform = 'r'
```

```
data.STATE_NAME
```

```
0      Washington
1      Montana
2      Maine
3      North Dakota
4      South Dakota
5      Wyoming
6      Wisconsin
7      Idaho
8      Vermont
9      Minnesota
10     Oregon
11     New Hampshire
12     Iowa
13     Massachusetts
14     Nebraska
15     New York
16     Pennsylvania
17     Connecticut
18     Rhode Island
19     New Jersey
20     Indiana
21     Nevada
22     Utah
23     California
24     Ohio
25     Illinois
26     Delaware
27     West Virginia
28     Maryland
29     Colorado
30     Kentucky
31     Kansas
32     Virginia
33     Missouri
34     Arizona
35     Oklahoma
36     North Carolina
37     Tennessee
38     Texas
39     New Mexico
40     Alabama
41     Mississippi
42     Georgia
43     South Carolina
44     Arkansas
45     Louisiana
46     Florida
47     Michigan
Name: STATE_NAME, dtype: object
```

```
f = ps.open(ps.examples.get_path("usjoin.csv"))
pci = np.array([f.by_col[str(y)] for y in range(1929, 2010)])
pci.shape
```

```
(81, 48)
```

```
pci = pci.T
```

```
pci.shape
```

```
(48, 81)
```

```
cnames = f.by_col('Name')
```

```
cnames
```

```
[ 'Alabama',
  'Arizona',
  'Arkansas',
  'California',
  'Colorado',
  'Connecticut',
  'Delaware',
  'Florida',
  'Georgia',
  'Idaho',
  'Illinois',
  'Indiana',
  'Iowa',
  'Kansas',
  'Kentucky',
  'Louisiana',
  'Maine',
  'Maryland',
  'Massachusetts',
  'Michigan',
  'Minnesota',
  'Mississippi',
  'Missouri',
  'Montana',
  'Nebraska',
  'Nevada',
  'New Hampshire',
  'New Jersey',
  'New Mexico',
  'New York',
  'North Carolina',
  'North Dakota',
  'Ohio',
  'Oklahoma',
  'Oregon',
  'Pennsylvania',
  'Rhode Island',
  'South Carolina',
  'South Dakota',
  'Tennessee',
  'Texas',
  'Utah',
  'Vermont',
  'Virginia',
  'Washington',
  'West Virginia',
  'Wisconsin',
  'Wyoming']
```

```
ids = [ cnames.index(name) for name in data.STATE_NAME]
```

```
ids
```

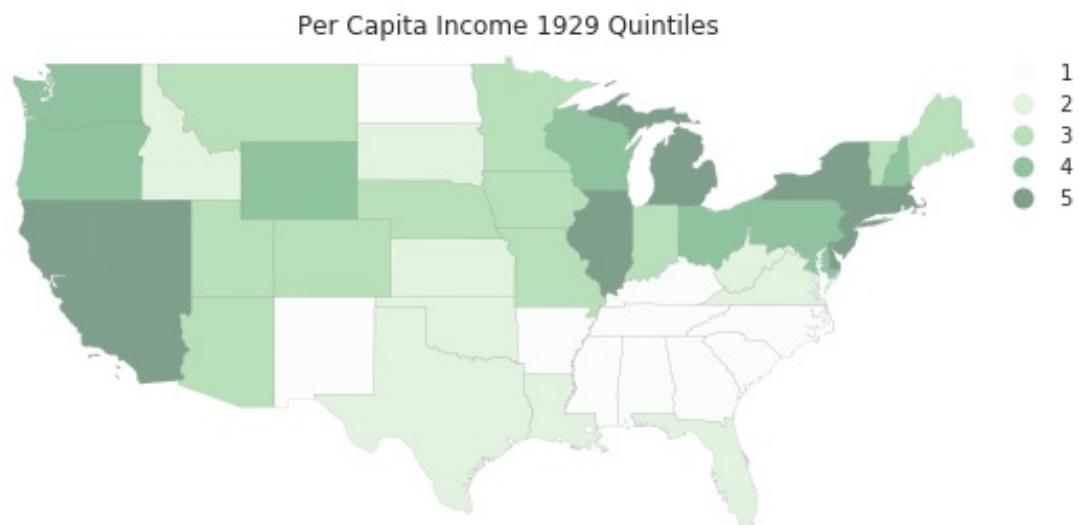
```
[44,  
 23,  
 16,  
 31,  
 38,  
 47,  
 46,  
 9,  
 42,  
 20,  
 34,  
 26,  
 12,  
 18,  
 24,  
 29,  
 35,  
 5,  
 36,  
 27,  
 11,  
 25,  
 41,  
 3,  
 32,  
 10,  
 6,  
 45,  
 17,  
 4,  
 14,  
 13,  
 43,  
 22,  
 1,  
 33,  
 30,  
 39,  
 40,  
 28,  
 0,  
 21,  
 8,  
 37,  
 2,  
 15,  
 7,  
 19]
```

```
pci = pci[ids]
RY = RY[ids]
```

```
import matplotlib.pyplot as plt

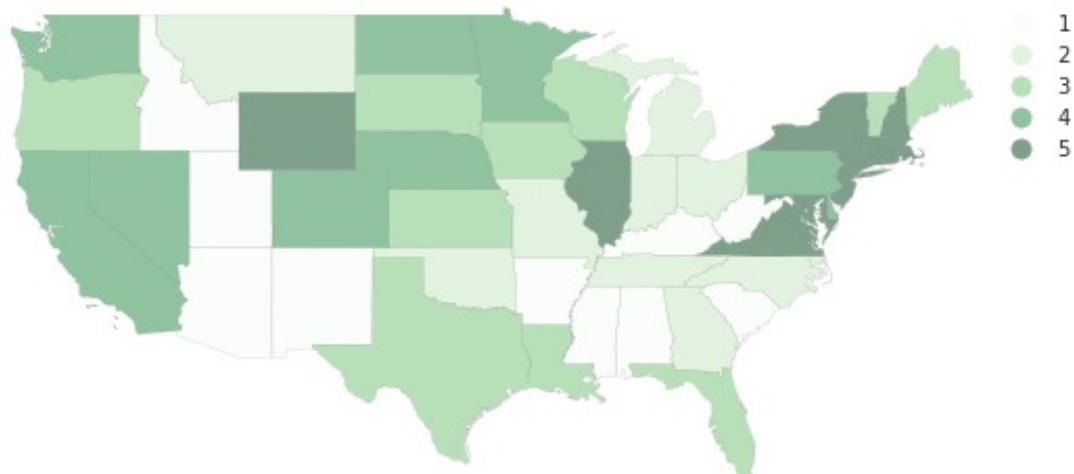
import geopandas as gpd
shp_link = ps.examples.get_path('us48.shp')
tx = gpd.read_file(shp_link)
pci29 = ps.Quantiles(pci[:,0], k=5)
f, ax = plt.subplots(1, figsize=(10, 5))
tx.assign(cl=pci29.yb+1).plot(column='cl', categorical=True, \
    k=5, cmap='Greens', linewidth=0.1, ax=ax, \
    edgecolor='grey', legend=True)
ax.set_axis_off()
plt.title('Per Capita Income 1929 Quintiles')

plt.show()
```



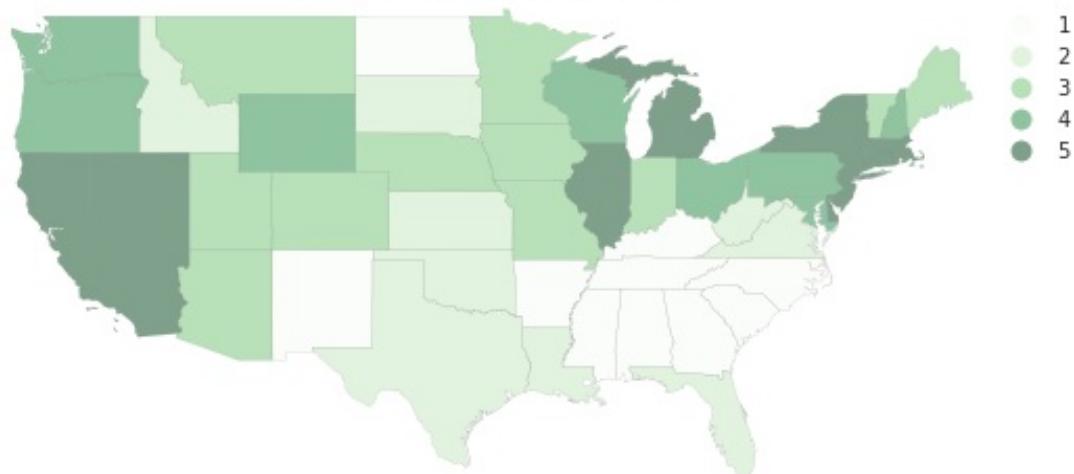
```
pci2009 = ps.Quantiles(pci[:, -1], k=5)
f, ax = plt.subplots(1, figsize=(10, 5))
tx.assign(cl=pci2009.yb+1).plot(column='cl', categorical=True, \
    k=5, cmap='Greens', linewidth=0.1, ax=ax, \
    edgecolor='grey', legend=True)
ax.set_axis_off()
plt.title('Per Capita Income 2009 Quintiles')
plt.show()
```

Per Capita Income 2009 Quintiles

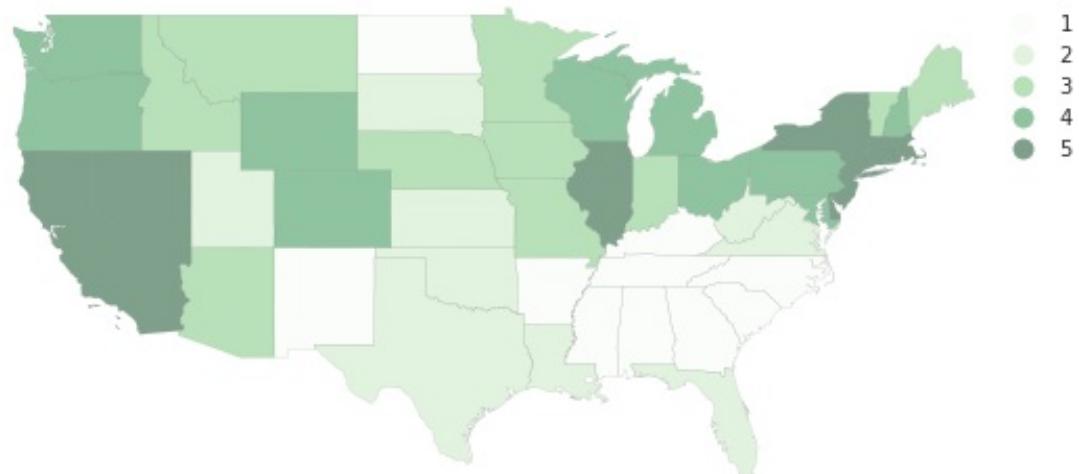


```
for y in range(2010-1929):
    pciy = ps.Quantiles(pci[:,y], k=5)
    f, ax = plt.subplots(1, figsize=(10, 5))
    tx.assign(cl=pciy.yb+1).plot(column='cl', categorical=True, \
        k=5, cmap='Greens', linewidth=0.1, ax=ax, \
        edgecolor='grey', legend=True)
    ax.set_axis_off()
    plt.title("Per Capita Income %d Quintiles"%(1929+y))
    plt.show()
```

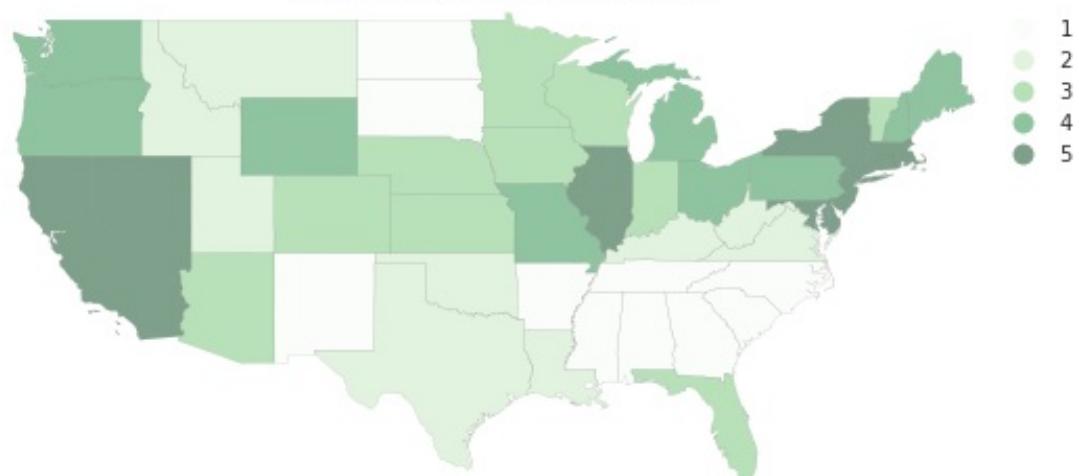
Per Capita Income 1929 Quintiles



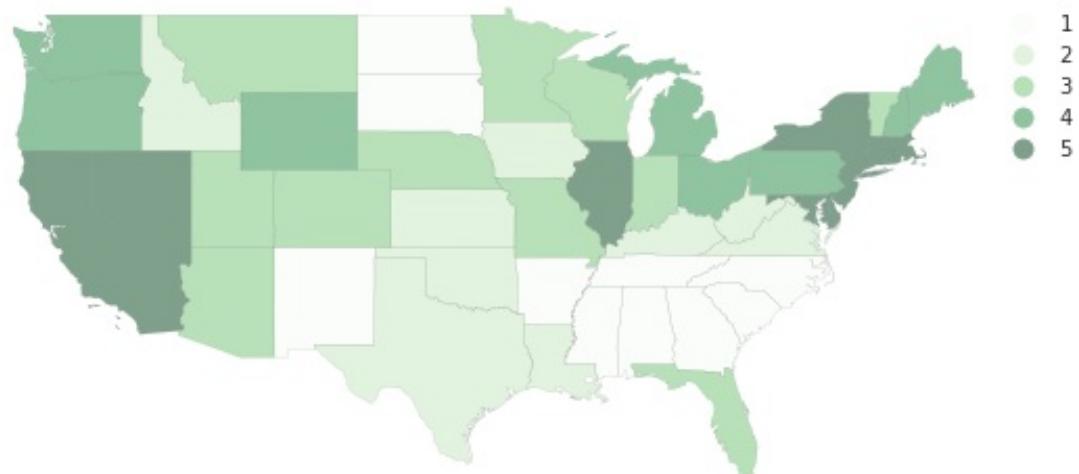
Per Capita Income 1930 Quintiles



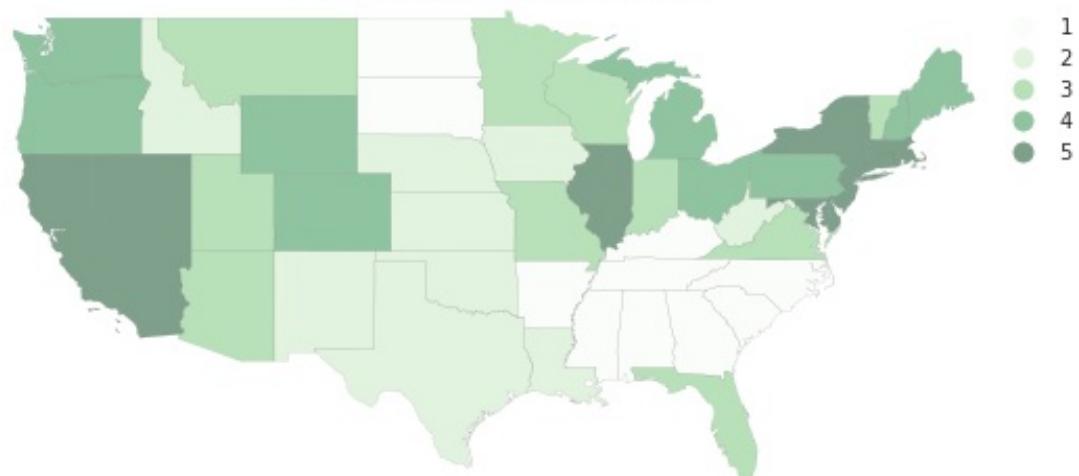
Per Capita Income 1931 Quintiles



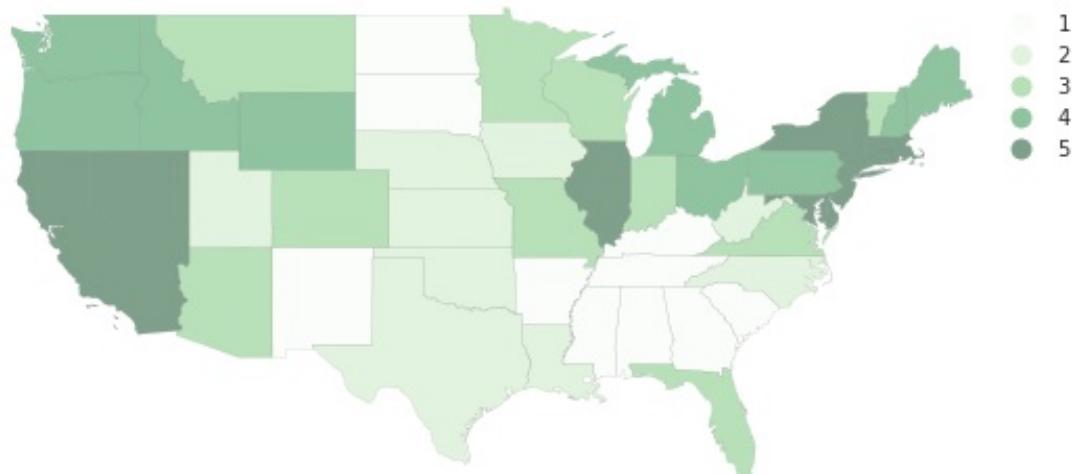
Per Capita Income 1932 Quintiles



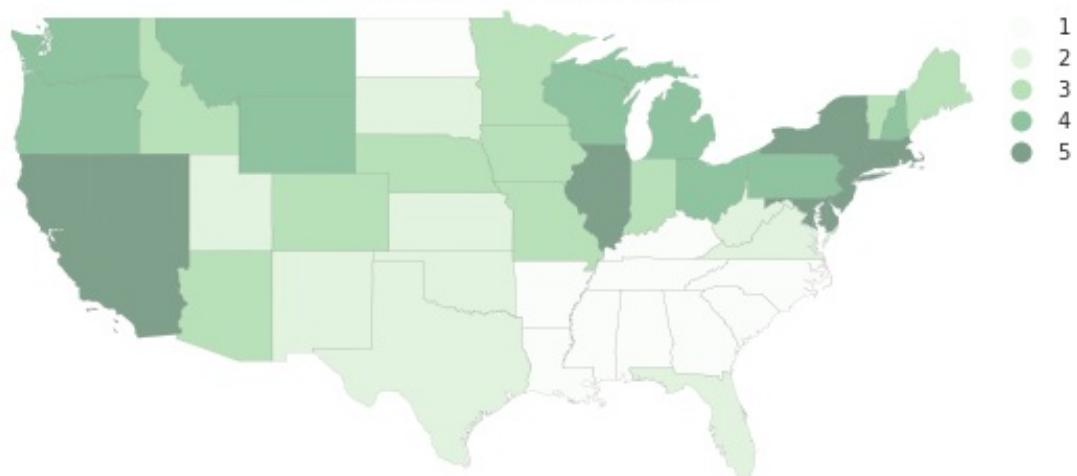
Per Capita Income 1933 Quintiles



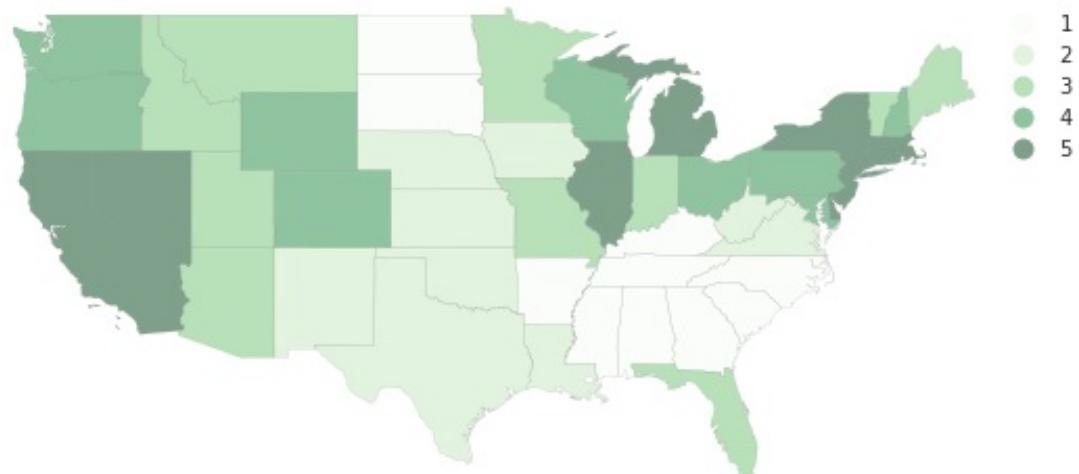
Per Capita Income 1934 Quintiles



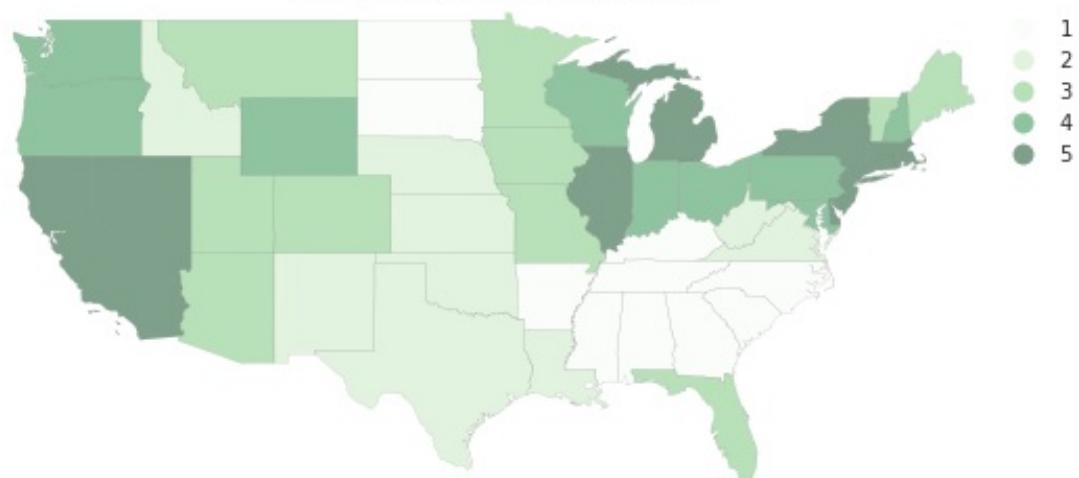
Per Capita Income 1935 Quintiles



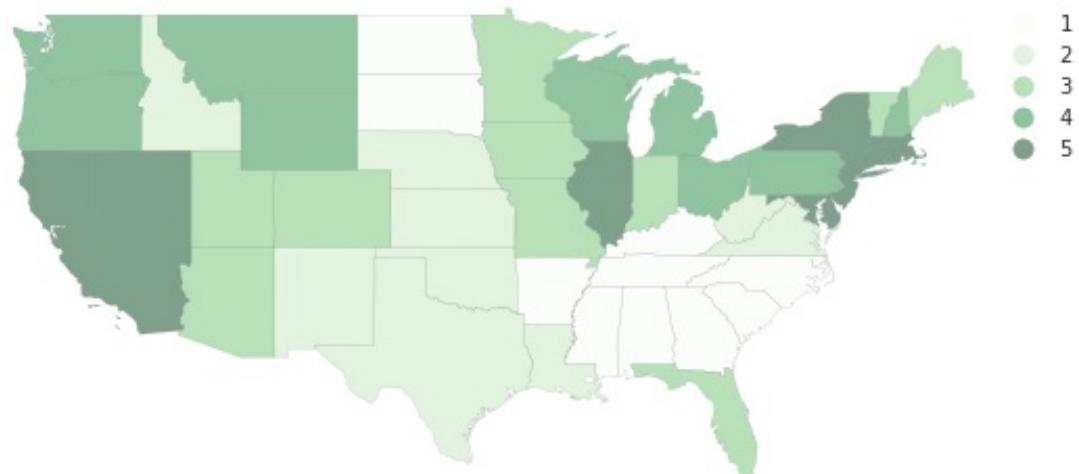
Per Capita Income 1936 Quintiles



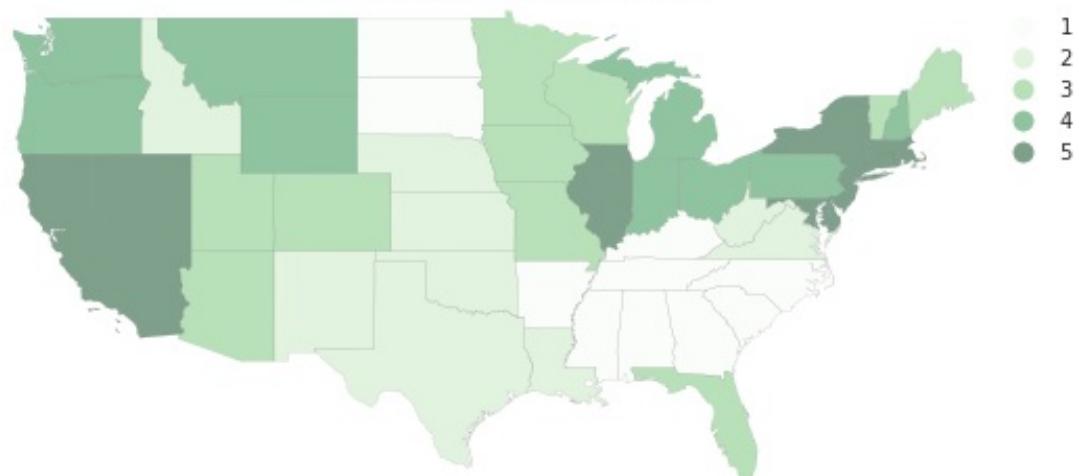
Per Capita Income 1937 Quintiles



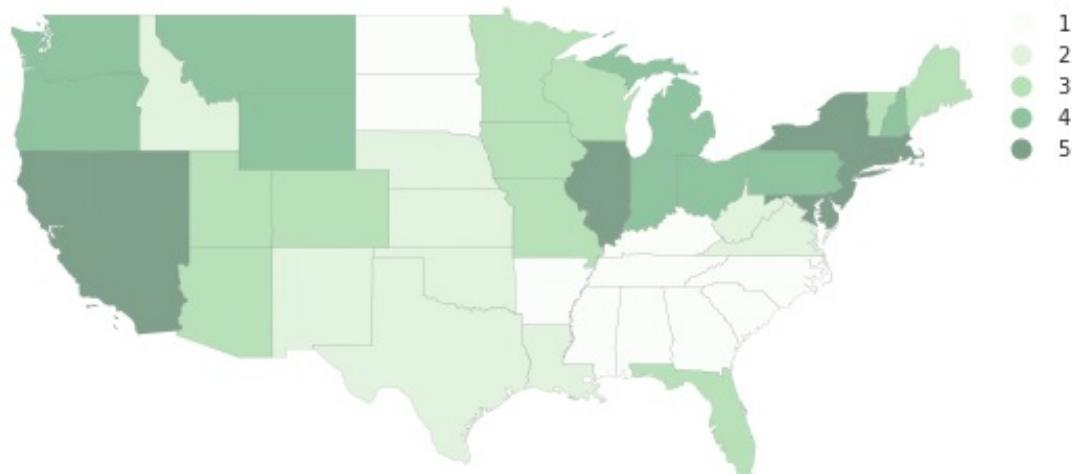
Per Capita Income 1938 Quintiles



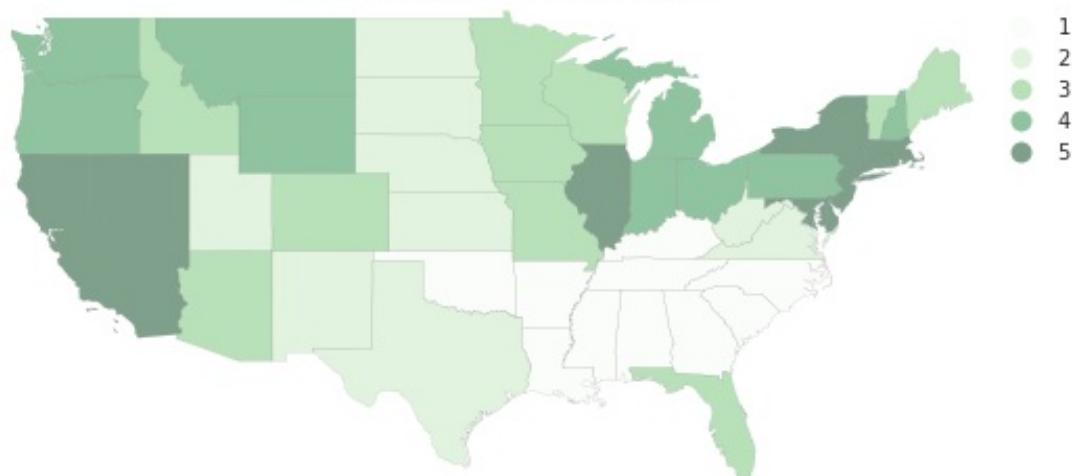
Per Capita Income 1939 Quintiles



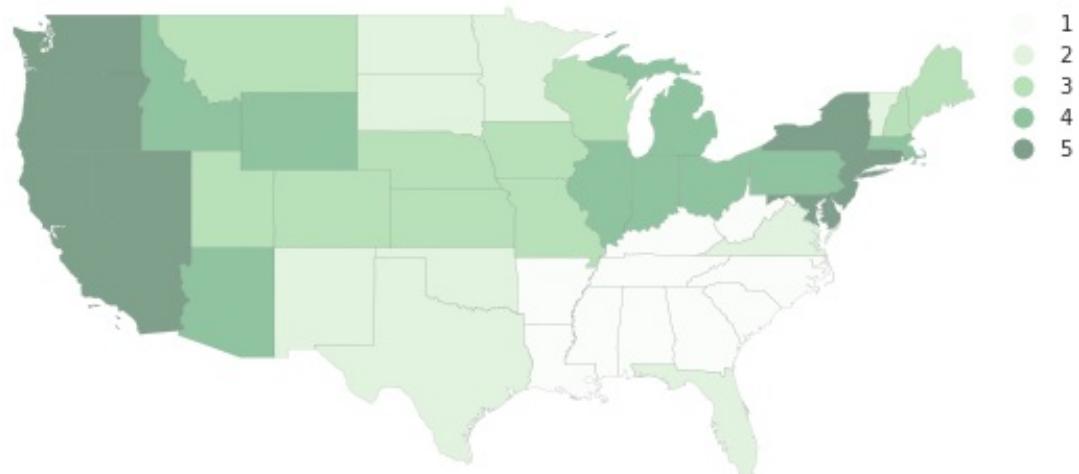
Per Capita Income 1940 Quintiles



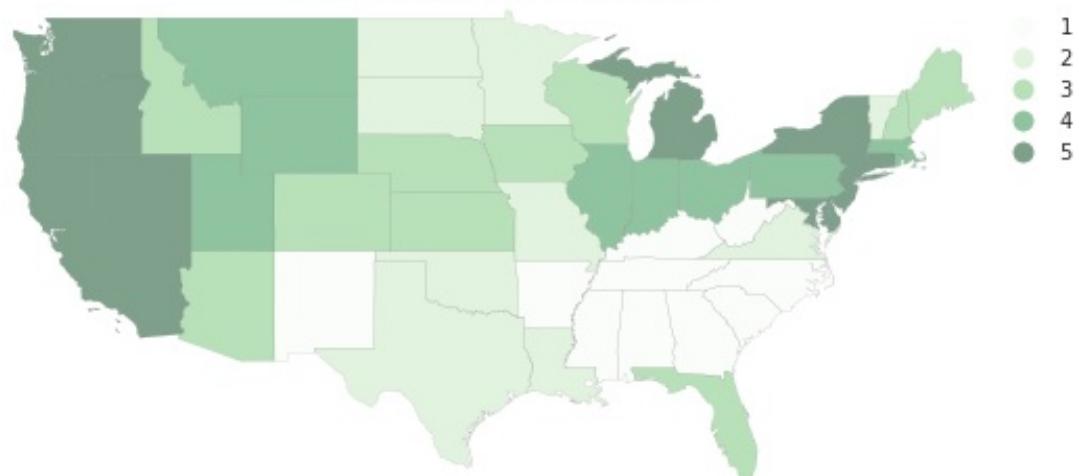
Per Capita Income 1941 Quintiles



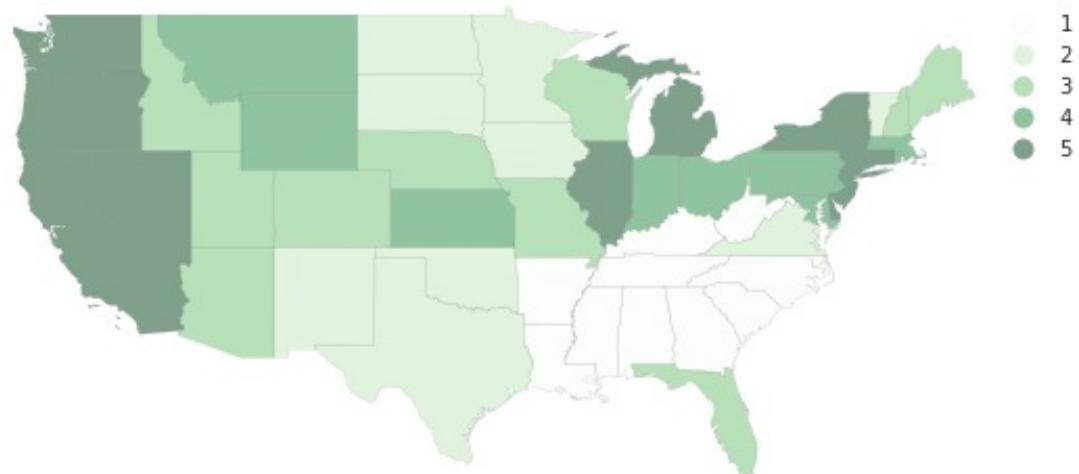
Per Capita Income 1942 Quintiles



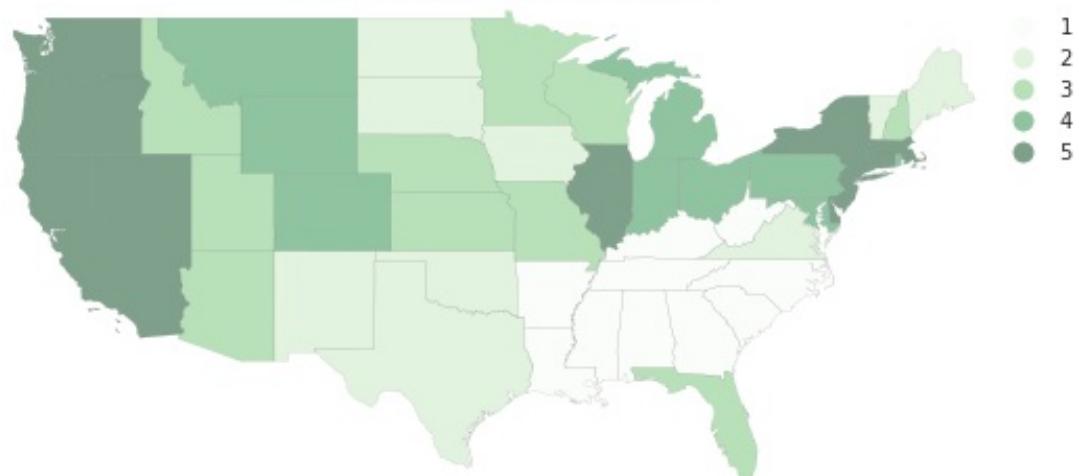
Per Capita Income 1943 Quintiles



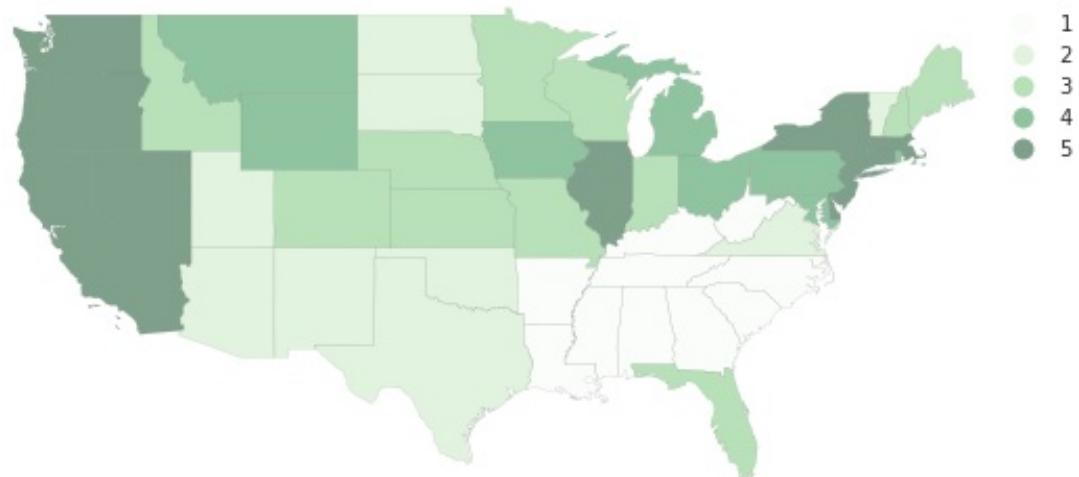
Per Capita Income 1944 Quintiles



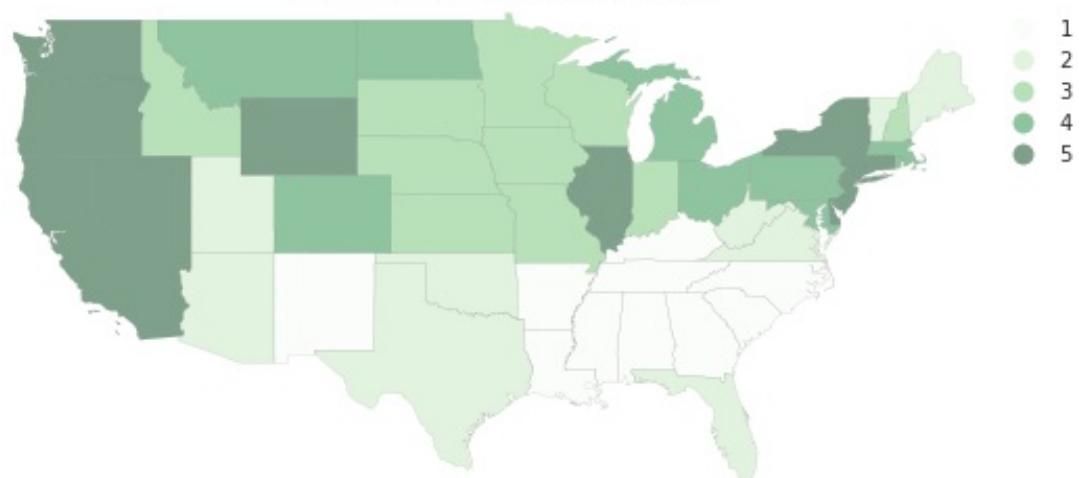
Per Capita Income 1945 Quintiles



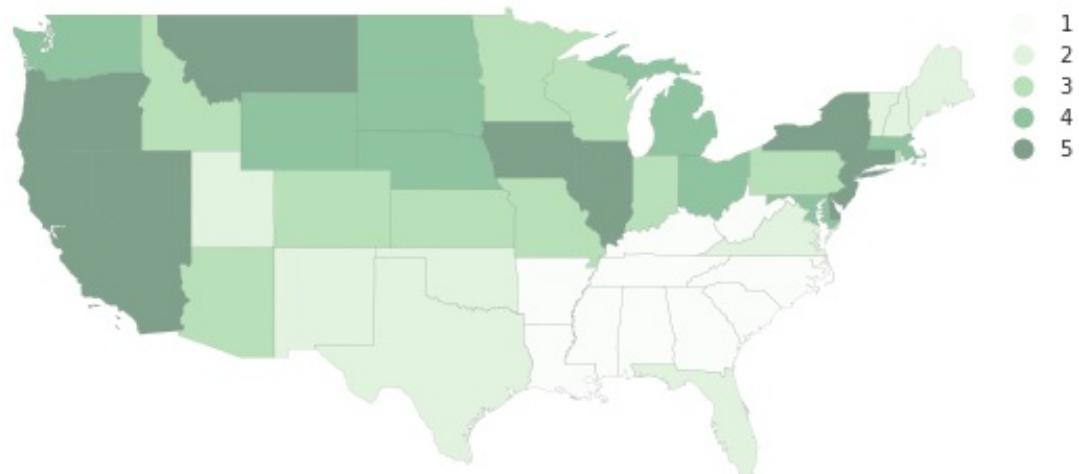
Per Capita Income 1946 Quintiles



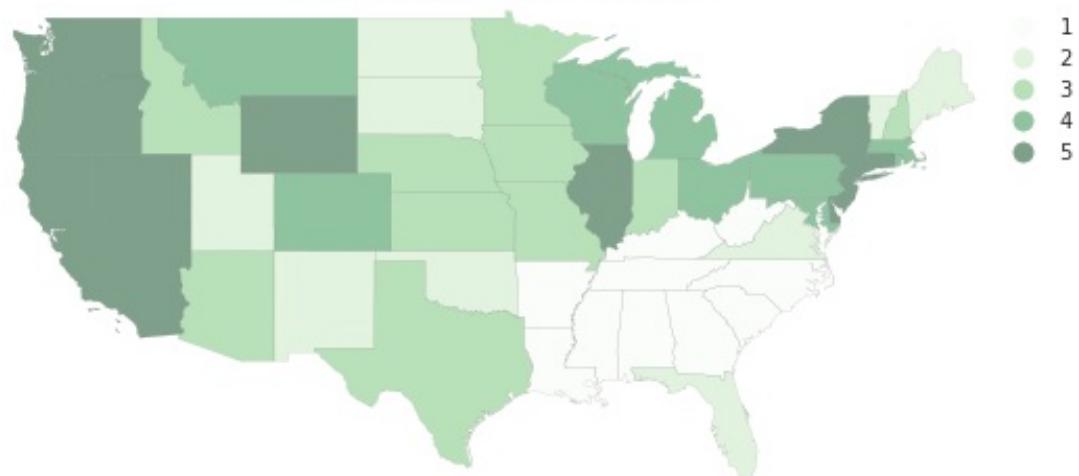
Per Capita Income 1947 Quintiles



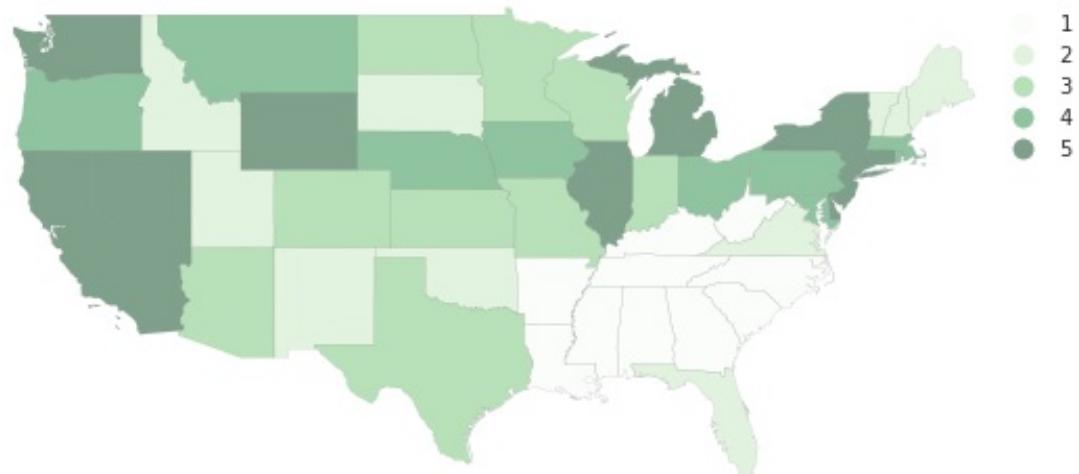
Per Capita Income 1948 Quintiles



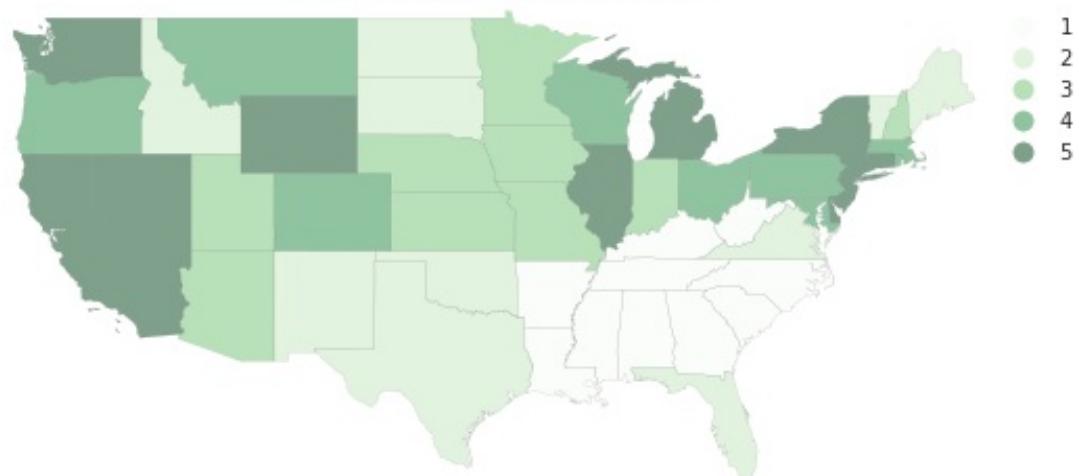
Per Capita Income 1949 Quintiles



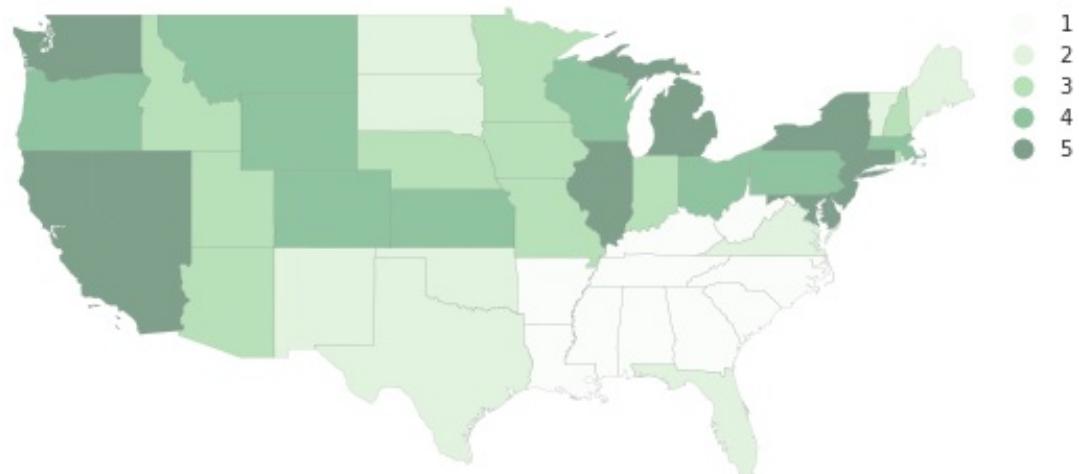
Per Capita Income 1950 Quintiles



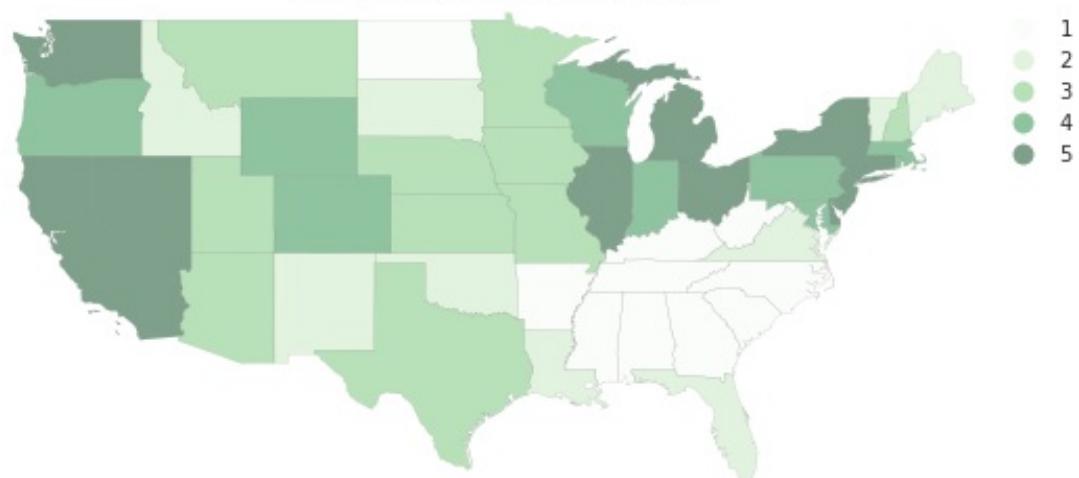
Per Capita Income 1951 Quintiles



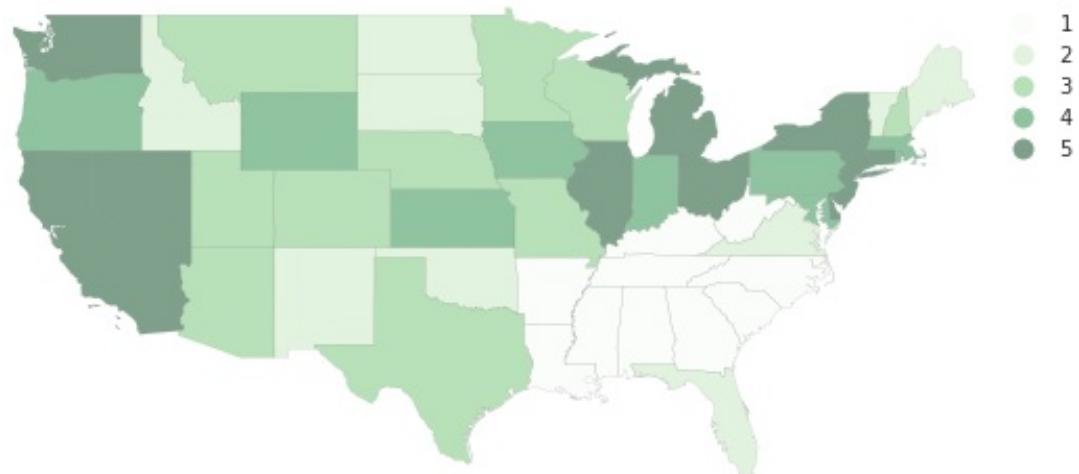
Per Capita Income 1952 Quintiles



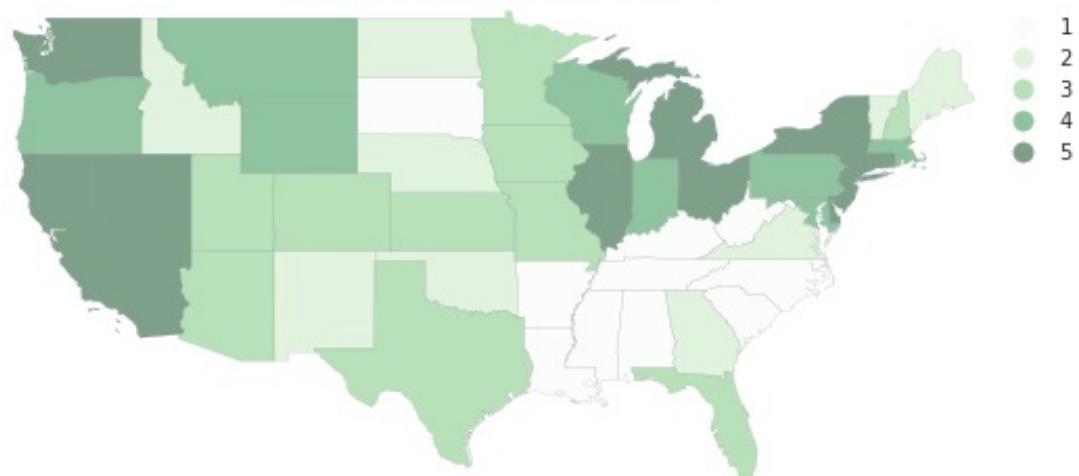
Per Capita Income 1953 Quintiles



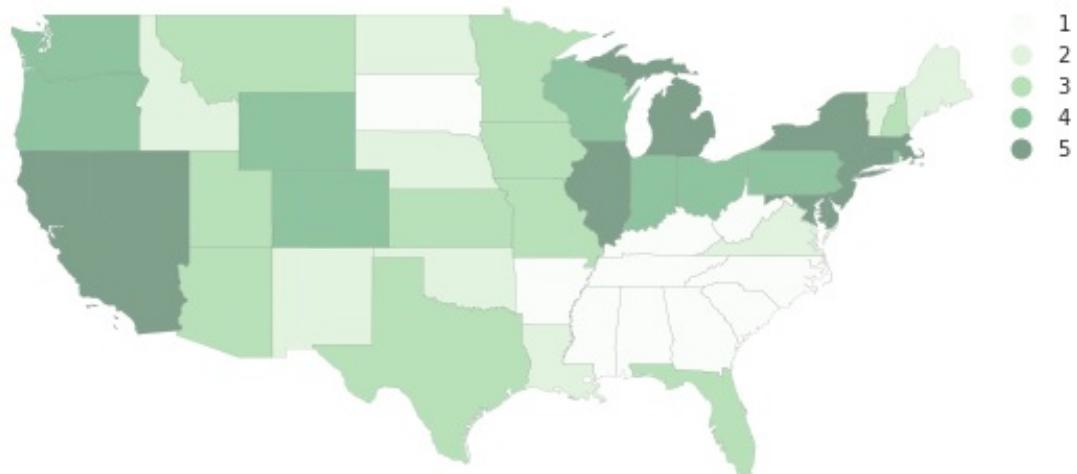
Per Capita Income 1954 Quintiles



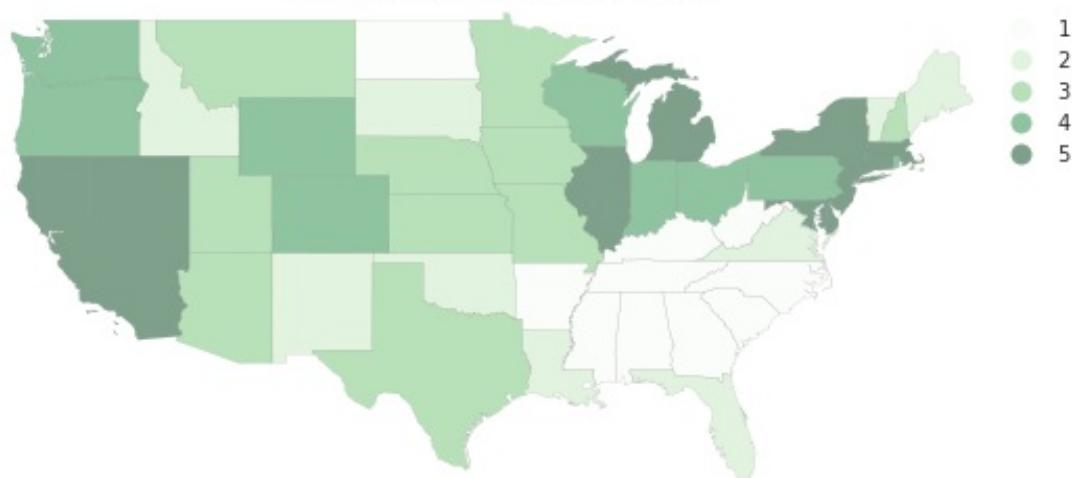
Per Capita Income 1955 Quintiles



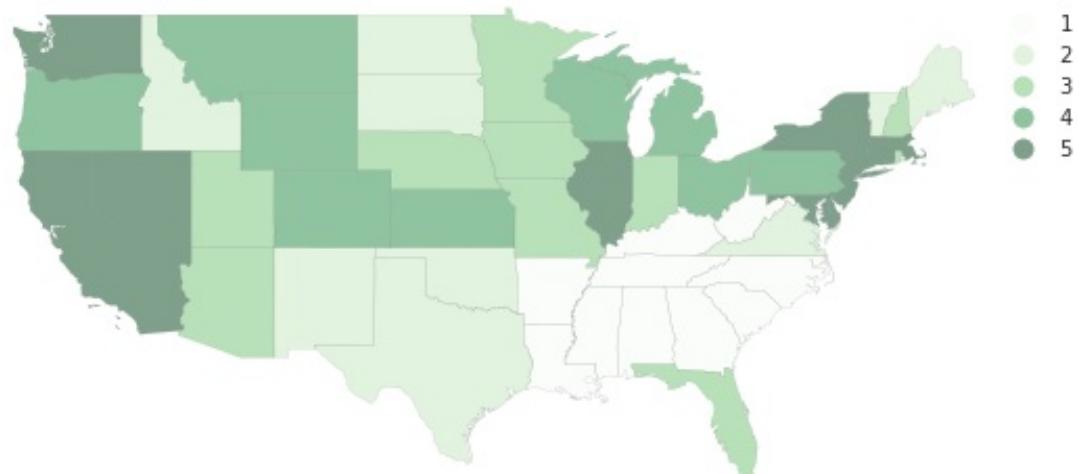
Per Capita Income 1956 Quintiles



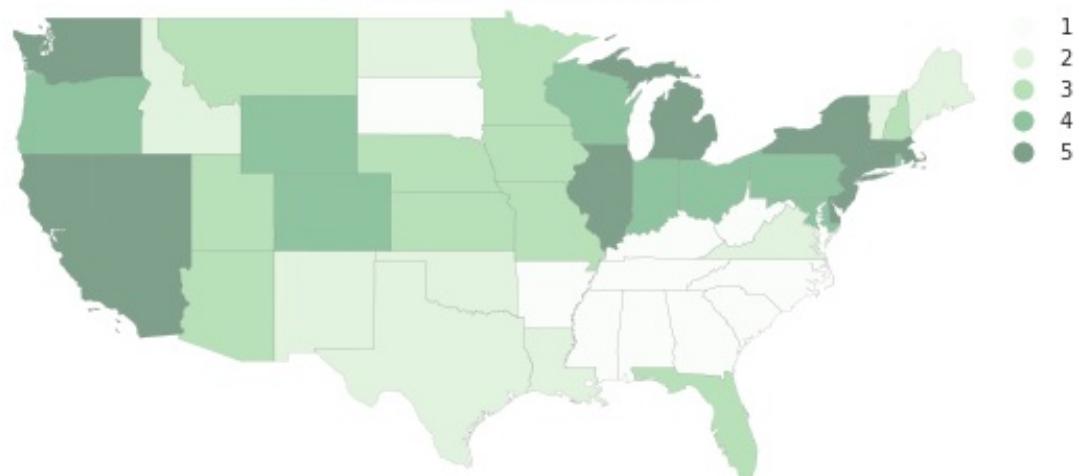
Per Capita Income 1957 Quintiles



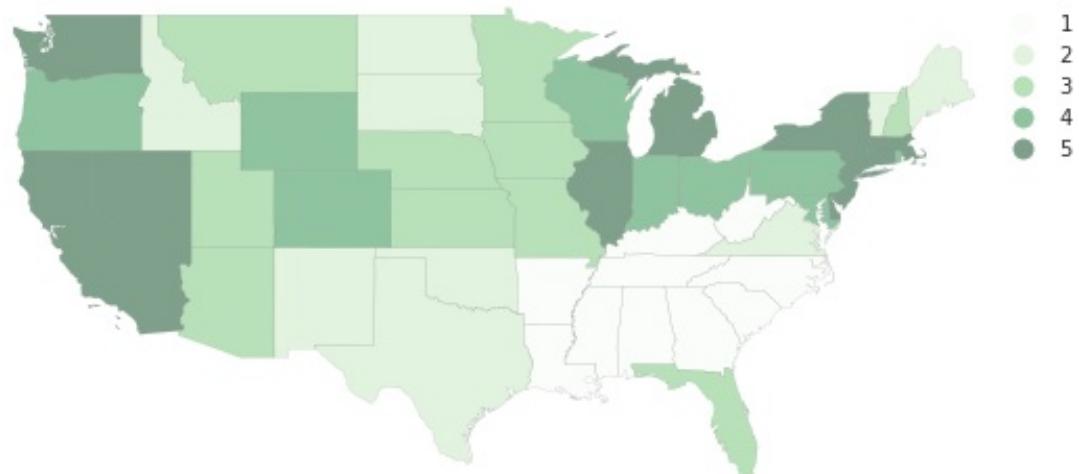
Per Capita Income 1958 Quintiles



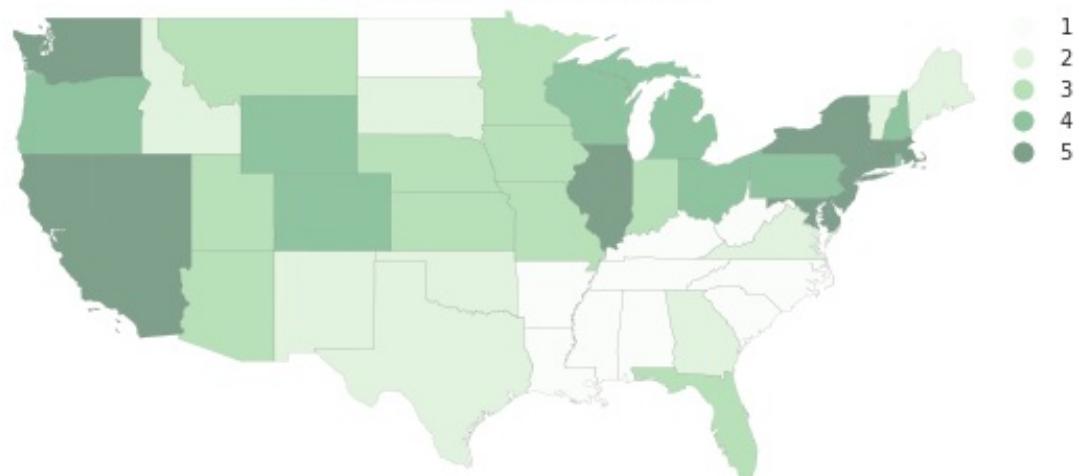
Per Capita Income 1959 Quintiles



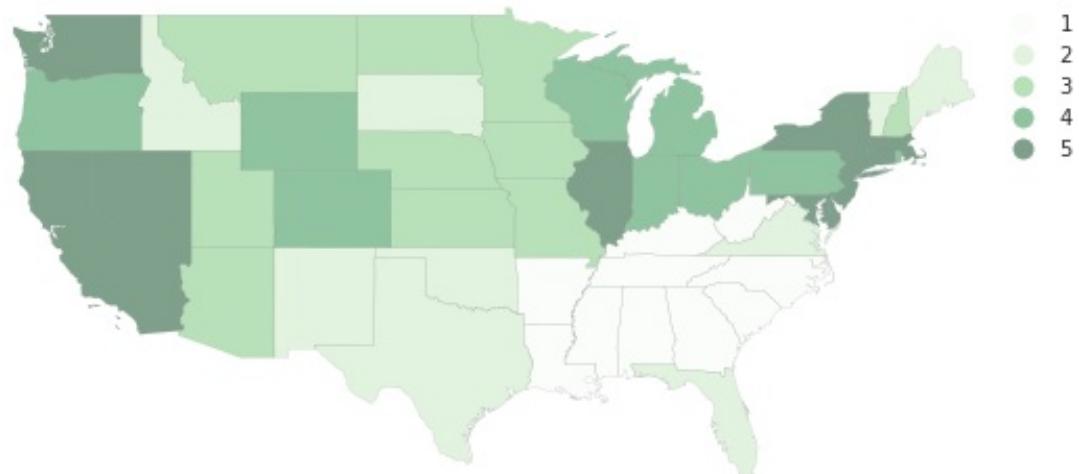
Per Capita Income 1960 Quintiles



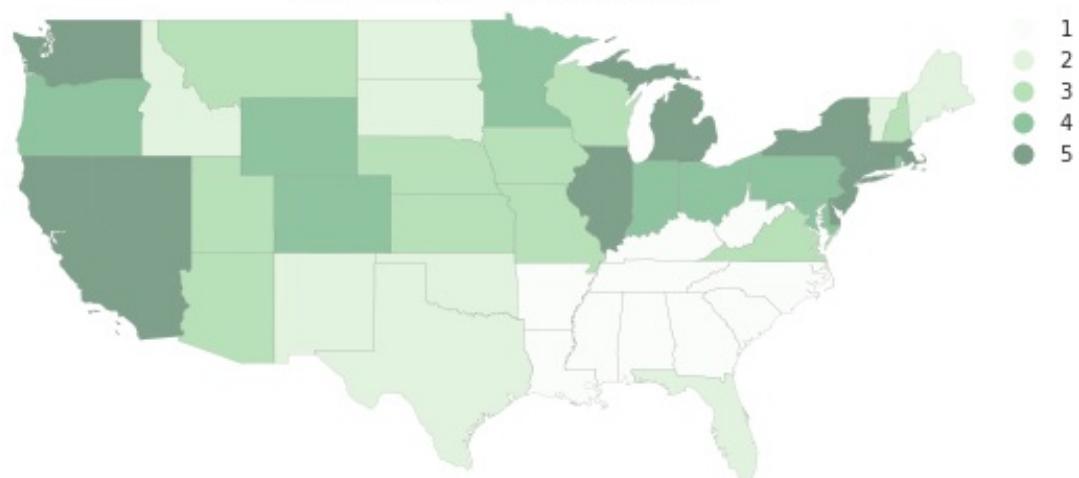
Per Capita Income 1961 Quintiles



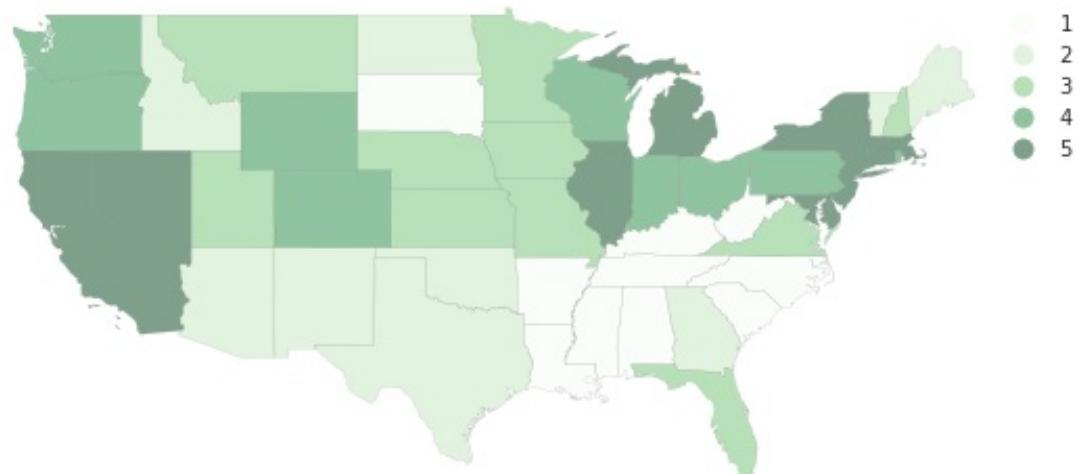
Per Capita Income 1962 Quintiles



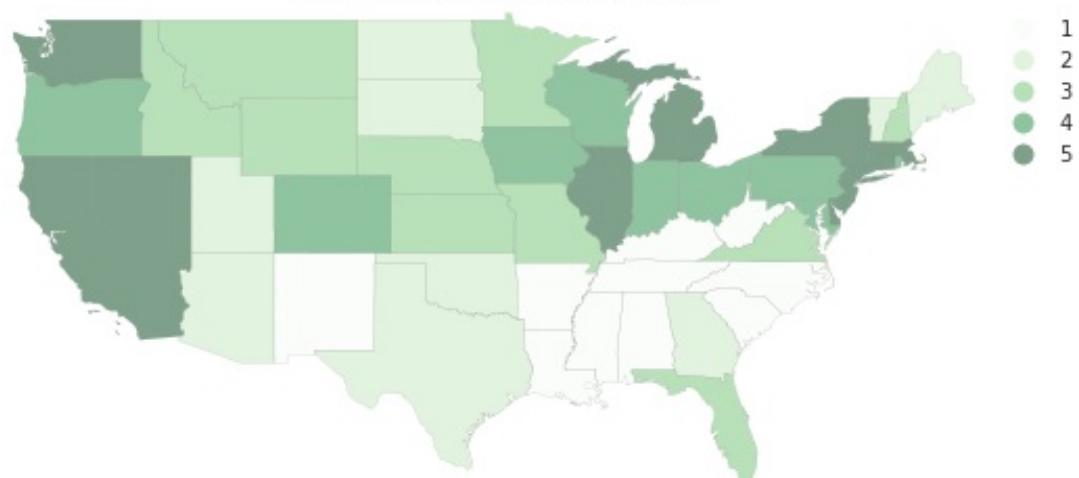
Per Capita Income 1963 Quintiles



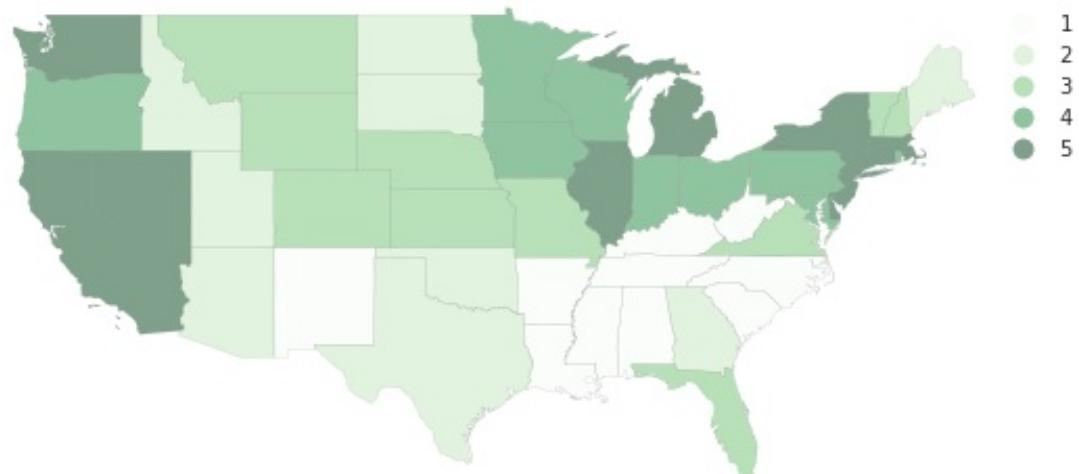
Per Capita Income 1964 Quintiles



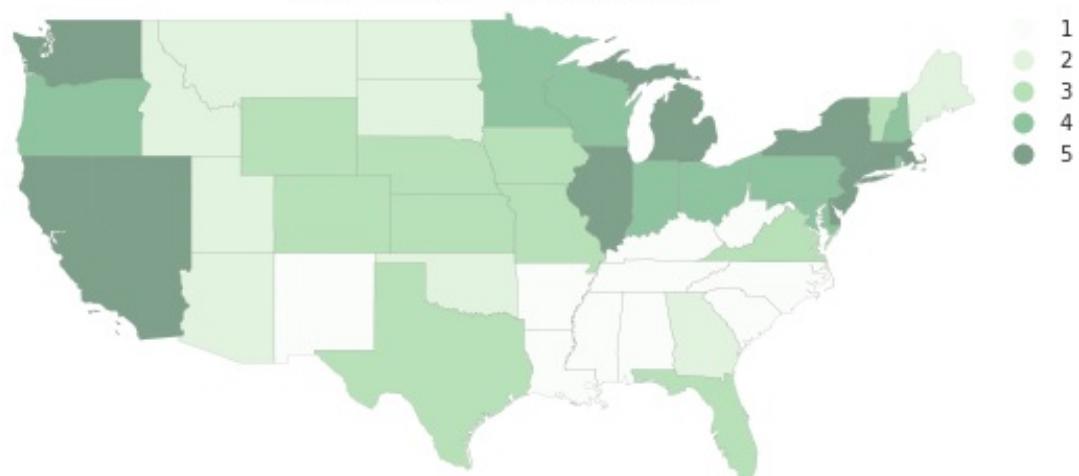
Per Capita Income 1965 Quintiles



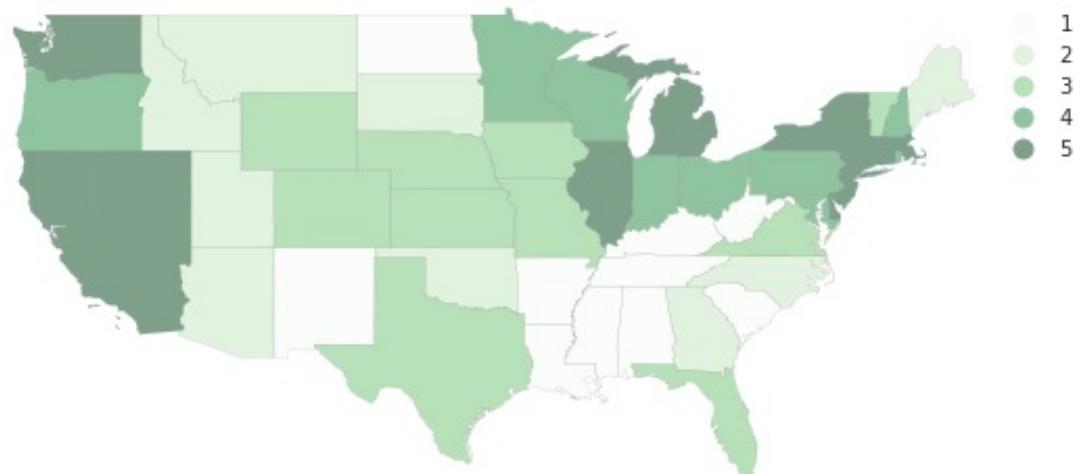
Per Capita Income 1966 Quintiles



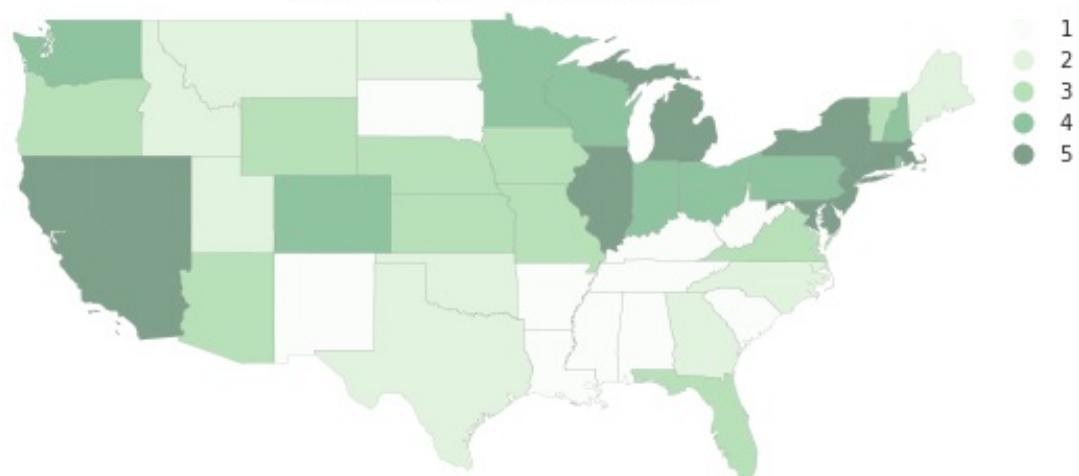
Per Capita Income 1967 Quintiles



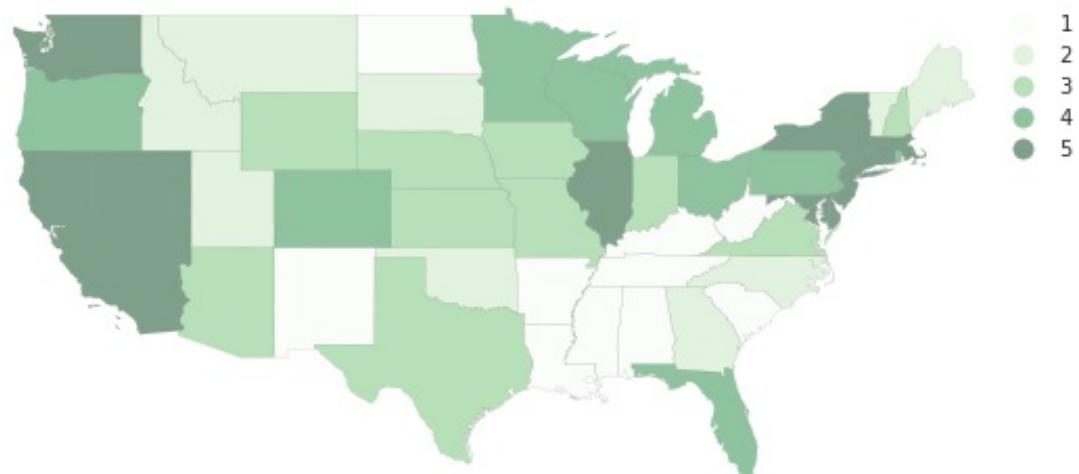
Per Capita Income 1968 Quintiles



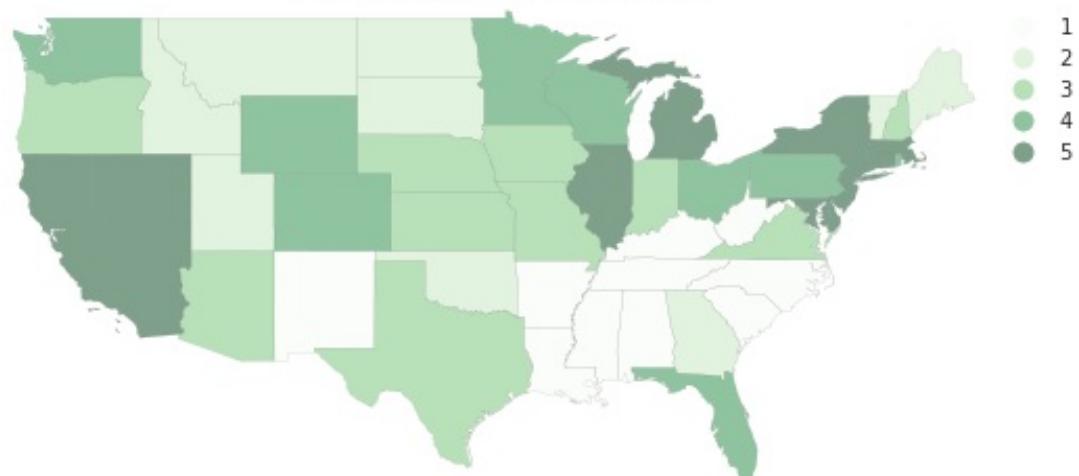
Per Capita Income 1969 Quintiles



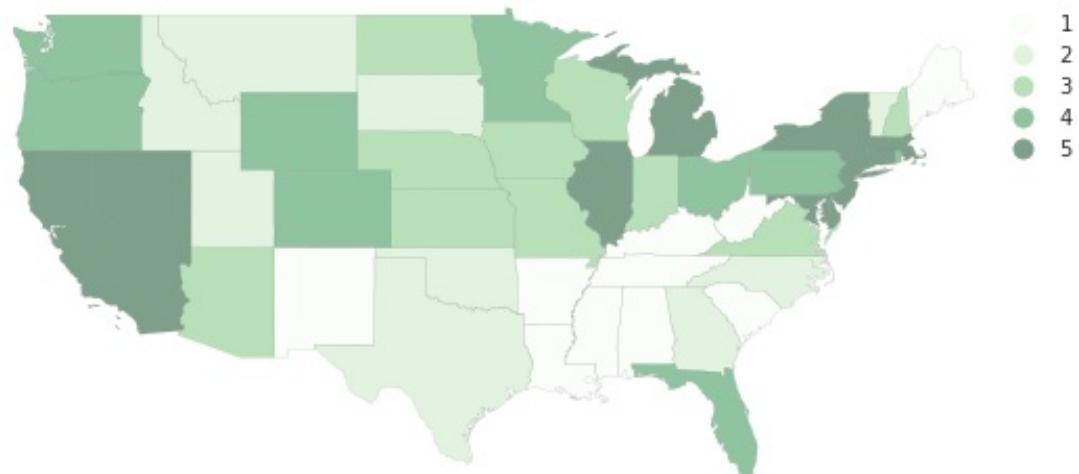
Per Capita Income 1970 Quintiles



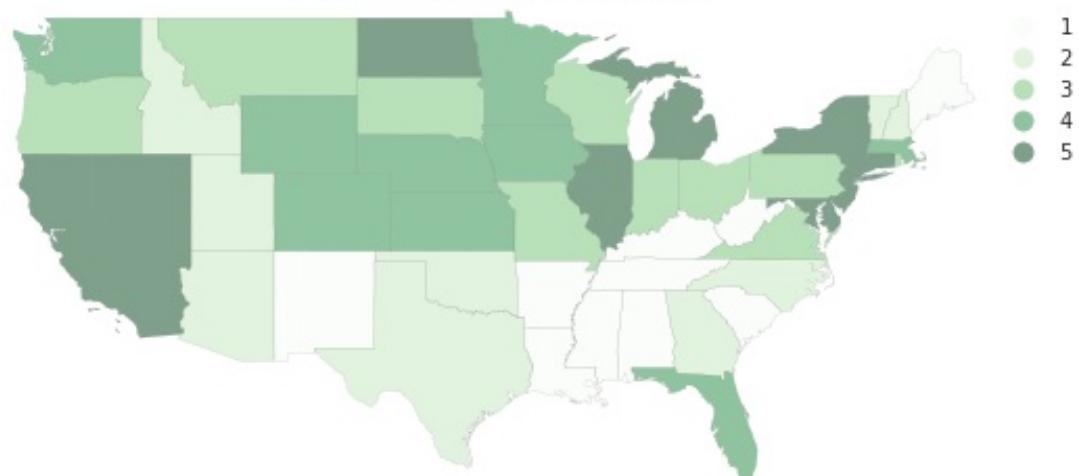
Per Capita Income 1971 Quintiles



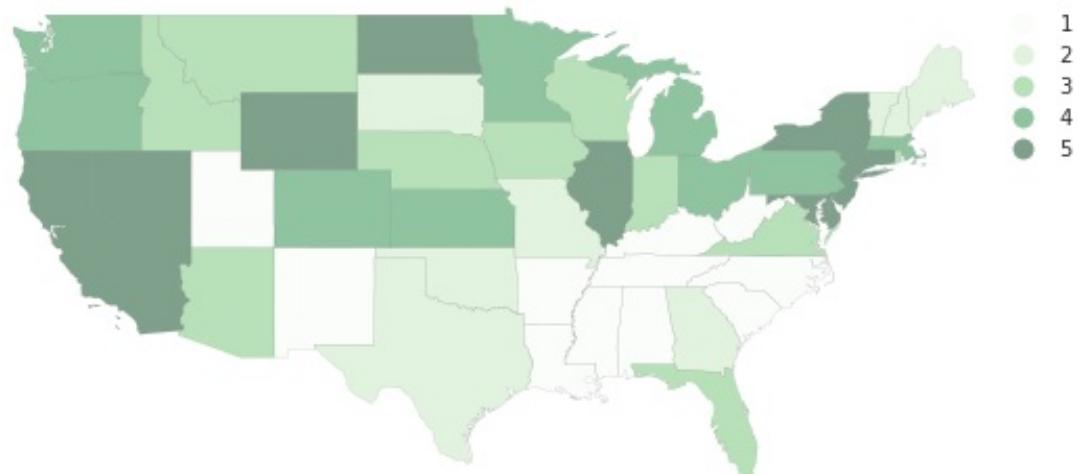
Per Capita Income 1972 Quintiles



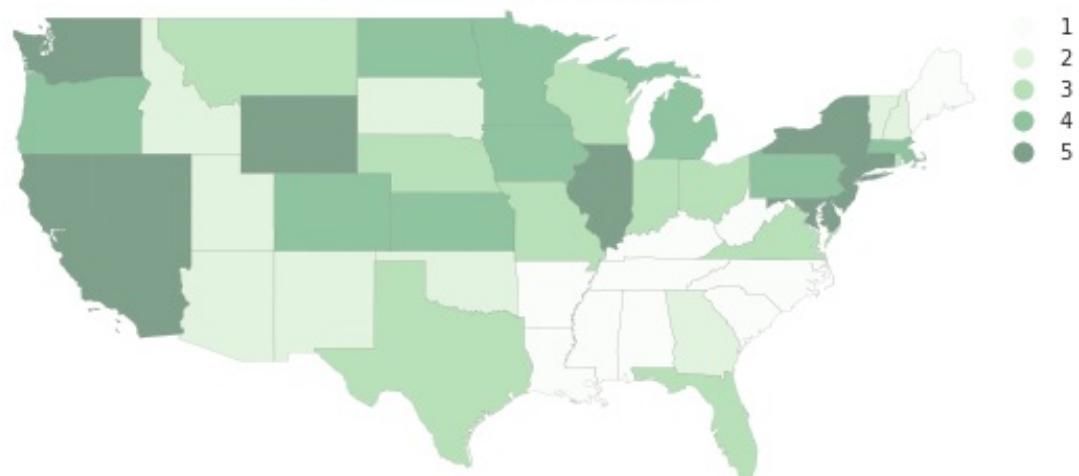
Per Capita Income 1973 Quintiles



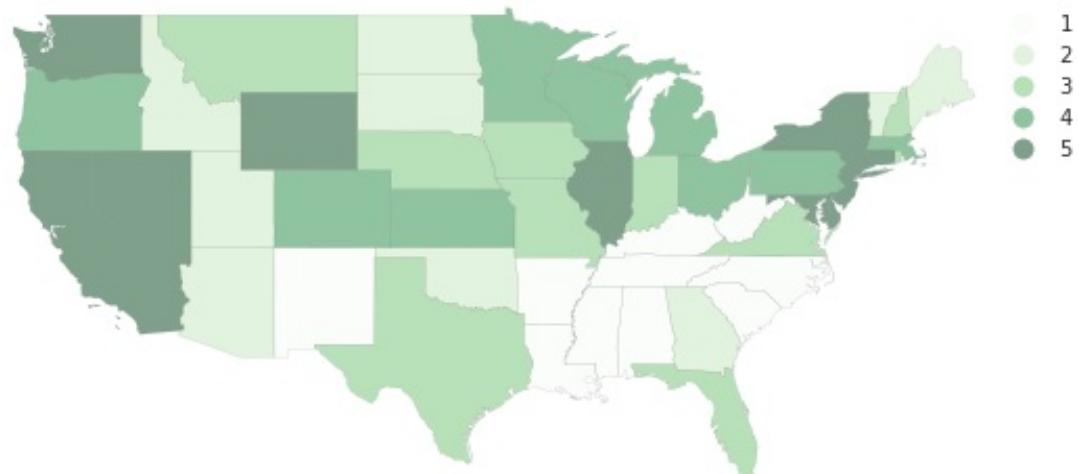
Per Capita Income 1974 Quintiles



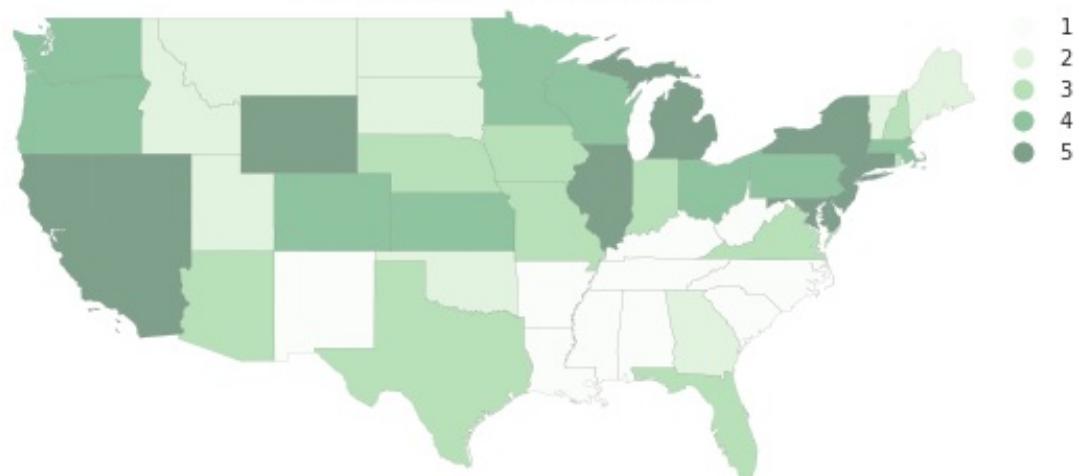
Per Capita Income 1975 Quintiles



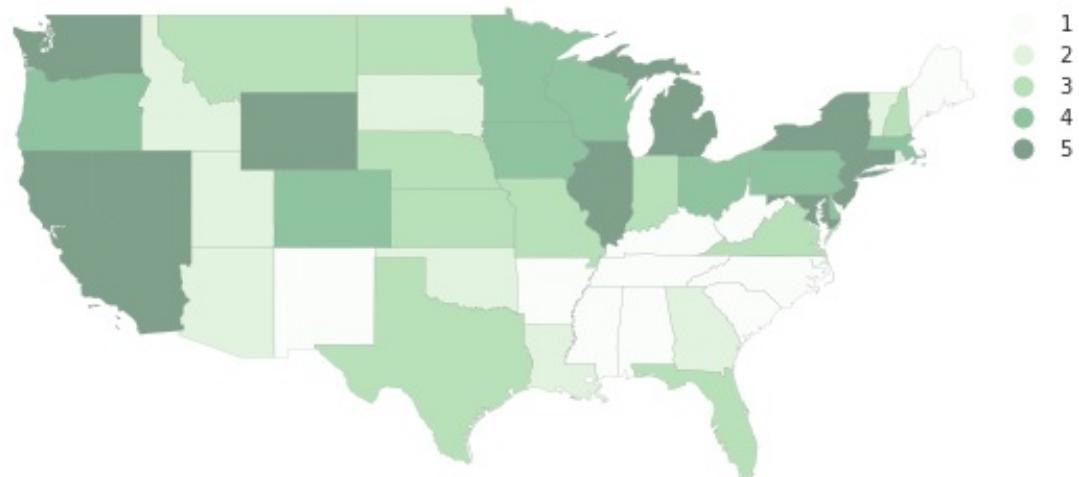
Per Capita Income 1976 Quintiles



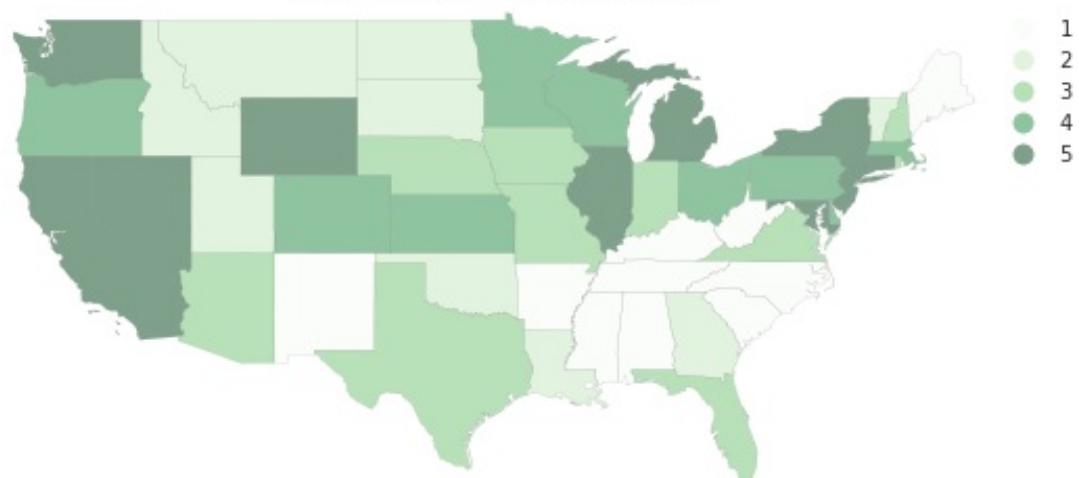
Per Capita Income 1977 Quintiles



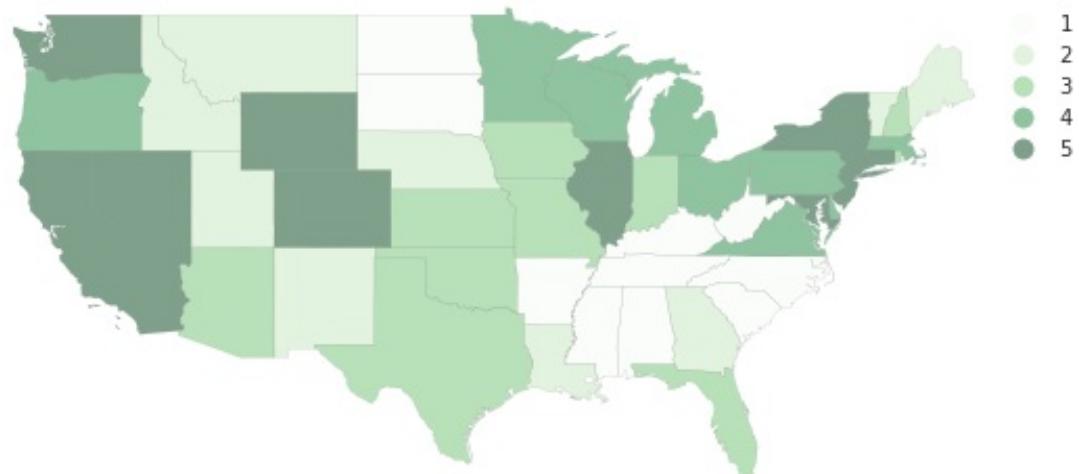
Per Capita Income 1978 Quintiles



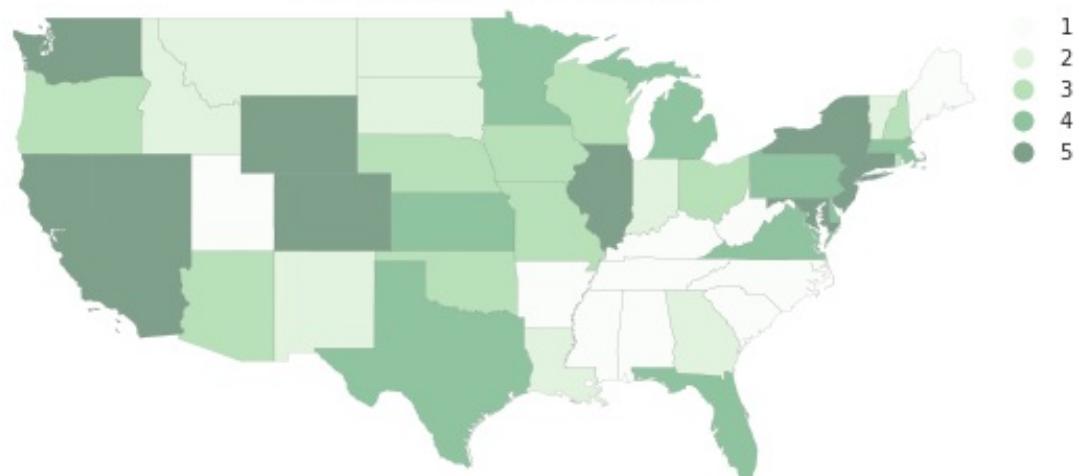
Per Capita Income 1979 Quintiles



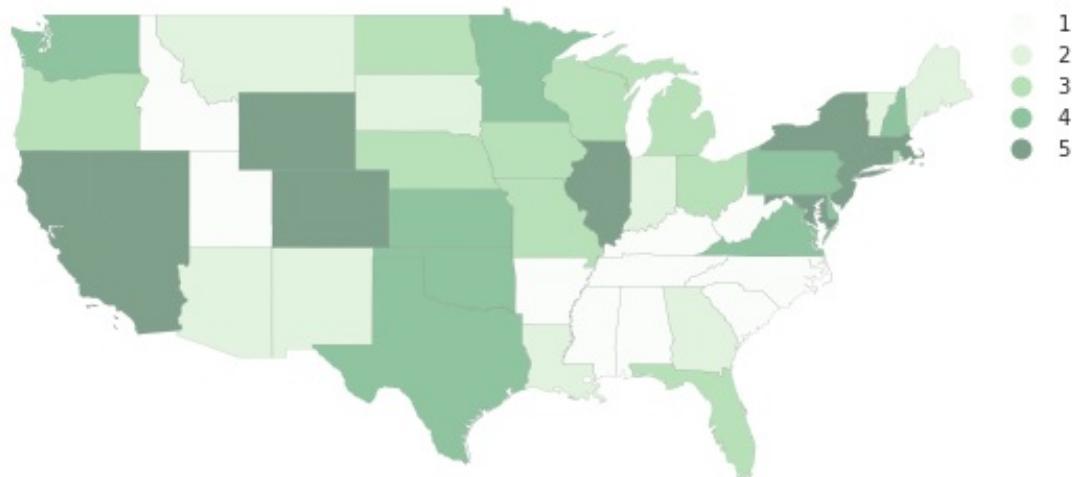
Per Capita Income 1980 Quintiles



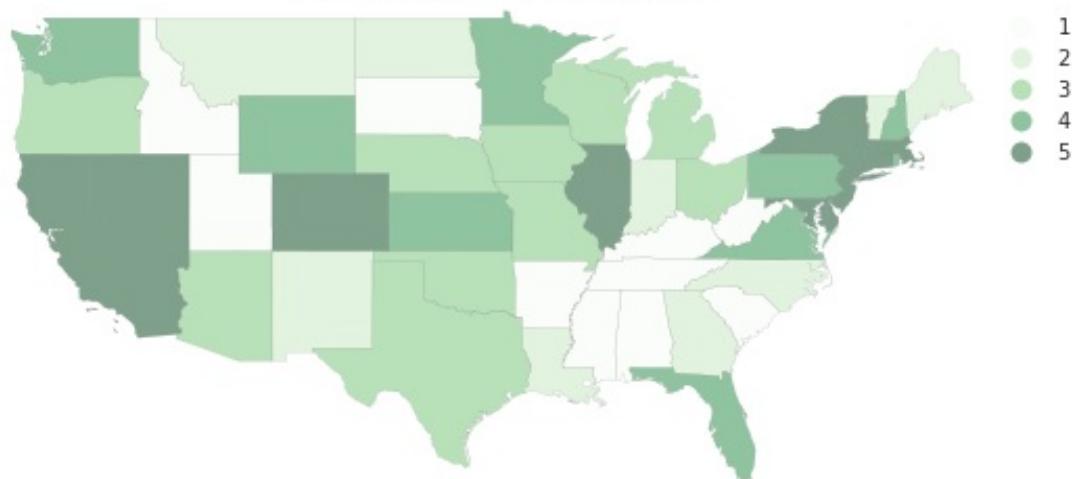
Per Capita Income 1981 Quintiles



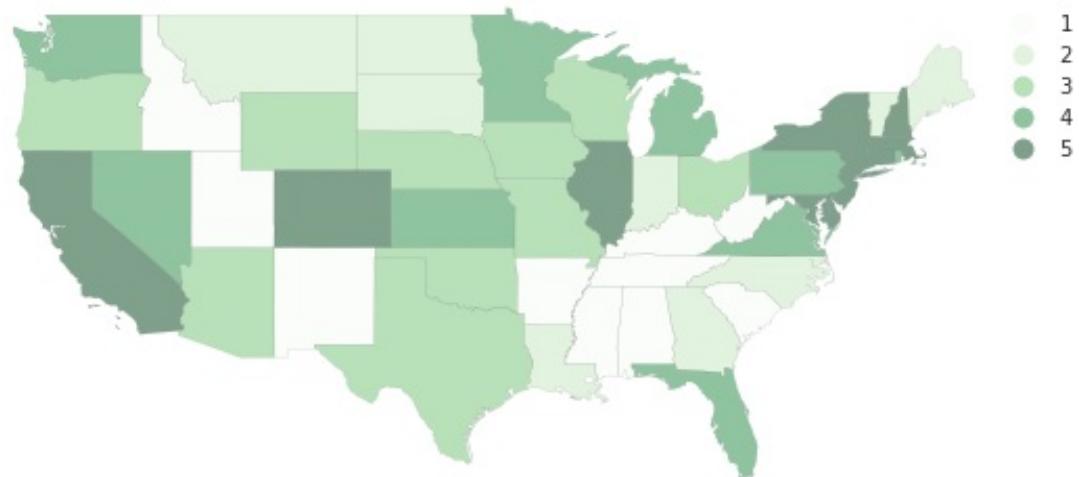
Per Capita Income 1982 Quintiles



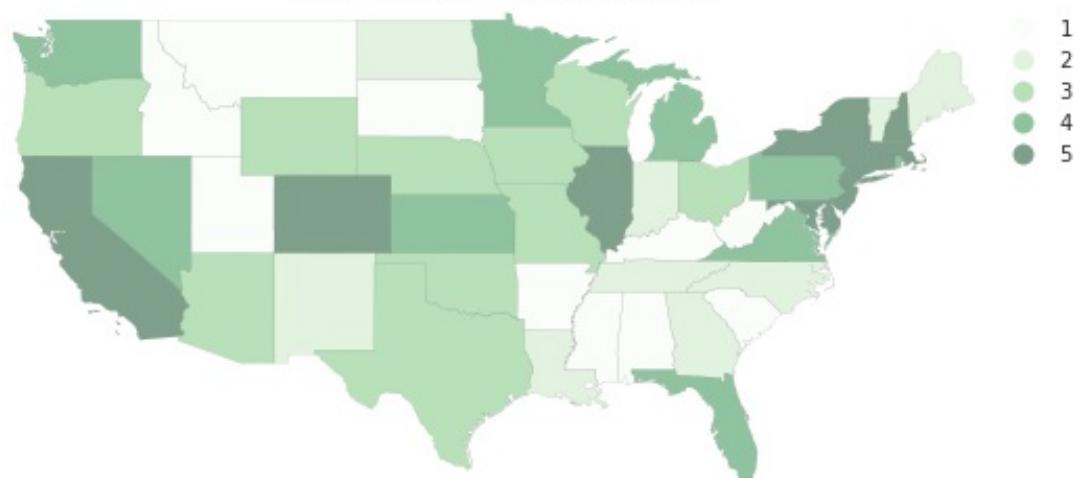
Per Capita Income 1983 Quintiles



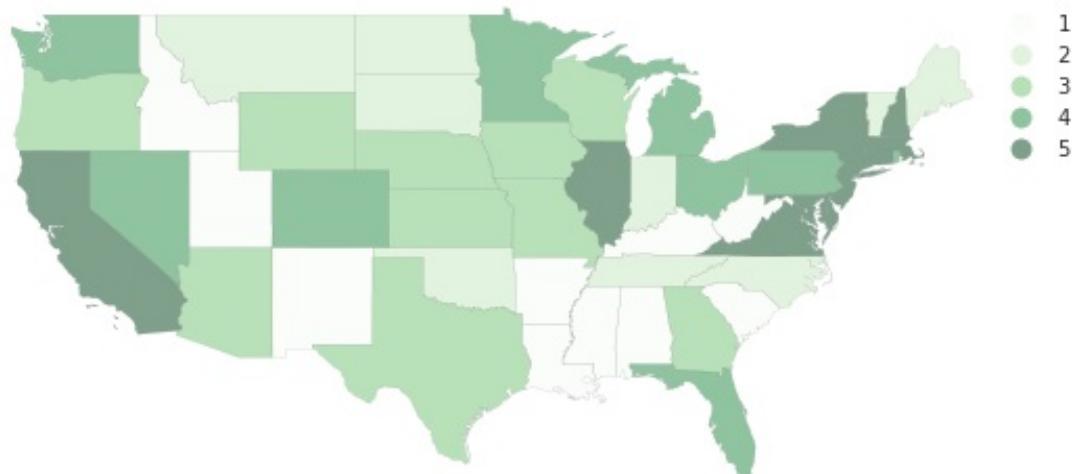
Per Capita Income 1984 Quintiles



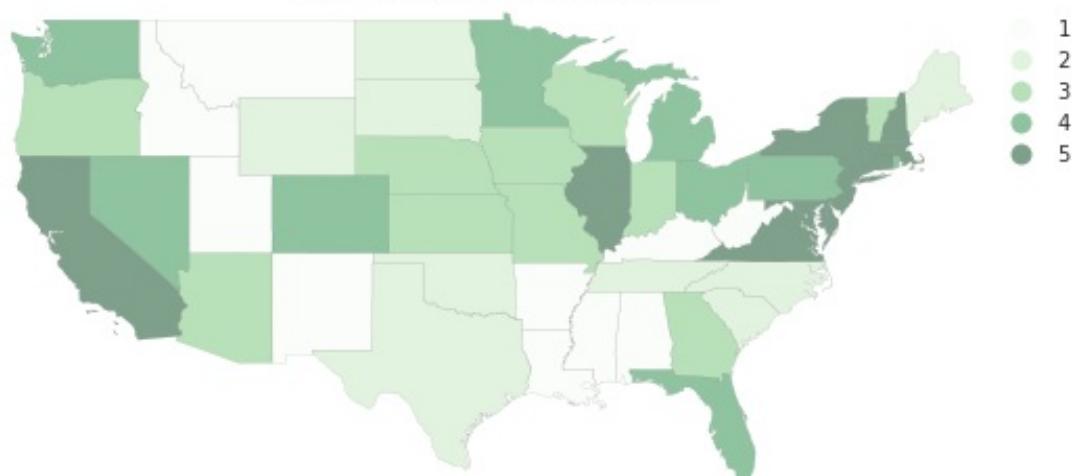
Per Capita Income 1985 Quintiles



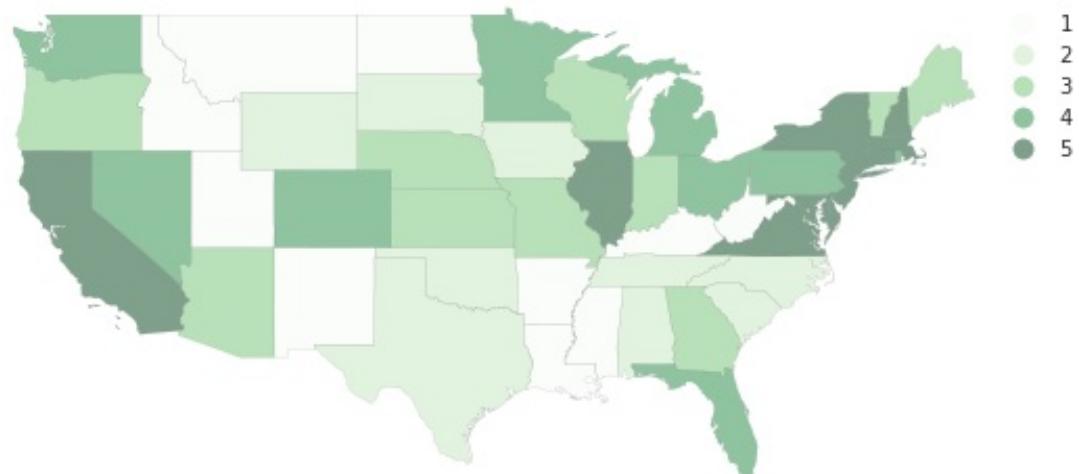
Per Capita Income 1986 Quintiles



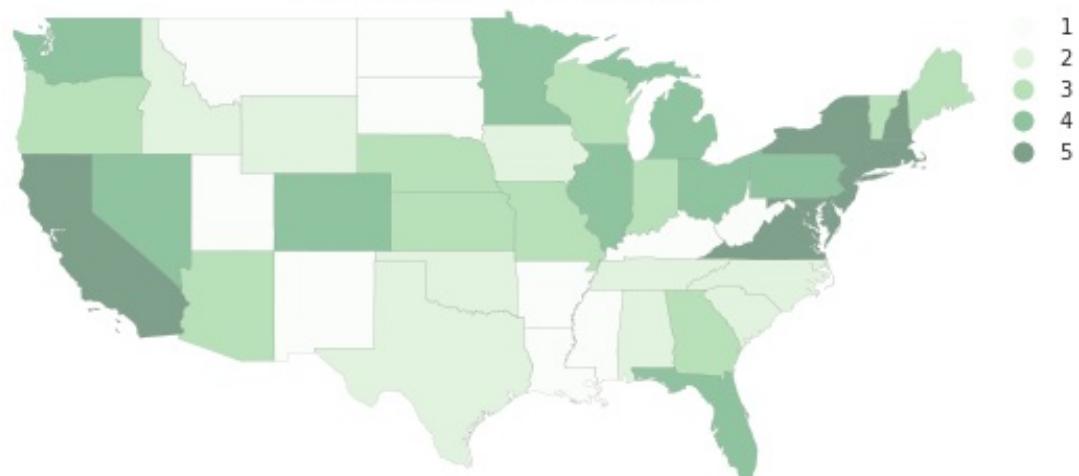
Per Capita Income 1987 Quintiles



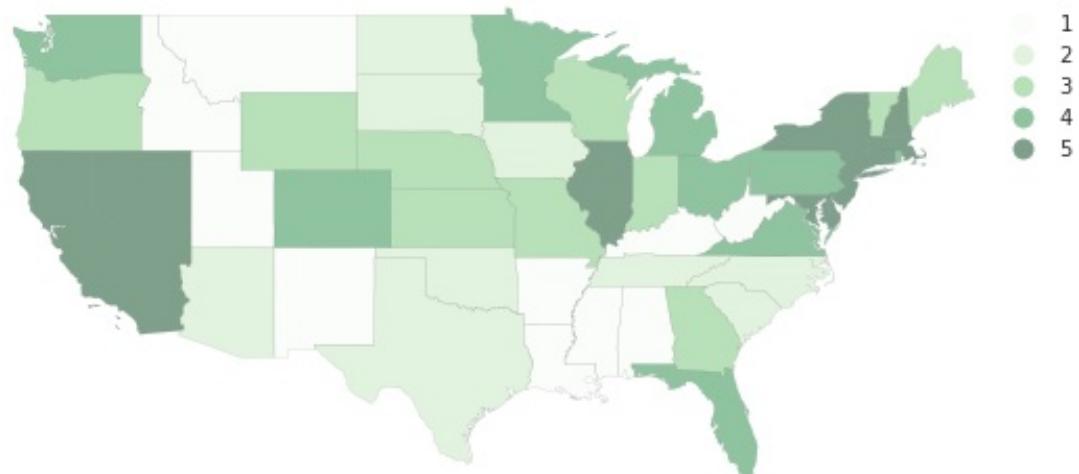
Per Capita Income 1988 Quintiles



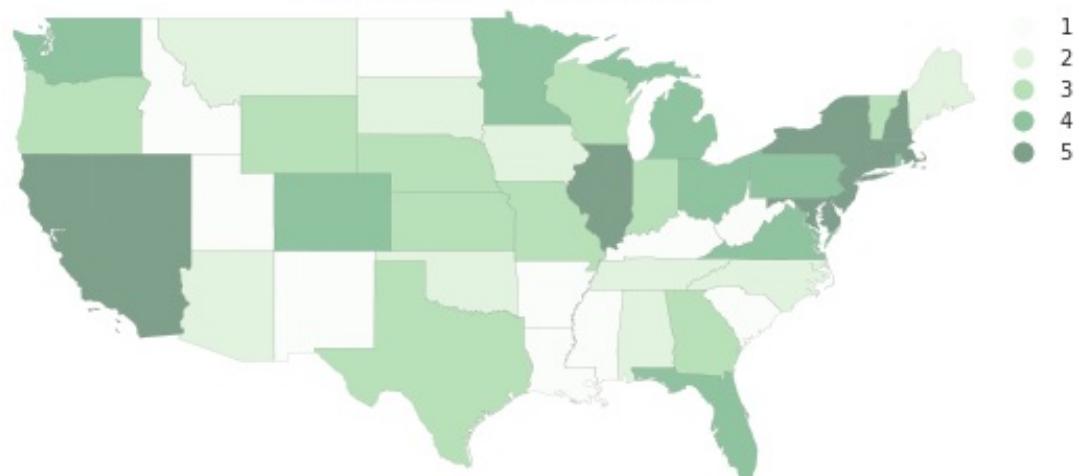
Per Capita Income 1989 Quintiles



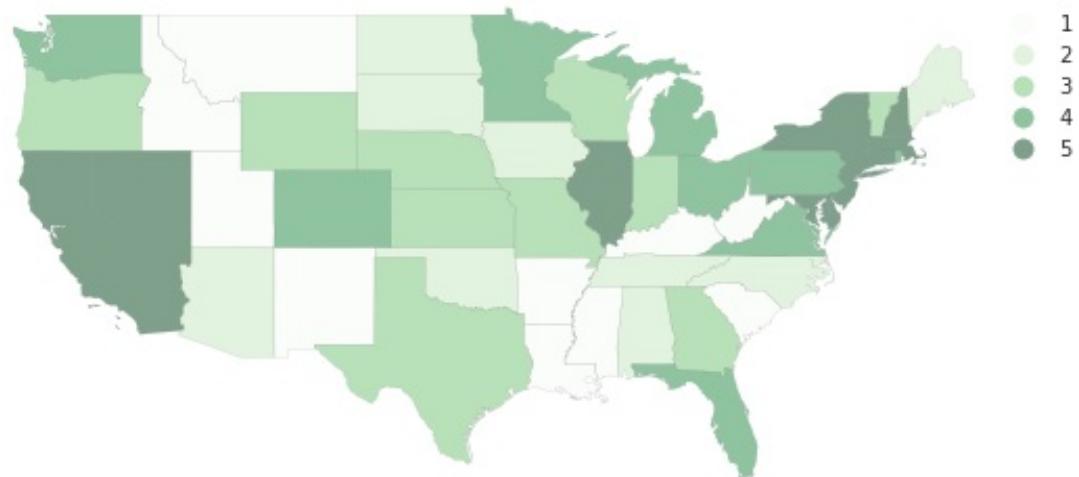
Per Capita Income 1990 Quintiles



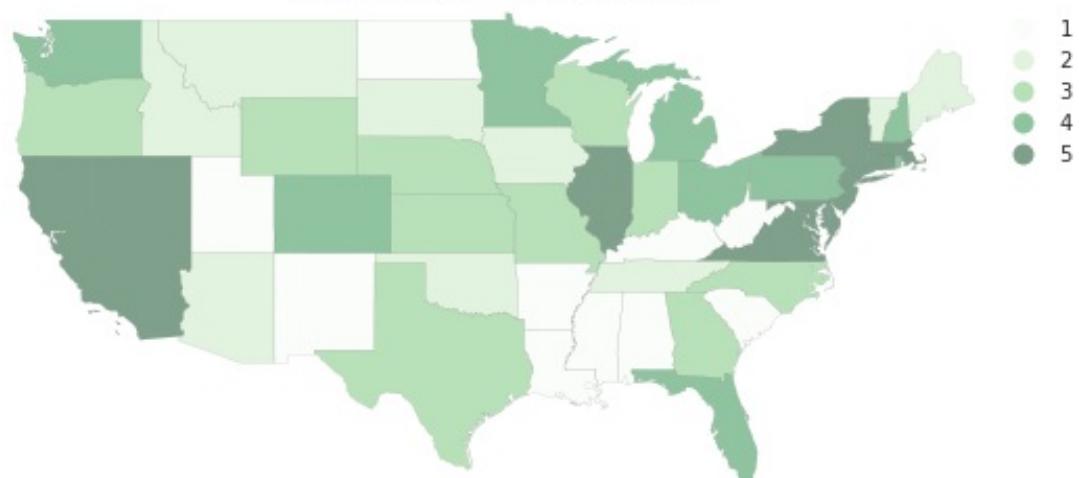
Per Capita Income 1991 Quintiles



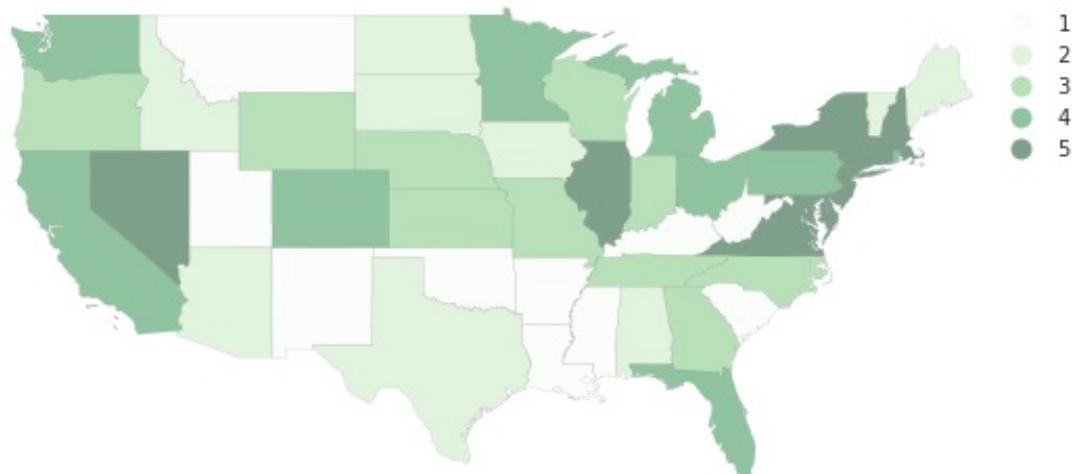
Per Capita Income 1992 Quintiles



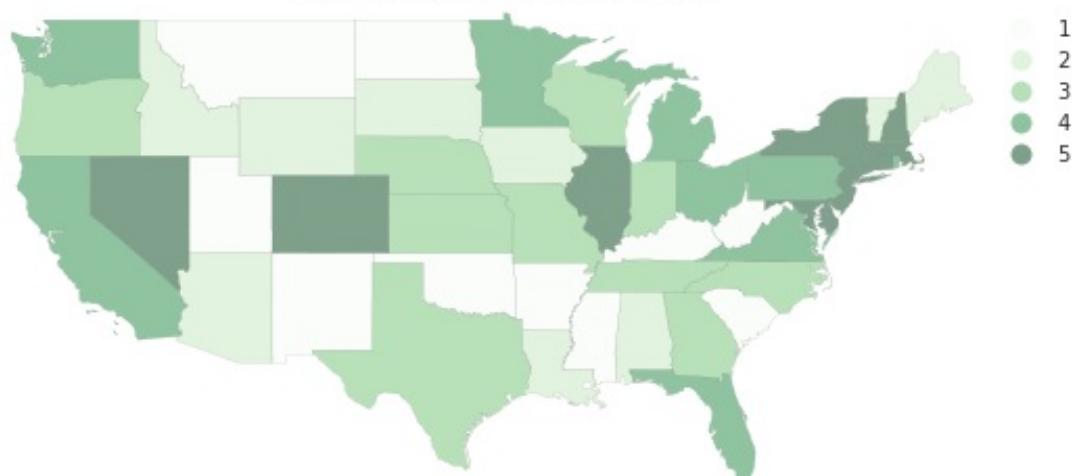
Per Capita Income 1993 Quintiles



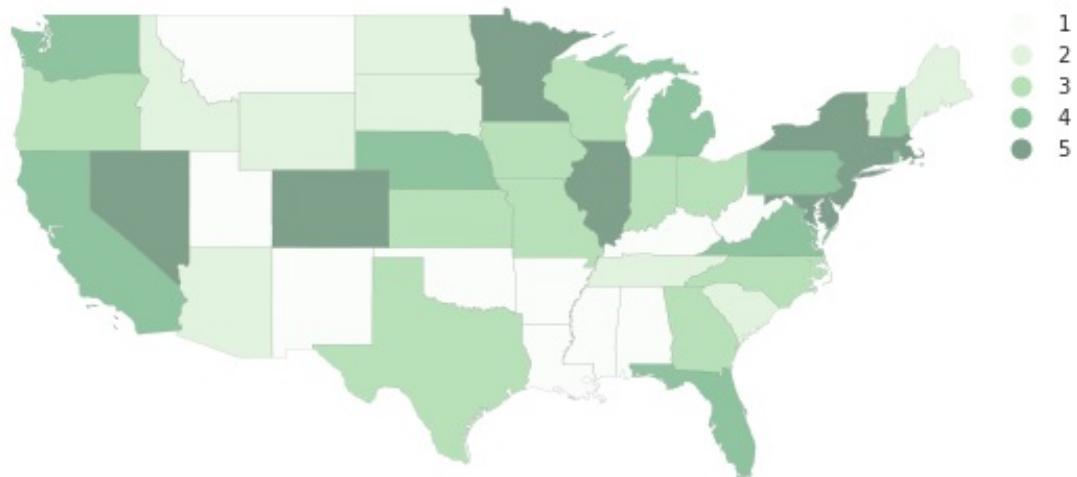
Per Capita Income 1994 Quintiles



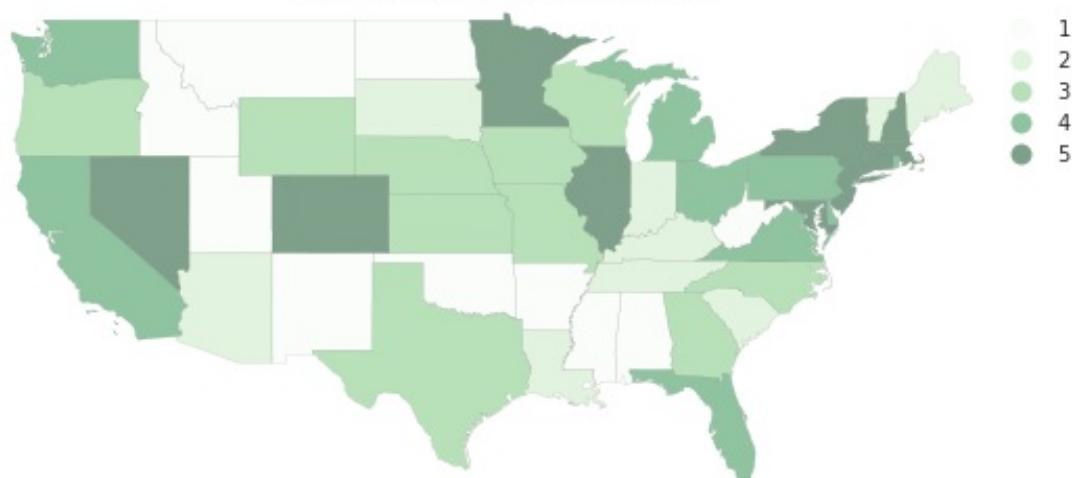
Per Capita Income 1995 Quintiles



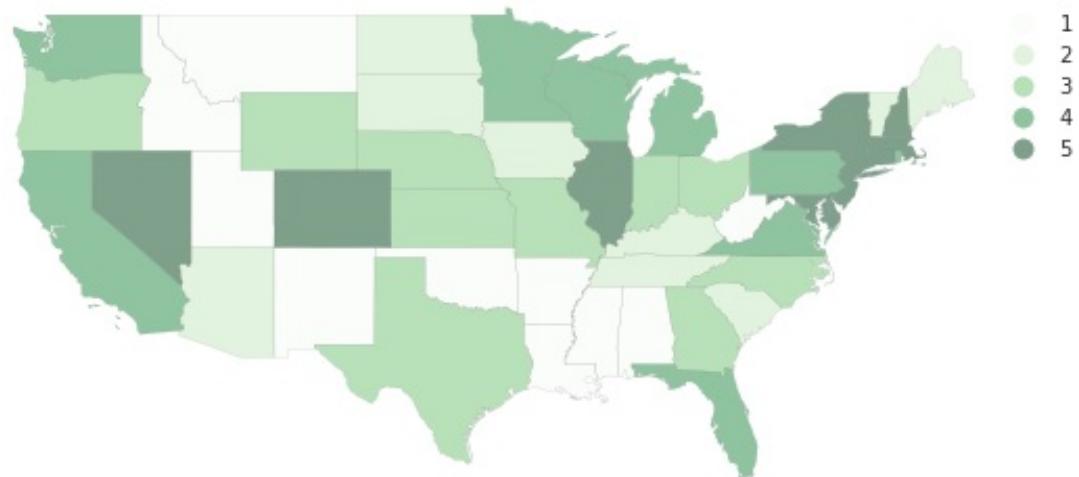
Per Capita Income 1996 Quintiles



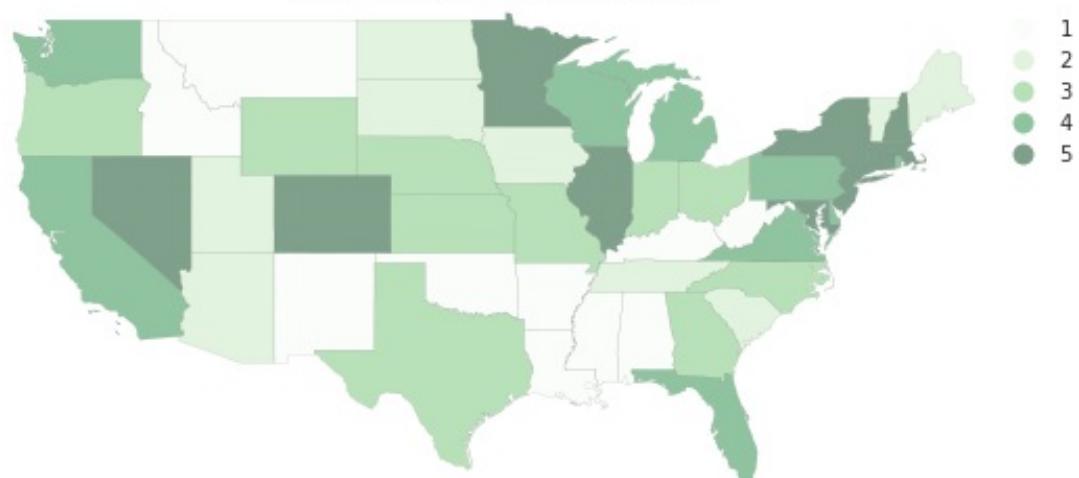
Per Capita Income 1997 Quintiles



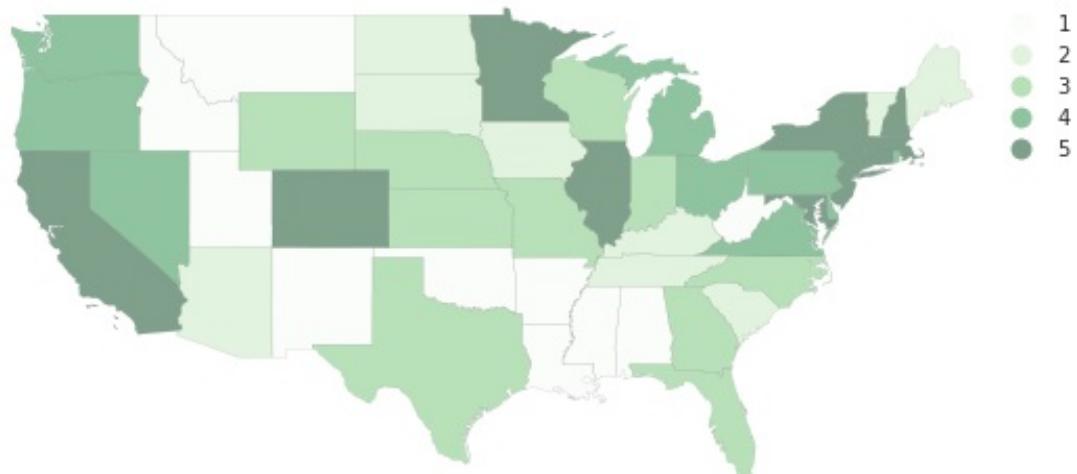
Per Capita Income 1998 Quintiles



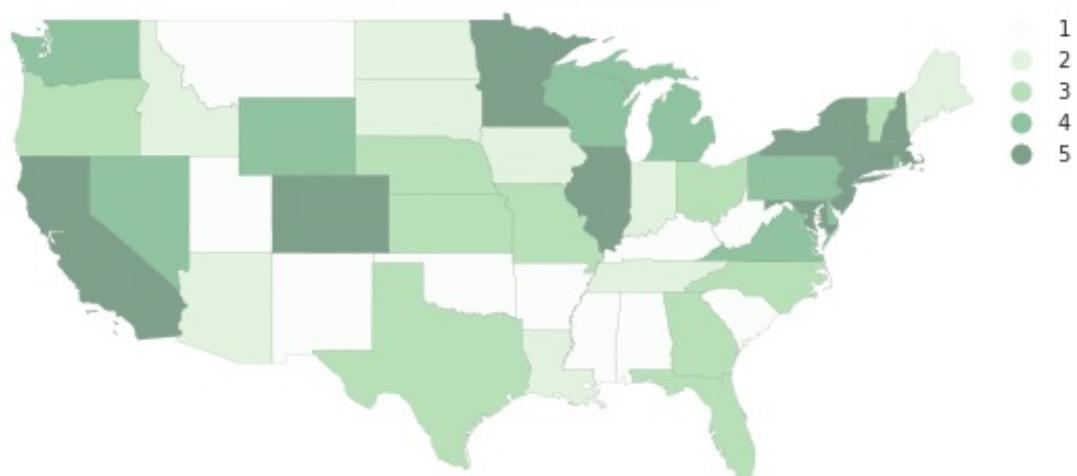
Per Capita Income 1999 Quintiles



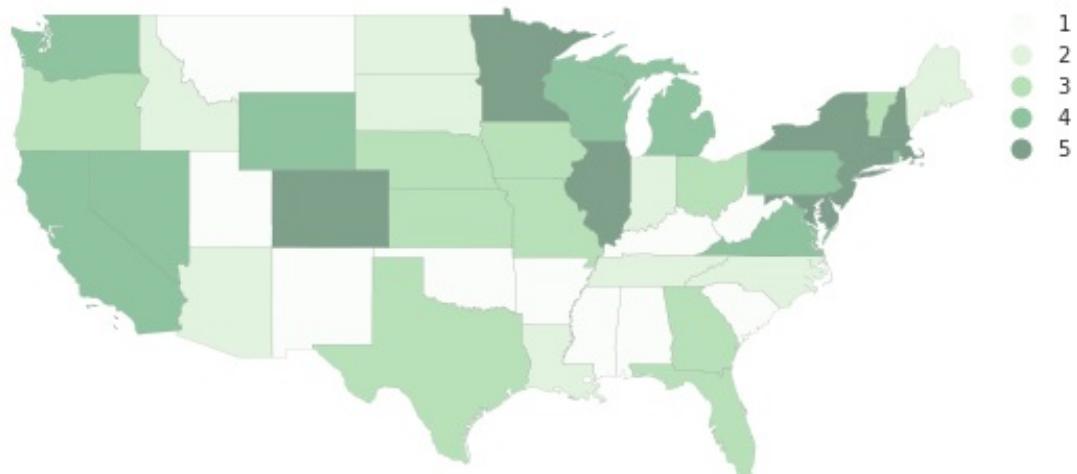
Per Capita Income 2000 Quintiles



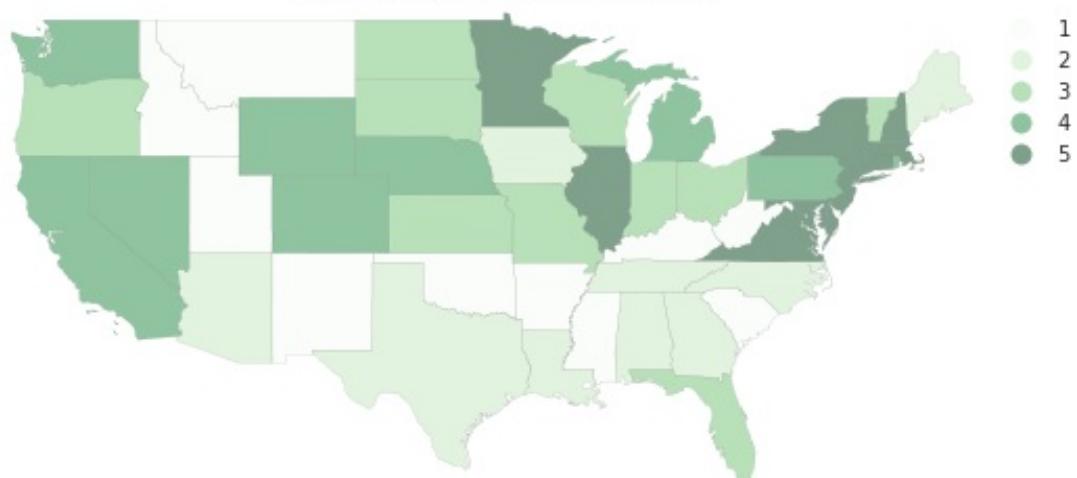
Per Capita Income 2001 Quintiles



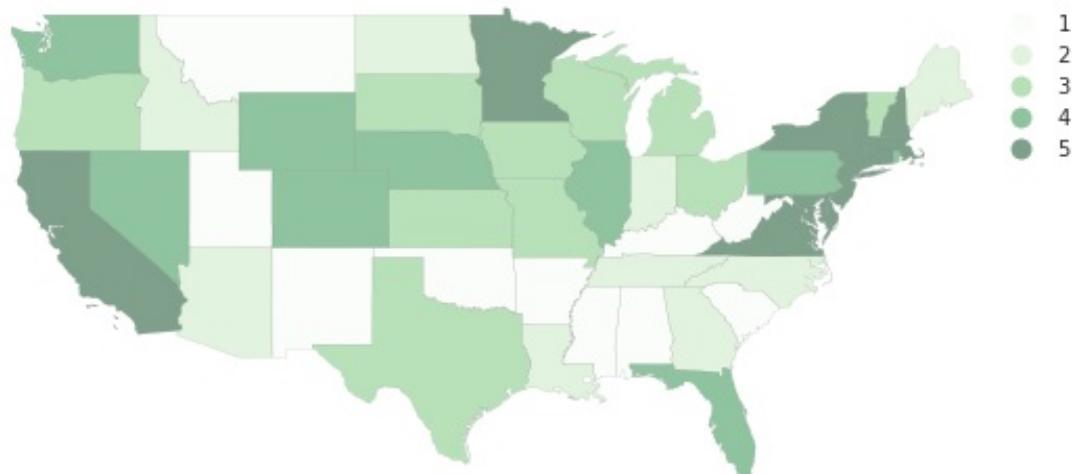
Per Capita Income 2002 Quintiles



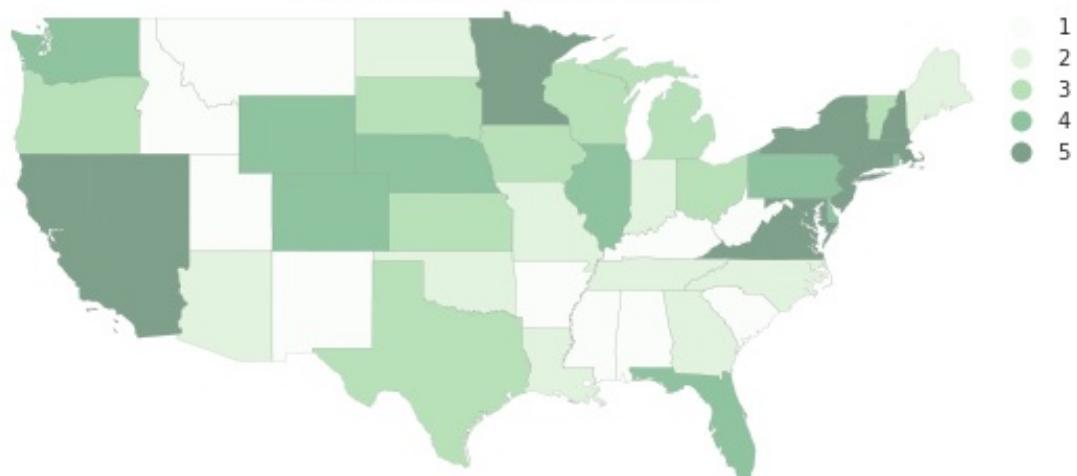
Per Capita Income 2003 Quintiles



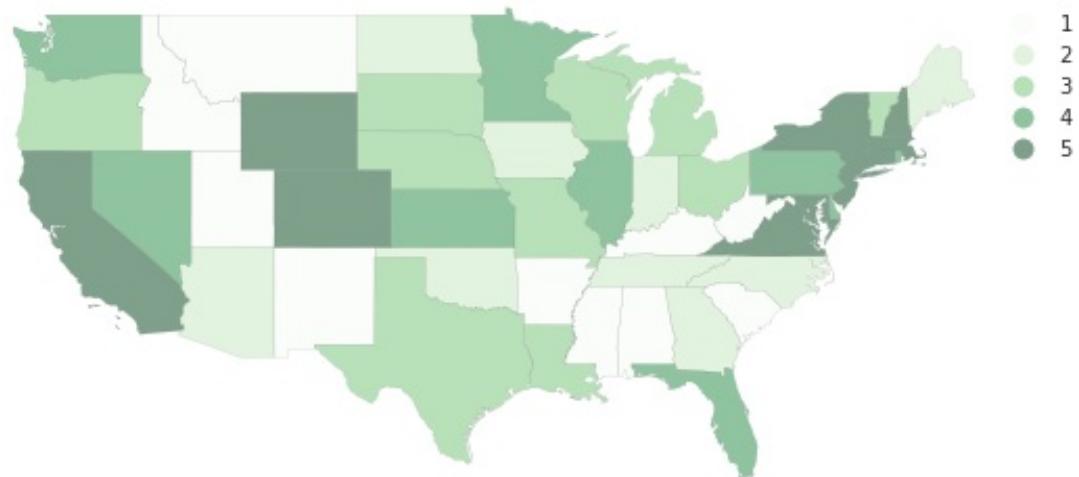
Per Capita Income 2004 Quintiles



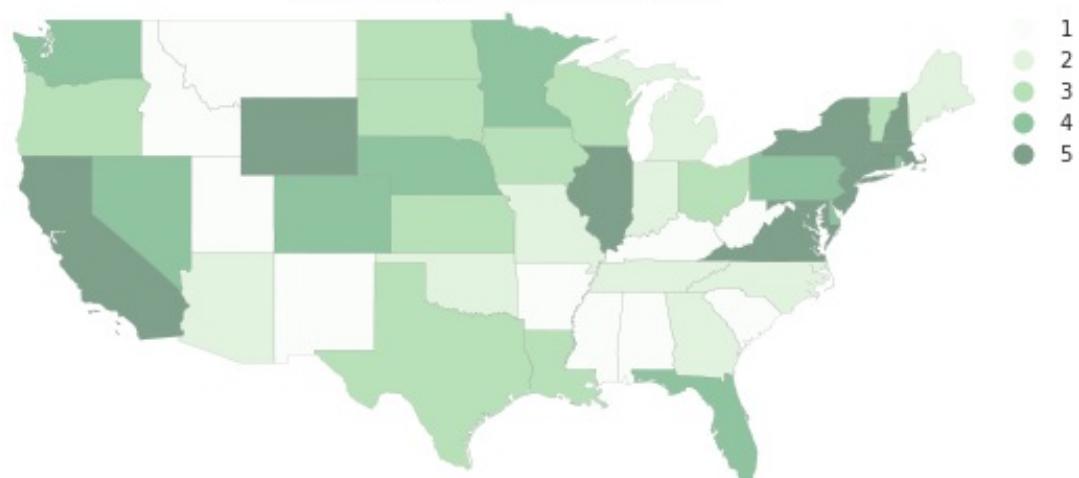
Per Capita Income 2005 Quintiles



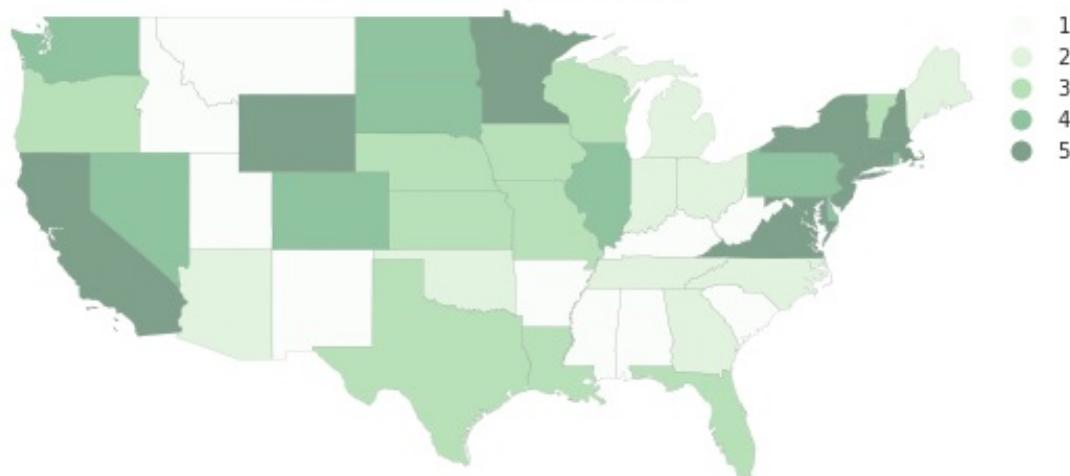
Per Capita Income 2006 Quintiles



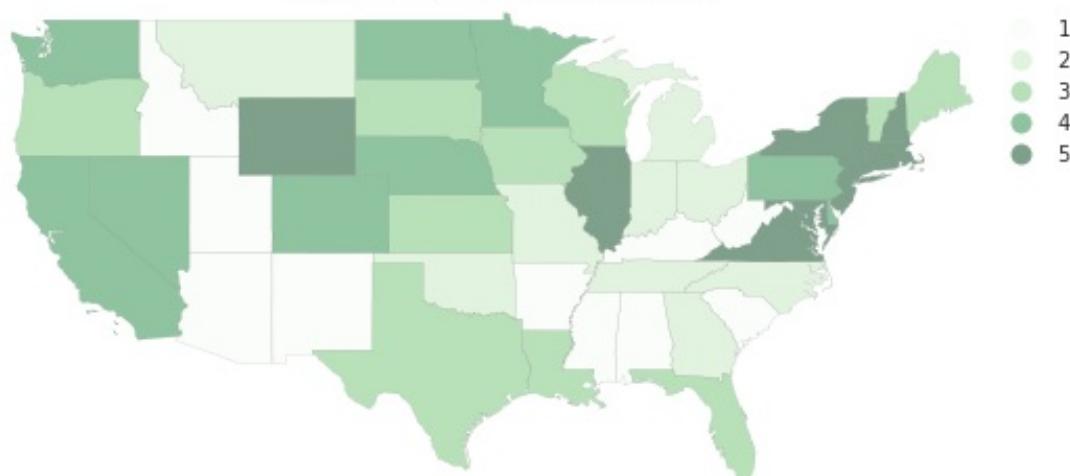
Per Capita Income 2007 Quintiles



Per Capita Income 2008 Quintiles



Per Capita Income 2009 Quintiles



Put series into cross-sectional quintiles (i.e., quintiles for each year).

```
q5 = np.array([ps.Quantiles(y).yb for y in pci.T]).transpose()
```

```
q5.shape
```

```
(48, 81)
```

```
q5[:, 0]
```

```
array([3, 2, 2, 0, 1, 3, 3, 1, 2, 2, 3, 3, 2, 4, 2, 4, 3, 4, 4, 4, 4, 2, 4, 2, 4, 2, 4, 4, 4, 1, 3, 2, 0, 1, 1, 2, 2, 1, 0, 0, 1, 0, 0, 0, 0, 0, 0, 0, 1, 1, 4])
```

```
pci.shape
```

```
(48, 81)
```

```
pci[0]
```

```
array([ 741,   658,   534,   402,   376,   443,   490,   569,   599,
       582,   614,   658,   864,  1196,  1469,  1527,  1419,  1401,
      1504,  1624,  1595,  1721,  1874,  1973,  2066,  2077,  2116,
      2172,  2262,  2281,  2380,  2436,  2535,  2680,  2735,  2858,
     3078,  3385,  3566,  3850,  4097,  4205,  4381,  4731,  5312,
     5919,  6533,  7181,  7832,  8887,  9965, 10913, 11903, 12431,
    13124, 14021, 14738, 15522, 16300, 17270, 18670, 20026, 20901,
    21917, 22414, 23119, 23878, 25287, 26817, 28632, 30392, 31528,
    32053, 32206, 32934, 34984, 35738, 38477, 40782, 41588, 40619])
```

we are looping over the rows of y which is ordered $T \times n$ (rows are cross sections, row 0 is the cross-section for period 0).

```
m5 = ps.Markov(q5)
```

```
m5.classes
```

```
array([0, 1, 2, 3, 4])
```

```
m5.transitions
```

```
array([[ 729.,   71.,    1.,    0.,    0.],
       [ 72.,  567.,   80.,    3.,    0.],
       [  0.,   81.,  631.,   86.,    2.],
       [  0.,    3.,   86.,  573.,   56.],
       [  0.,    0.,    1.,   57.,  741.]])
```

```
np.set_printoptions(3, suppress=True)
m5.p
```

```
matrix([[ 0.91 ,  0.089,  0.001,  0.    ,  0.    ],
       [ 0.1   ,  0.785,  0.111,  0.004,  0.    ],
       [ 0.    ,  0.101,  0.789,  0.107,  0.003],
       [ 0.    ,  0.004,  0.12 ,  0.798,  0.078],
       [ 0.    ,  0.    ,  0.001,  0.071,  0.927]])
```

```
m5.steady_state #steady state distribution
```

```
matrix([[ 0.208],
       [ 0.187],
       [ 0.207],
       [ 0.188],
       [ 0.209]])
```

```
fmp = ps.ergodic.fmp(m5.p) #first mean passage time
fmp
```

```
matrix([[ 4.814,  11.503,  29.609,  53.386, 103.598],
       [ 42.048,    5.34 ,  18.745,   42.5  ,  92.713],
       [ 69.258,  27.211,   4.821,  25.272,  75.433],
       [ 84.907,  42.859,  17.181,   5.313,   51.61 ],
       [ 98.413,  56.365,  30.66 ,  14.212,   4.776]])
```

For a state with income in the first quintile, it takes on average 11.5 years for it to first enter the second quintile, 29.6 to get to the third quintile, 53.4 years to enter the fourth, and 103.6 years to reach the richest quintile.

But, this approach assumes the movement of a state in the income distribution is independent of the movement of its neighbors or the position of the neighbors in the distribution. Does spatial context matter?

Dynamics of Spatial Dependence

Create a queen contiguity matrix that is row standardized

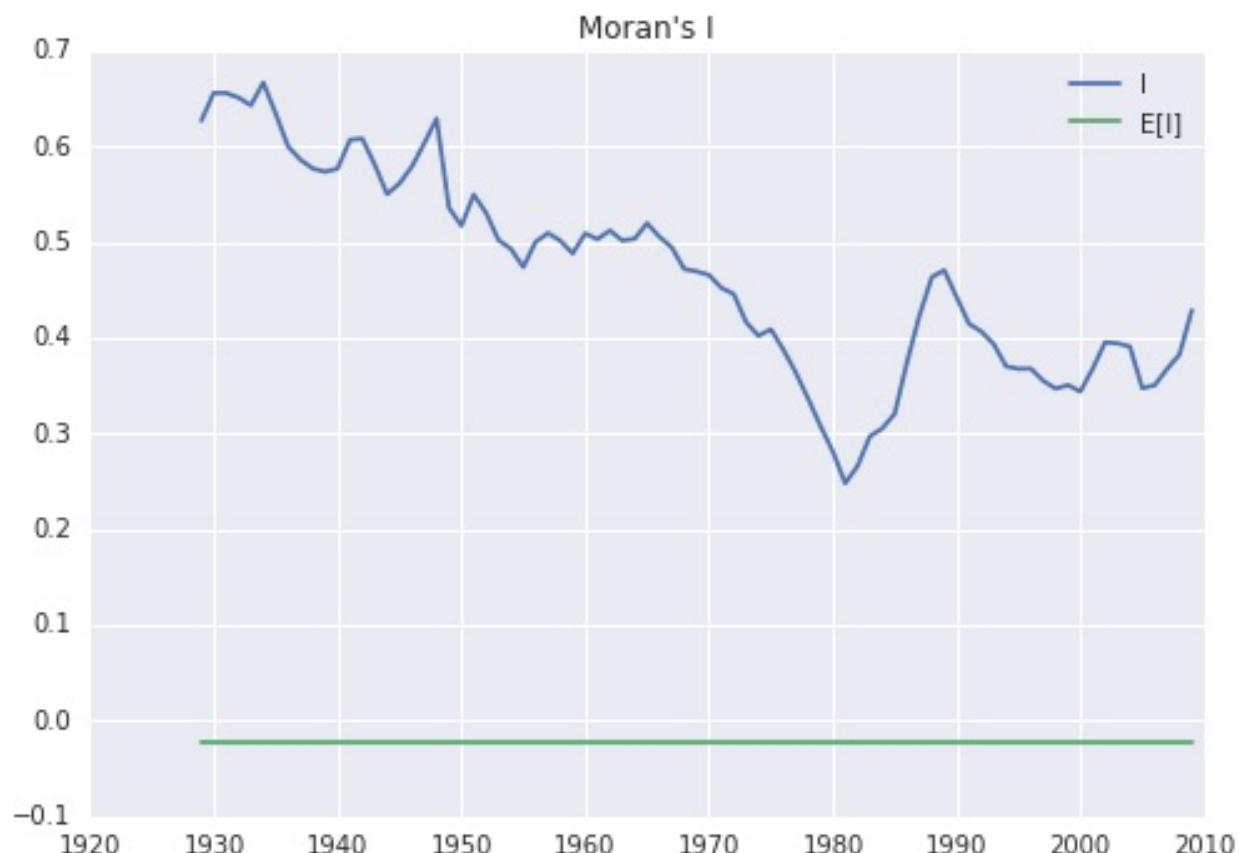
```
w = ps.queen_from_shapefile(ps.examples.get_path('us48.shp'))
w.transform = 'R'
```

```
mits = [ps.Moran(cs, w) for cs in RY.T]
```

```
res = np.array([(m.I, m.EI, m.p_sim, m.z_sim) for m in mits])
```

```
plt.plot(years, res[:,0], label='I')
plt.plot(years, res[:,1], label='E[I]')
plt.title("Moran's I")
plt.legend()
```

```
<matplotlib.legend.Legend at 0x7fc1189b5e80>
```



```
plt.plot(years, res[:, -1])
plt.ylim(0, 7.0)
plt.title('z-values, I')
```

```
<matplotlib.text.Text at 0x7fc129b46b70>
```



Spatial Markov

```
pci.shape
```

```
(48, 81)
```

```
rpci = pci / pci.mean(axis=0)
```

```
rpci[:, 0]
```

```
array([ 1.204,  0.962,  0.977,  0.621,  0.692,  1.097,  1.094,  0.824,
       1.031,  0.974,  1.086,  1.115,  0.944,  1.473,  0.969,  1.873,
       1.255,  1.664,  1.421,  1.492,  0.987,  1.411,  0.896,  1.611,
       1.253,  1.541,  1.677,  0.748,  1.248,  1.031,  0.639,  0.865,
       0.705,  1.009,  0.975,  0.74 ,  0.54 ,  0.614,  0.779,  0.666,
       0.525,  0.465,  0.564,  0.441,  0.504,  0.673,  0.842,  1.284])
```

```
rpci[:,0].mean()
```

```
0.9999999999999999
```

```
sm = ps.Spatial_Markov(rpci, W, fixed=True, k=5)
```

```
sm.p
```

```
matrix([[ 0.915,  0.075,  0.009,  0.001,  0.    ],
       [ 0.066,  0.827,  0.105,  0.001,  0.001],
       [ 0.005,  0.103,  0.794,  0.095,  0.003],
       [ 0.    ,  0.009,  0.094,  0.849,  0.048],
       [ 0.    ,  0.    ,  0.    ,  0.062,  0.938]])
```

```
for p in sm.P:
    print(p)
```

```
[[ 0.963  0.03   0.006  0.      0.      ]
 [ 0.06   0.832   0.107  0.      0.      ]
 [ 0.     0.14    0.74   0.12   0.      0.      ]
 [ 0.     0.036   0.321   0.571  0.071]
 [ 0.     0.      0.     0.167  0.833]]]

[[ 0.798  0.168   0.034  0.      0.      ]
 [ 0.075  0.882   0.042  0.      0.      ]
 [ 0.005  0.07    0.866   0.059  0.      0.      ]
 [ 0.     0.      0.064   0.902  0.034]
 [ 0.     0.      0.     0.194  0.806]]]

[[ 0.847  0.153   0.      0.      0.      0.      ]
 [ 0.081  0.789   0.129  0.      0.      0.      ]
 [ 0.005  0.098   0.793   0.098  0.005]
 [ 0.     0.      0.094   0.871  0.035]
 [ 0.     0.      0.     0.102  0.898]]]

[[ 0.885  0.098   0.      0.016  0.      0.      ]
 [ 0.039  0.814   0.14   0.     0.008]
 [ 0.005  0.094   0.777   0.119  0.005]
 [ 0.     0.023   0.129   0.754  0.094]
 [ 0.     0.      0.     0.097  0.903]]]

[[ 0.333  0.667   0.      0.      0.      0.      ]
 [ 0.048  0.774   0.161   0.016  0.      0.      ]
 [ 0.011  0.161   0.747   0.08   0.      0.      ]
 [ 0.     0.01   0.062   0.896  0.031]
 [ 0.     0.      0.     0.024  0.976]]]
```

sm.S

```
array([[ 0.435,   0.264,   0.204,   0.068,   0.029],
       [ 0.134,   0.34 ,   0.252,   0.233,   0.041],
       [ 0.121,   0.211,   0.264,   0.29 ,   0.114],
       [ 0.078,   0.197,   0.254,   0.225,   0.247],
       [ 0.018,   0.2 ,   0.19 ,   0.255,   0.337]])
```

```
for f in sm.F:
    print(f)
```

```
[[ 2.298 28.956 46.143 80.81 279.429]
 [ 33.865 3.795 22.571 57.238 255.857]
 [ 43.602 9.737 4.911 34.667 233.286]
 [ 46.629 12.763 6.257 14.616 198.619]
 [ 52.629 18.763 12.257 6. 34.103]]
[[ 7.468 9.706 25.768 74.531 194.234]
 [ 27.767 2.942 24.971 73.735 193.438]
 [ 53.575 28.484 3.976 48.763 168.467]
 [ 72.036 46.946 18.462 4.284 119.703]
 [ 77.179 52.089 23.604 5.143 24.276]]
[[ 8.248 6.533 18.388 40.709 112.767]
 [ 47.35 4.731 11.854 34.175 106.234]
 [ 69.423 24.767 3.795 22.321 94.38 ]
 [ 83.723 39.067 14.3 3.447 76.367]
 [ 93.523 48.867 24.1 9.8 8.793]]
[[ 12.88 13.348 19.834 28.473 55.824]
 [ 99.461 5.064 10.545 23.051 49.689]
 [ 117.768 23.037 3.944 15.084 43.579]
 [ 127.898 32.439 14.569 4.448 31.631]
 [ 138.248 42.789 24.919 10.35 4.056]]
[[ 56.282 1.5 10.572 27.022 110.543]
 [ 82.922 5.009 9.072 25.522 109.043]
 [ 97.177 19.531 5.26 21.424 104.946]
 [ 127.141 48.741 33.296 3.918 83.522]
 [ 169.641 91.241 75.796 42.5 2.965]]
```

```
sm.summary()
```

```
-----
          Spatial Markov Test
-----
Number of classes: 5
Number of transitions: 3840
Number of regimes: 5
Regime names: LAG0, LAG1, LAG2, LAG3, LAG4
-----
      Test           LR           Chi-2
Stat.       170.659      200.624
      DOF            60            60
p-value      0.000      0.000
-----
      P(H0)        C0         C1         C2         C3         C4
      C0     0.915     0.075     0.009     0.001     0.000
      C1     0.066     0.827     0.105     0.001     0.001
      C2     0.005     0.103     0.794     0.095     0.003
      C3     0.000     0.009     0.094     0.849     0.048
      C4     0.000     0.000     0.000     0.062     0.938
-----
```

P(LAG0)	C0	C1	C2	C3	C4
C0	0.963	0.030	0.006	0.000	0.000
C1	0.060	0.832	0.107	0.000	0.000
C2	0.000	0.140	0.740	0.120	0.000
C3	0.000	0.036	0.321	0.571	0.071
C4	0.000	0.000	0.000	0.167	0.833

P(LAG1)	C0	C1	C2	C3	C4
C0	0.798	0.168	0.034	0.000	0.000
C1	0.075	0.882	0.042	0.000	0.000
C2	0.005	0.070	0.866	0.059	0.000
C3	0.000	0.000	0.064	0.902	0.034
C4	0.000	0.000	0.000	0.194	0.806

P(LAG2)	C0	C1	C2	C3	C4
C0	0.847	0.153	0.000	0.000	0.000
C1	0.081	0.789	0.129	0.000	0.000
C2	0.005	0.098	0.793	0.098	0.005
C3	0.000	0.000	0.094	0.871	0.035
C4	0.000	0.000	0.000	0.102	0.898

P(LAG3)	C0	C1	C2	C3	C4
C0	0.885	0.098	0.000	0.016	0.000
C1	0.039	0.814	0.140	0.000	0.008
C2	0.005	0.094	0.777	0.119	0.005
C3	0.000	0.023	0.129	0.754	0.094
C4	0.000	0.000	0.000	0.097	0.903

P(LAG4)	C0	C1	C2	C3	C4
C0	0.333	0.667	0.000	0.000	0.000
C1	0.048	0.774	0.161	0.016	0.000
C2	0.011	0.161	0.747	0.080	0.000
C3	0.000	0.010	0.062	0.896	0.031
C4	0.000	0.000	0.000	0.024	0.976

Part II

Point Patterns

IPYNB

NOTE: some of this material has been ported and adapted from "Lab 9" in [Arribas-Bel \(2016\)](#).

This notebook covers a brief introduction on how to visualize and analyze point patterns. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import numpy as np
import pandas as pd
import geopandas as gpd
import seaborn as sns
import matplotlib.pyplot as plt
import folium
```

Data preparation

Let us first set the paths to the datasets we will be using:

```
# Adjust this to point to the right file in your computer
listings_link = '../data/listings.csv.gz'
```

The core dataset we will use is `listings.csv`, which contains a lot of information about each individual location listed at AirBnb within Austin:

```
lst = pd.read_csv(listings_link)
lst.info()

<class 'pandas.core.frame.DataFrame'>
RangeIndex: 5835 entries, 0 to 5834
Data columns (total 92 columns):
id                      5835 non-null int64
listing_url              5835 non-null object
```

scrape_id	5835 non-null int64
last_scraped	5835 non-null object
name	5835 non-null object
summary	5373 non-null object
space	4475 non-null object
description	5832 non-null object
experiences_offered	5835 non-null object
neighborhood_overview	3572 non-null object
notes	2413 non-null object
transit	3492 non-null object
thumbnail_url	5542 non-null object
medium_url	5542 non-null object
picture_url	5835 non-null object
xl_picture_url	5542 non-null object
host_id	5835 non-null int64
host_url	5835 non-null object
host_name	5820 non-null object
host_since	5820 non-null object
host_location	5810 non-null object
host_about	3975 non-null object
host_response_time	4177 non-null object
host_response_rate	4177 non-null object
host_acceptance_rate	3850 non-null object
host_is_superhost	5820 non-null object
host_thumbnail_url	5820 non-null object
host_picture_url	5820 non-null object
host_neighbourhood	4977 non-null object
host_listings_count	5820 non-null float64
host_total_listings_count	5820 non-null float64
host_verifications	5835 non-null object
host_has_profile_pic	5820 non-null object
host_identity_verified	5820 non-null object
street	5835 non-null object
neighbourhood	4800 non-null object
neighbourhood_cleansed	5835 non-null int64
neighbourhood_group_cleansed	0 non-null float64
city	5835 non-null object
state	5835 non-null object
zipcode	5810 non-null float64
market	5835 non-null object
smart_location	5835 non-null object
country_code	5835 non-null object
country	5835 non-null object
latitude	5835 non-null float64
longitude	5835 non-null float64
is_location_exact	5835 non-null object
property_type	5835 non-null object
room_type	5835 non-null object
accommodates	5835 non-null int64
bathrooms	5789 non-null float64
bedrooms	5829 non-null float64
beds	5812 non-null float64

```

bed_type                      5835 non-null object
amenities                     5835 non-null object
square_feet                   302 non-null float64
price                         5835 non-null object
weekly_price                  2227 non-null object
monthly_price                 1717 non-null object
security_deposit              2770 non-null object
cleaning_fee                  3587 non-null object
guests_included               5835 non-null int64
extra_people                  5835 non-null object
minimum_nights                5835 non-null int64
maximum_nights                5835 non-null int64
calendar_updated              5835 non-null object
has_availability              5835 non-null object
availability_30                5835 non-null int64
availability_60                5835 non-null int64
availability_90                5835 non-null int64
availability_365               5835 non-null int64
calendar_last_scraped         5835 non-null object
number_of_reviews              5835 non-null int64
first_review                  3827 non-null object
last_review                   3829 non-null object
review_scores_rating           3789 non-null float64
review_scores_accuracy         3776 non-null float64
review_scores_cleanliness      3778 non-null float64
review_scores_checkin          3778 non-null float64
review_scores_communication    3778 non-null float64
review_scores_location          3779 non-null float64
review_scores_value             3778 non-null float64
requires_license               5835 non-null object
license                        1 non-null float64
jurisdiction_names             0 non-null float64
instant_bookable               5835 non-null object
cancellation_policy            5835 non-null object
require_guest_profile_picture  5835 non-null object
require_guest_phone_verification 5835 non-null object
calculated_host_listings_count 5835 non-null int64
reviews_per_month              3827 non-null float64
dtypes: float64(20), int64(14), object(58)
memory usage: 4.1+ MB

```

It turns out that one record displays a very odd location and, for the sake of the illustration, we will remove it:

```

odd = lst.loc[lst.longitude>-80, ['longitude', 'latitude']]
odd

```

	longitude	latitude
5832	-5.093682	43.214991

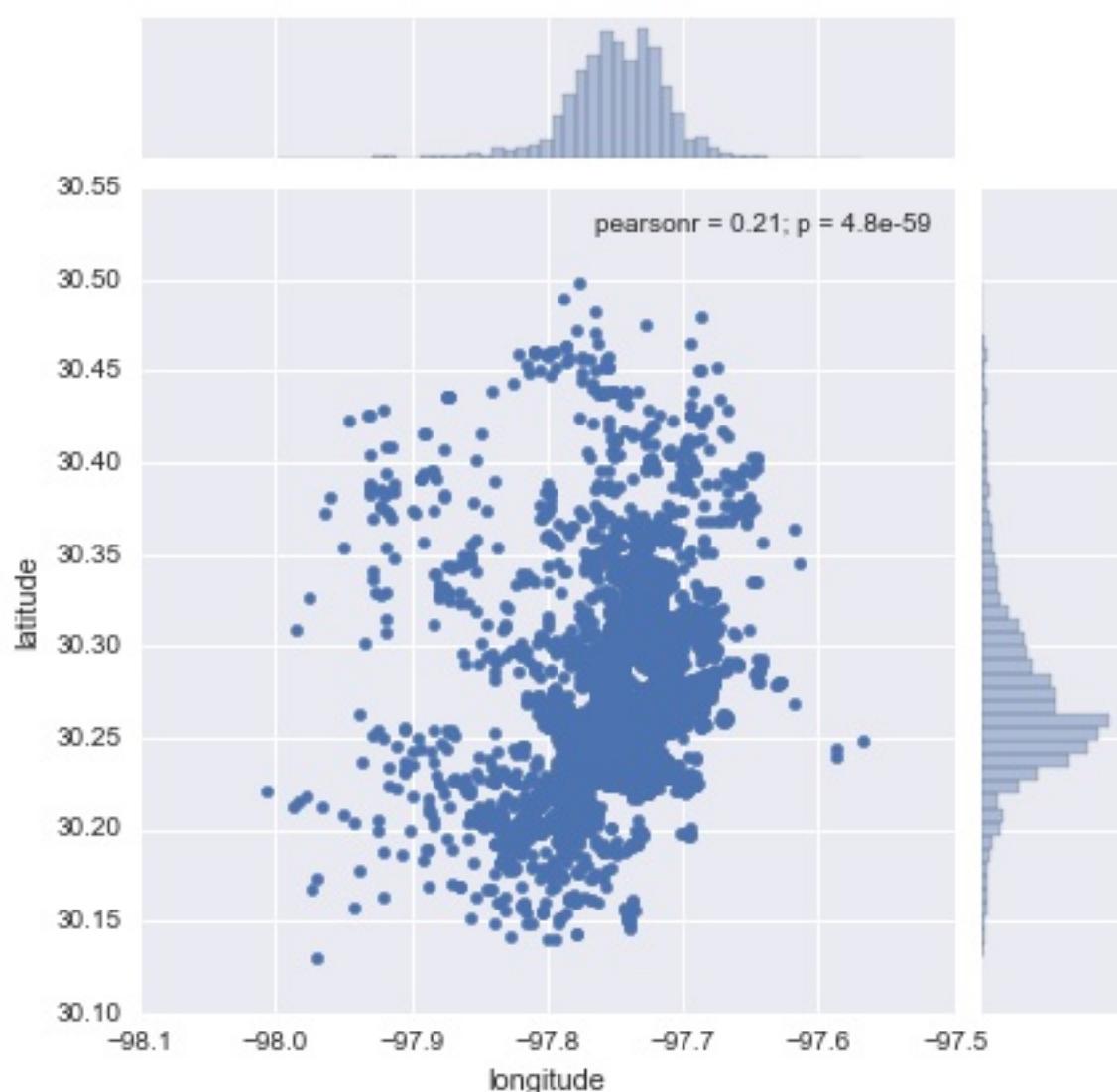
```
lst = lst.drop(odd.index)
```

Point Visualization

The most straightforward way to get a first glimpse of the distribution of the data is to plot their latitude and longitude:

```
sns.jointplot?
```

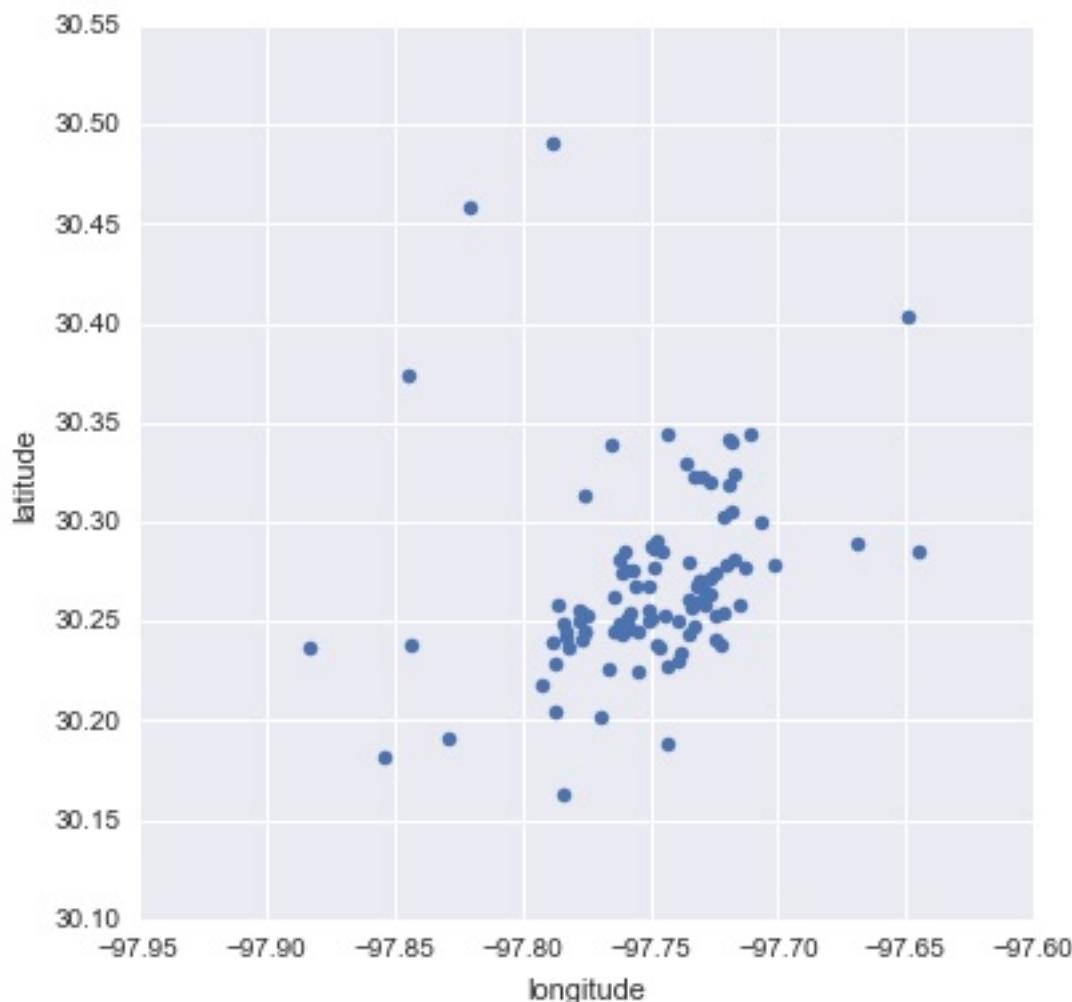
```
sns.jointplot(x="longitude", y="latitude", data=lst);
```



Now this does not necessarily tell us much about the dataset or the distribution of locations within Austin. There are two main challenges in interpreting the plot: one, there is lack of context, which means the points are not identifiable over space (unless you are so familiar with lon/lat pairs that they have a clear meaning to you); and two, in the center of the plot, there are so many points that it is hard to tell any pattern other than a big blurb of blue.

Let us first focus on the first problem, geographical context. The quickest and easiest way to provide context to this set of points is to overlay a general map. If we had an image with the map or a set of several data sources that we could aggregate to create a map, we could build it from scratch. But in the XXI Century, the easiest is to overlay our point dataset on top of a web map. In this case, we will use [Leaflet](#), and we will convert our underlying `matplotlib` points with `mplleaflet`. The full dataset (+5k observations) is a bit too much for leaflet to plot it directly on screen, so we will obtain a random sample of 100 points:

```
# NOTE: `mpld3.display` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering Leaflet map.
rids = np.arange(lst.shape[0])
np.random.shuffle(rids)
f, ax = plt.subplots(1, figsize=(6, 6))
lst.iloc[rids[:100], :].plot(kind='scatter', x='longitude', y='latitude',
                             s=30, linewidth=0, ax=ax);
#mpld3.display(fig=f, )
```



This map allows us to get a much better sense of where the points are and what type of location they might be in. For example, now we can see that the big blue blurb has to do with the urbanized core of Austin.

bokeh alternative

Leaflet is not the only technology to display data on maps, although it is probably the default option in many cases. When the data is larger than "acceptable", we need to resort to more technically sophisticated alternatives. One option is provided by `bokeh` and its `datashaded`

submodule (see [here](#) for a very nice introduction to the library, from where this example has been adapted).

Before we delve into `bokeh`, let us reproject our original data (lon/lat coordinates) into Web Mercator, as `bokeh` will expect them. To do that, we turn the coordinates into a `GeoSeries`:

```
from shapely.geometry import Point
xys_wb = gpd.GeoSeries(lst[['longitude', 'latitude']].apply(Point, axis=1), \
                       crs="+init=epsg:4326")
xys_wb = xys_wb.to_crs(epsg=3857)
x_wb = xys_wb.apply(lambda i: i.x)
y_wb = xys_wb.apply(lambda i: i.y)
```

Now we are ready to setup the plot in `bokeh`:

```
from bokeh.plotting import figure, output_notebook, show
from bokeh.tile_providers import STAMEN_TERRAIN
output_notebook()

minx, miny, maxx, maxy = xys_wb.total_bounds
y_range = miny, maxy
x_range = minx, maxx

def base_plot(tools='pan,wheel_zoom,reset', plot_width=600, plot_height=400, **plot_args):
    p = figure(tools=tools, plot_width=plot_width, plot_height=plot_height,
               x_range=x_range, y_range=y_range, outline_line_color=None,
               min_border=0, min_border_left=0, min_border_right=0,
               min_border_top=0, min_border_bottom=0, **plot_args)

    p.axis.visible = False
    p.xgrid.grid_line_color = None
    p.ygrid.grid_line_color = None
    return p

options = dict(line_color=None, fill_color='#800080', size=4)
```

```
<div class="bk-banner">
    <a href="http://bokeh.pydata.org" target="_blank" class="bk-logo bk-logo-small bk-logo-notebook"></a>
    <span id="efa98bda-2ccf-4dbf-ae97-94033d60c79b">Loading BokehJS . . .</span>
</div>
```

And good to go for mapping!

```
# NOTE: `show` turned off to be able to compile the website,  
#       comment out the last line of this cell for rendering.  
p = base_plot()  
p.add_tile(STAMEN_TERRAIN)  
p.circle(x=x_wb, y=y_wb, **options)  
#show(p)
```

```
<bokeh.models.renderers.GlyphRenderer at 0x1052bb5f8>
```

As you can quickly see, `bokeh` is substantially faster at rendering larger amounts of data.

The second problem we have spotted with the first scatter is that, when the number of points grows, at some point it becomes impossible to discern anything other than a big blur of color. To some extent, interactivity gets at that problem by allowing the user to zoom in until every point is an entity on its own. However, there exist techniques that allow to summarize the data to be able to capture the overall pattern at once. Traditionally, kernel density estimation (KDE) has been one of the most common solutions by approximating a continuous surface of point intensity. In this context, however, we will explore a more recent alternative suggested by the `datashader` library (see the [paper](#) if interested in more details).

Arguably, our dataset is not large enough to justify the use of a reduction technique like `datashader`, but we will create the plot for the sake of the illustration. Keep in mind, the usefulness of this approach increases the more points you need to be plotting.

```

# NOTE: `show` turned off to be able to compile the website,
#       comment out the last line of this cell for rendering.

import datashader as ds
from datashader.callbacks import InteractiveImage
from datashader.colors import viridis
from datashader import transfer_functions as tf
from bokeh.tile_providers import STAMEN_TONER

p = base_plot()
p.add_tile(STAMEN_TONER)

pts = pd.DataFrame({'x': x_wb, 'y': y_wb})
pts['count'] = 1
def create_image90(x_range, y_range, w, h):
    cvs = ds.Canvas(plot_width=w, plot_height=h, x_range=x_range, y_range=y_range)
    agg = cvs.points(pts, 'x', 'y', ds.count('count'))
    img = tf.interpolate(agg.where(agg > np.percentile(agg, 90)), \
                         cmap=viridis, how='eq_hist')
    return tf.dynspread(img, threshold=0.1, max_px=4)

#InteractiveImage(p, create_image90)

```

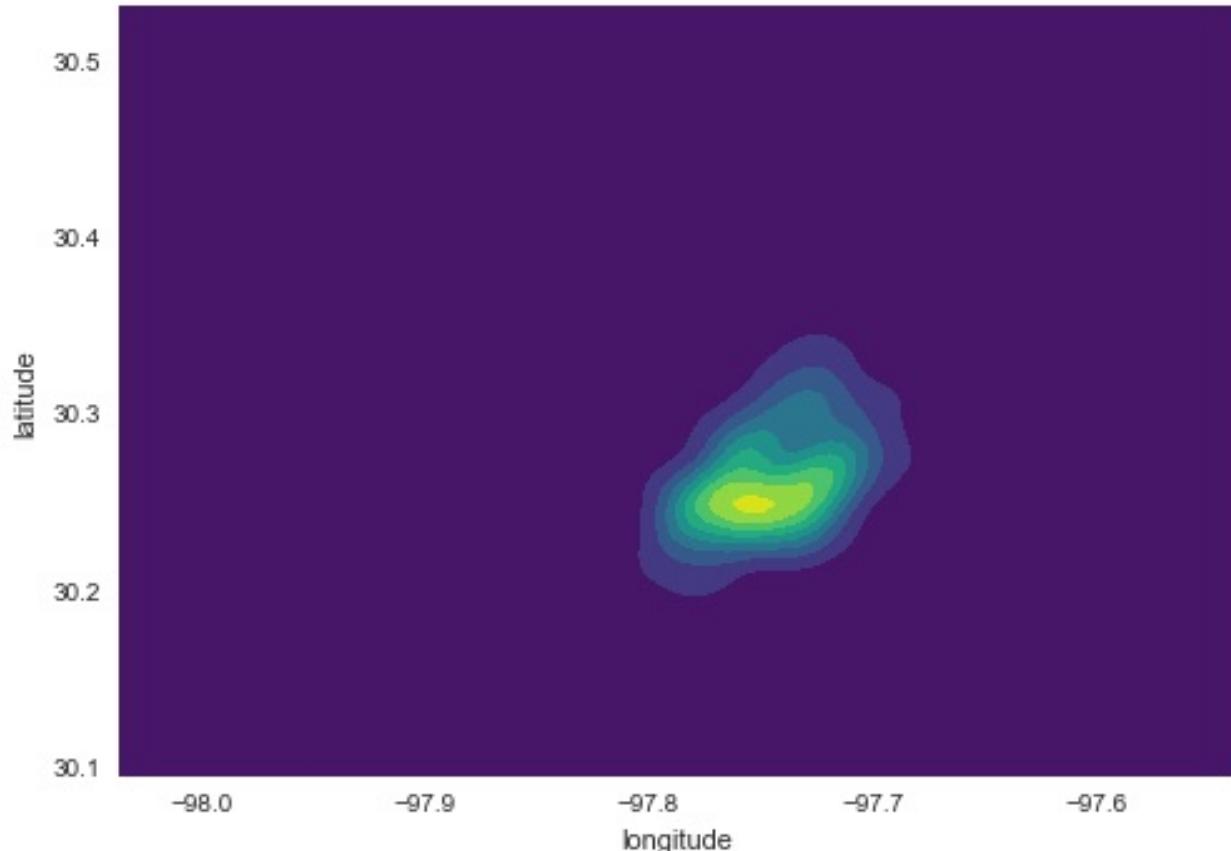
The key advantage of `datashader` is that it decouples the point processing from the plotting. That is the bit that allows it to be scalable to truly large datasets (e.g. millions of points). Essentially, the approach is based on generating a very fine grid, counting points within pixels, and encoding the count into a color scheme. In our map, this is not particularly effective because we do not have too many points (the previous plot is probably a more effective one) and essentially there is a pixel per location of every point. However, hopefully this example shows how to create this kind of scalable maps.

Kernel Density Estimation

A common alternative when the number of points grows is to replace plotting every single point by estimating the continuous observed probability distribution. In this case, we will not be visualizing the points themselves, but an abstracted surface that models the probability of point density over space. The most commonly used method to do this is the so called kernel density estimate (KDE). The idea behind KDEs is to count the number of points in a continuous way. Instead of using discrete counting, where you include a point in the count if it is inside a certain boundary and ignore it otherwise, KDEs use functions (kernels) that include points but give different weights to each one depending of how far of the location where we are counting the point is.

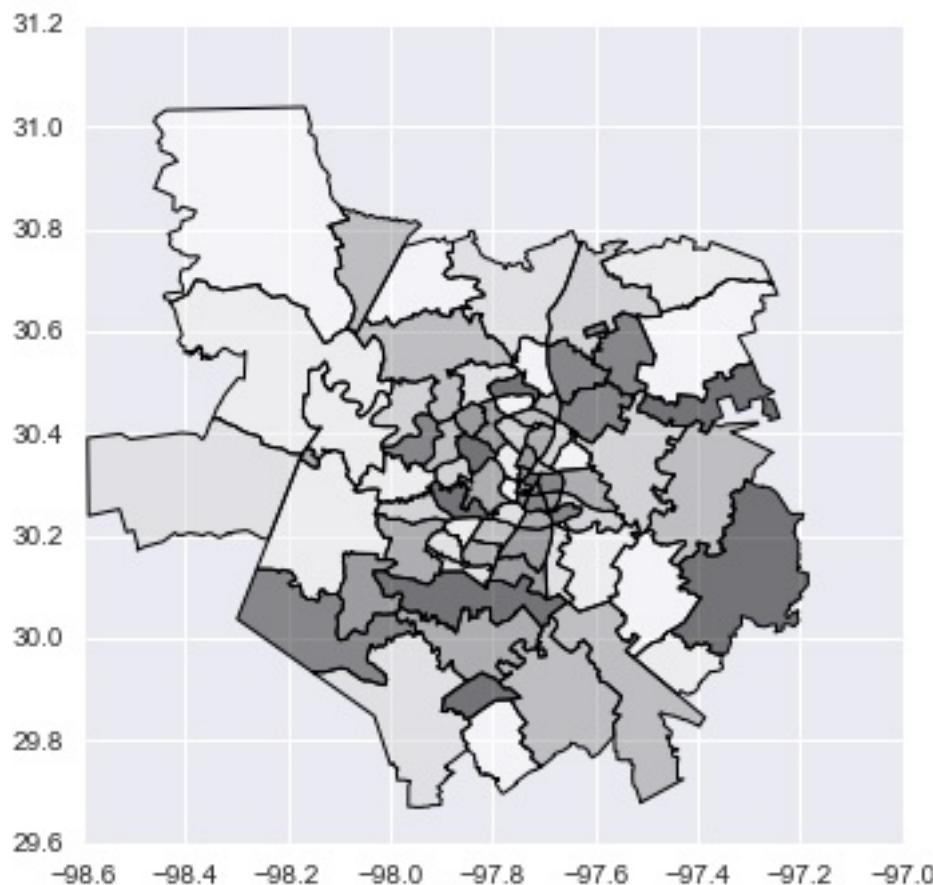
Creating a KDE is very straightforward in Python. In its simplest form, we can run the following single line of code:

```
sns.kdeplot(lst['longitude'], lst['latitude'], shade=True, cmap='viridis');
```



Now, if we want to include additional layers of data to provide context, we can do so in the same way we would layer up different elements in `matplotlib`. Let us load first the Zip codes in Austin, for example:

```
zc = gpd.read_file('../data/Zipcodes.geojson')
zc.plot();
```



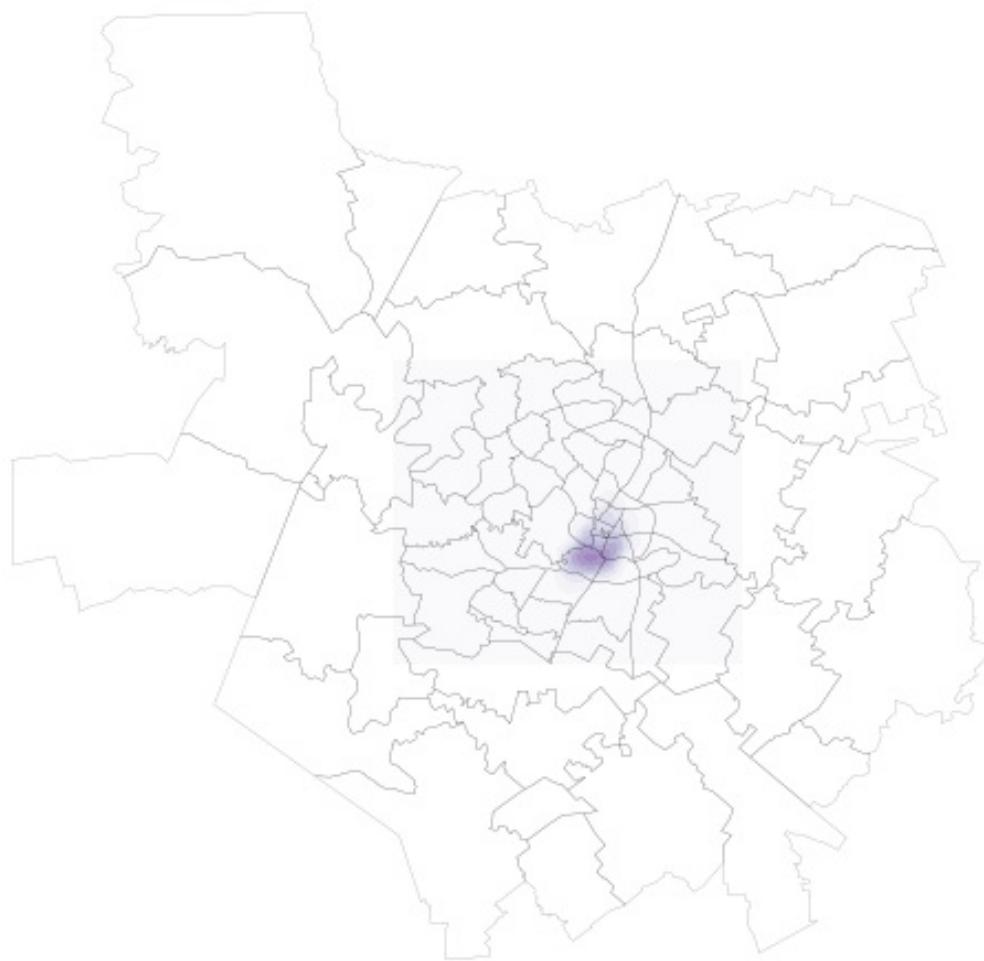
And, to overlay both layers:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='white', linewidth=0.1, ax=ax)

sns.kdeplot(lst['longitude'], lst['latitude'], \
            shade=True, cmap='Purples', \
            ax=ax);

ax.set_axis_off()
plt.axis('equal')
plt.show()
```



Exercise

Split the dataset by type of property and create a map for the five most common types.

Consider the following sorting of property types:

```
lst.property_type.groupby(lst.property_type)\n    .count()\n    .sort_values(ascending=False)
```

```
property_type
House           3549
Apartment        1855
Condominium      106
Loft             83
Townhouse        57
Other            47
Bed & Breakfast 37
Camper/RV         34
Bungalow          18
Cabin             17
Tent              11
Villa              7
Treehouse          7
Earth House        2
Chalet             1
Hut                1
Boat                1
Tipi                1
Name: property_type, dtype: int64
```

Spatial Clustering

IPYNB

NOTE: much of this material has been ported and adapted from "Lab 8" in [Arribas-Bel \(2016\)](#).

This notebook covers a brief introduction to spatial regression. To demonstrate this, we will use a dataset of all the AirBnb listings in the city of Austin (check the Data section for more information about the dataset).

Many questions and topics are complex phenomena that involve several dimensions and are hard to summarize into a single variable. In statistical terms, we call this family of problems *multivariate*, as opposed to *univariate* cases where only a single variable is considered in the analysis. Clustering tackles this kind of questions by reducing their dimensionality -the number of relevant variables the analyst needs to look at- and converting it into a more intuitive set of classes that even non-technical audiences can look at and make sense of. For this reason, it is widely used in applied contexts such as policymaking or marketing. In addition, since these methods do not require many preliminary assumptions about the structure of the data, it is a commonly used exploratory tool, as it can quickly give clues about the shape, form and content of a dataset.

The core idea of statistical clustering is to summarize the information contained in several variables by creating a relatively small number of categories. Each observation in the dataset is then assigned to one, and only one, category depending on its values for the variables originally considered in the classification. If done correctly, the exercise reduces the complexity of a multi-dimensional problem while retaining all the meaningful information contained in the original dataset. This is because, once classified, the analyst only needs to look at in which category every observation falls into, instead of considering the multiple values associated with each of the variables and trying to figure out how to put them together in a coherent sense. When the clustering is performed on observations that represent areas, the technique is often called geodemographic analysis.

The basic premise of the exercises we will be doing in this notebook is that, through the characteristics of the houses listed in AirBnb, we can learn about the geography of Austin. In particular, we will try to classify the city's zipcodes into a small number of groups that will allow us to extract some patterns about the main kinds of houses and areas in the city.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pysal as ps
import geopandas as gpd
from sklearn import cluster
from sklearn.preprocessing import scale

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
zc_link = '../data/Zipcodes.geojson'
```

Before anything, let us load the main dataset:

```
lst = pd.read_csv(abb_link)
```

Originally, this is provided at the individual level. Since we will be working in terms of neighborhoods and areas, we will need to aggregate them to that level. For this illustration, we will be using the following subset of variables:

```
varis = ['bedrooms', 'bathrooms', 'beds']
```

This will allow us to capture the main elements that describe the "look and feel" of a property and, by aggregation, of an area or neighborhood. All of the variables above are numerical values, so a sensible way to aggregate them is by obtaining the average (of bedrooms, etc.) per zipcode.

```
aves = lst.groupby('zipcode')[varis].mean()
aves.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 3 columns):
bedrooms      47 non-null float64
bathrooms     47 non-null float64
beds          47 non-null float64
dtypes: float64(3)
memory usage: 1.5 KB
```

In addition to these variables, it would be good to include also a sense of what proportions of different types of houses each zipcode has. For example, one can imagine that neighborhoods with a higher proportion of condos than single-family homes will probably look and feel more urban. To do this, we need to do some data munging:

```
types = pd.get_dummies(lst['property_type'])
prop_types = types.join(lst['zipcode'])\
    .groupby('zipcode')\
    .sum()
prop_types_pct = (prop_types * 100.).div(prop_types.sum(axis=1), axis=0)
prop_types_pct.info()
```

```
<class 'pandas.core.frame.DataFrame'>
Float64Index: 47 entries, 33558.0 to 78759.0
Data columns (total 18 columns):
Apartment        47 non-null float64
Bed & Breakfast 47 non-null float64
Boat             47 non-null float64
Bungalow         47 non-null float64
Cabin            47 non-null float64
Camper/RV         47 non-null float64
Chalet           47 non-null float64
Condominium       47 non-null float64
Earth House       47 non-null float64
House             47 non-null float64
Hut               47 non-null float64
Loft              47 non-null float64
Other             47 non-null float64
Tent              47 non-null float64
Tipi              47 non-null float64
Townhouse         47 non-null float64
Treehouse         47 non-null float64
Villa             47 non-null float64
dtypes: float64(18)
memory usage: 7.0 KB
```

Now we bring both sets of variables together:

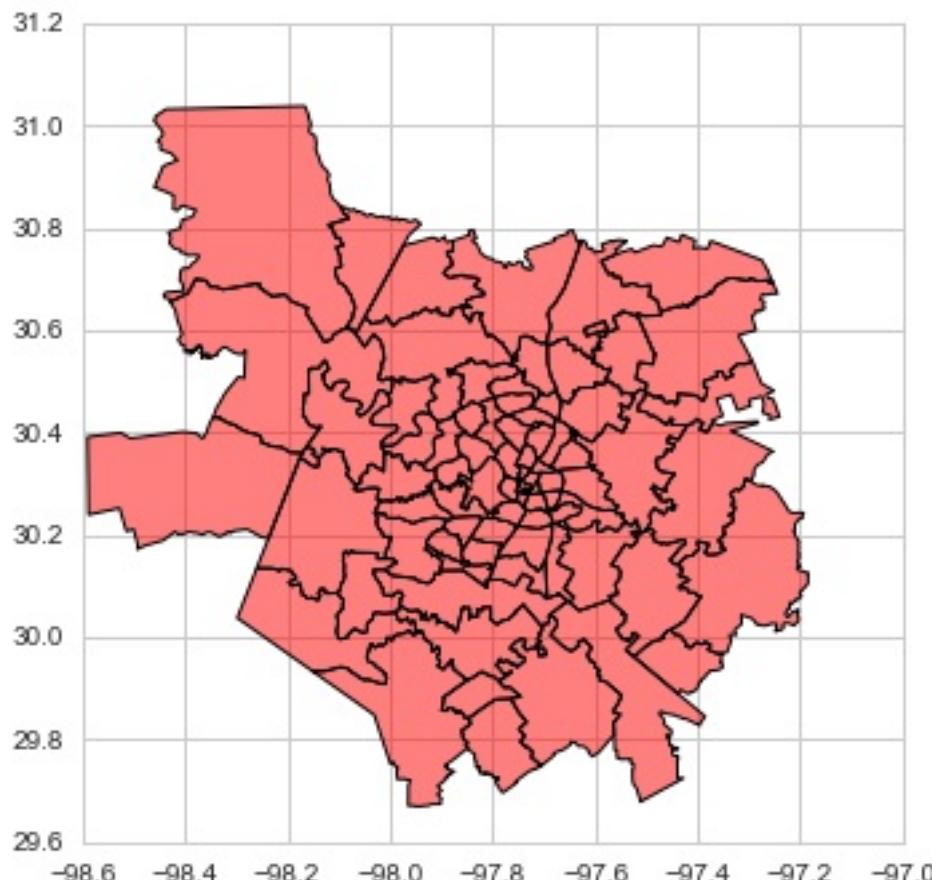
```
aves_props = aves.join(prop_types_pct)
```

And since we will be feeding this into the clustering algorithm, we will first standardize the columns:

```
db = pd.DataFrame(\n    scale(aves_props), \n    index=aves_props.index, \n    columns=aves_props.columns)\\n    .rename(lambda x: str(int(x)))
```

Now let us bring geography in:

```
zc = gpd.read_file(zc_link)\nzc.plot(color='red');
```



And combine the two:

```
zdb = zc[['geometry', 'zipcode', 'name']].join(db, on='zipcode')\\n    .dropna()
```

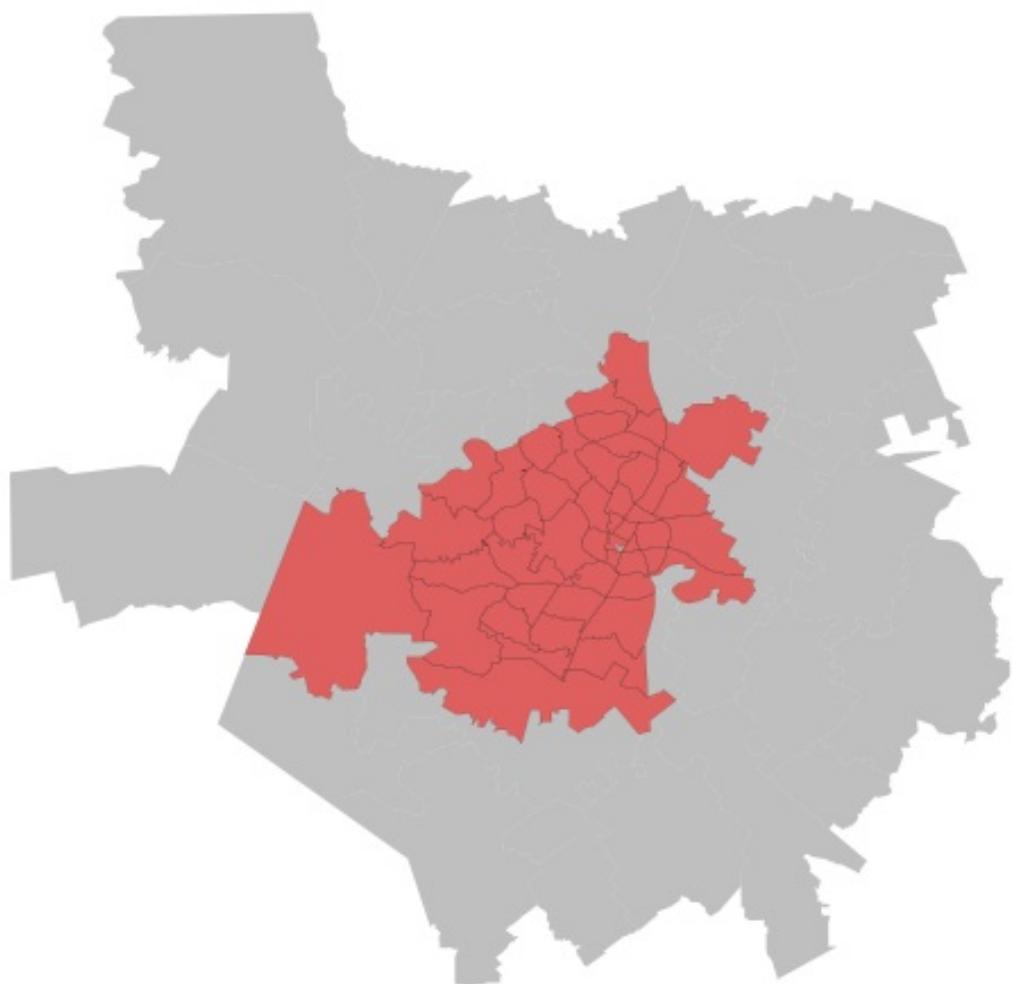
To get a sense of which areas we have lost:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zc.plot(color='grey', linewidth=0, ax=ax)
zdb.plot(color='red', linewidth=0.1, ax=ax)

ax.set_axis_off()

plt.show()
```



Geodemographic analysis

The main intuition behind geodemographic analysis is to group disparate areas of a city or region into a small set of classes that capture several characteristics shared by those in the same group. By doing this, we can get a new perspective not only on the types of areas in a city, but on how they are distributed over space. In the context of our AirBnb data analysis, the idea is that we can group different zipcodes of Austin based on the type of houses listed on the website. This will give us a hint into the geography of AirBnb in the Texan tech capital.

Although there exist many techniques to statistically group observations in a dataset, all of them are based on the premise of using a set of attributes to define classes or categories of observations that are similar *within* each of them, but differ *between* groups. How similarity within groups and dissimilarity between them is defined and how the classification algorithm is operationalized is what makes techniques differ and also what makes each of them particularly well suited for specific problems or types of data. As an illustration, we will only dip our toes into one of these methods, K-means, which is probably the most commonly used technique for statistical clustering.

Technically speaking, we describe the method and the parameters on the following line of code, where we specifically ask for five groups:

```
cluster.KMeans?
```

```
km5 = cluster.KMeans(n_clusters=5)
```

Following the `sklearn` pipeline approach, all the heavy-lifting of the clustering happens when we `fit` the model to the data:

```
zdb
```

	geometry	zipcode	name	bedrooms	bath
0	(POLYGON ((-97.63657664958002 30.4026577780347...))	78754	AUSTIN	-0.558936	-0.51
2	(POLYGON ((-97.89001573769271 30.2094119733396...))	78739	AUSTIN	-0.852262	-1.02
4	(POLYGON ((-97.87302802996035 30.4380637200422...))	78732	AUSTIN	2.411830	2.28

Spatial clustering

5	(POLYGON ((-98.01875589320069 30.2416650359924...	78737	AUSTIN	0.491860	0.34
6	(POLYGON ((-97.72792246181841 30.3290554493485...	78756	AUSTIN	0.172221	-0.1
7	(POLYGON ((-97.66539631711731 30.2856390650983...	78723	AUSTIN	0.809783	0.57
12	(POLYGON ((-97.84294910551613 30.3230986344075...	78733	AUSTIN	0.646410	0.70
14	(POLYGON ((-97.77663063721972 30.3230061422016...	78746	AUSTIN	0.709989	1.03
15	(POLYGON ((-97.79105326806381 30.4504847141011...	78750	AUSTIN	0.339864	1.02
17	(POLYGON ((-97.86241464122685 30.2970038099151...	78735	AUSTIN	1.165182	1.52
24	(POLYGON ((-97.736172946862 30.15986516866584,...	78747	AUSTIN	-0.765989	0.38
26	(POLYGON ((-97.70676595120581 30.3221053986124...	78751	AUSTIN	-0.243801	-0.4
28	(POLYGON ((-98.00621936585073 30.3607912172675...	78738	AUSTIN	0.786911	0.41
30	(POLYGON ((-97.6233816077967 30.26459422030998...	78725	AUSTIN	0.786911	0.23
31	(POLYGON ((-97.67105719588658 30.4713206458632...	78728	AUSTIN	-1.180096	-1.2
32	(POLYGON ((-97.70652535557282 30.2502416627252...	78741	AUSTIN	-0.196593	-0.2

Spatial clustering

33	(POLYGON ((-97.68335342189505 30.2143231792978...)	78744	AUSTIN	0.280258	0.06
36	(POLYGON ((-97.76605641344037 30.3129541271378...)	78703	AUSTIN	0.520842	0.43
37	(POLYGON ((-97.82331016792597 30.2348601368125...)	78749	AUSTIN	0.281109	0.41
39	(POLYGON ((-97.9221254299873 30.2930035379774,...)	78736	AUSTIN	0.505910	0.33
41	(POLYGON ((-97.70394537118878 30.2825715190553...)	78722	AUSTIN	-0.497665	-0.80
42	(POLYGON ((-97.66539631711731 30.2856390650983...)	78721	AUSTIN	0.529327	-0.01
43	(POLYGON ((-97.69186077161216 30.2484769354393...)	78702	AUSTIN	0.555498	0.22
44	(POLYGON ((-97.84397384513375 30.3935729185546...)	78730	AUSTIN	2.004582	2.71
45	(POLYGON ((-97.73177659202942 30.5876874089334...)	78681	ROUND ROCK	-1.180096	-1.24
46	(POLYGON ((-97.74597901667516 30.3613774432803...)	78731	AUSTIN	1.207048	1.13
47	(POLYGON ((-97.90409880079483 30.1645969721025...)	78652	MANCHACA	-1.180096	-1.24
49	(POLYGON ((-97.79207307087505 30.2318294401264...)	78745	AUSTIN	0.311501	-0.14
50	(POLYGON ((-97.83150003630331 30.1899955351445...)	78748	AUSTIN	-0.305871	-0.41
	(POLYGON				

Spatial clustering

51	((-97.73600098711186 30.3785689883760...	78757	AUSTIN	0.313372	0.09
53	(POLYGON ((-97.70739686222652 30.4748248576790...	78717	AUSTIN	-1.180096	-0.76
54	(POLYGON ((-97.72137830361672 30.4735808118949...	78729	AUSTIN	-0.618094	-0.66
55	(POLYGON ((-98.12139104830618 30.3647789869612...	78620	DRIPPING SPRINGS	-1.180096	-1.24
56	(POLYGON ((-97.7544731627255 30.26506651013847...	78704	AUSTIN	0.353945	0.04
57	(POLYGON ((-97.68634026954943 30.3325933205724...	78752	AUSTIN	-0.161881	-0.76
59	(POLYGON ((-97.56072444145862 30.2490283208835...	78724	AUSTIN	0.786911	1.15
61	(POLYGON ((-97.92086446378035 30.4349563110724...	78734	AUSTIN	2.332417	2.92
62	(POLYGON ((-97.85677348291654 30.1304340182730...	78610	BUDA	-1.180096	-1.24
66	(POLYGON ((-97.67150729716587 30.4243750876293...	78753	AUSTIN	-0.280893	-0.14
69	(POLYGON ((-97.8258020623902 30.45545039884336...	78726	AUSTIN	-0.056092	-0.66
70	(POLYGON ((-97.57652979651098 30.5008564155205...	78660	PFLUGERVILLE	-1.180096	-1.24
74	(POLYGON ((-97.72977818013679 30.2987072033635...	78705	AUSTIN	-0.561441	-0.57
75	(POLYGON ((-97.747440320186	78759	AUSTIN	-0.726172	-0.56

	30.42161819776979,...					
76	(POLYGON ((-97.69729138340391 30.4400371609576...)	78727	TRAVIS	-0.501818	-0.51	
77	(POLYGON ((-97.71223026174552 30.3480047024702...)	78758	AUSTIN	-0.406192	-0.41	
78	(POLYGON ((-97.73598800490392 30.2510129132375...)	78701	AUSTIN	-0.162204	-0.16	

46 rows × 24 columns

```
km5cls = km5.fit(zdb.drop(['geometry', 'name'], axis=1).values)
```

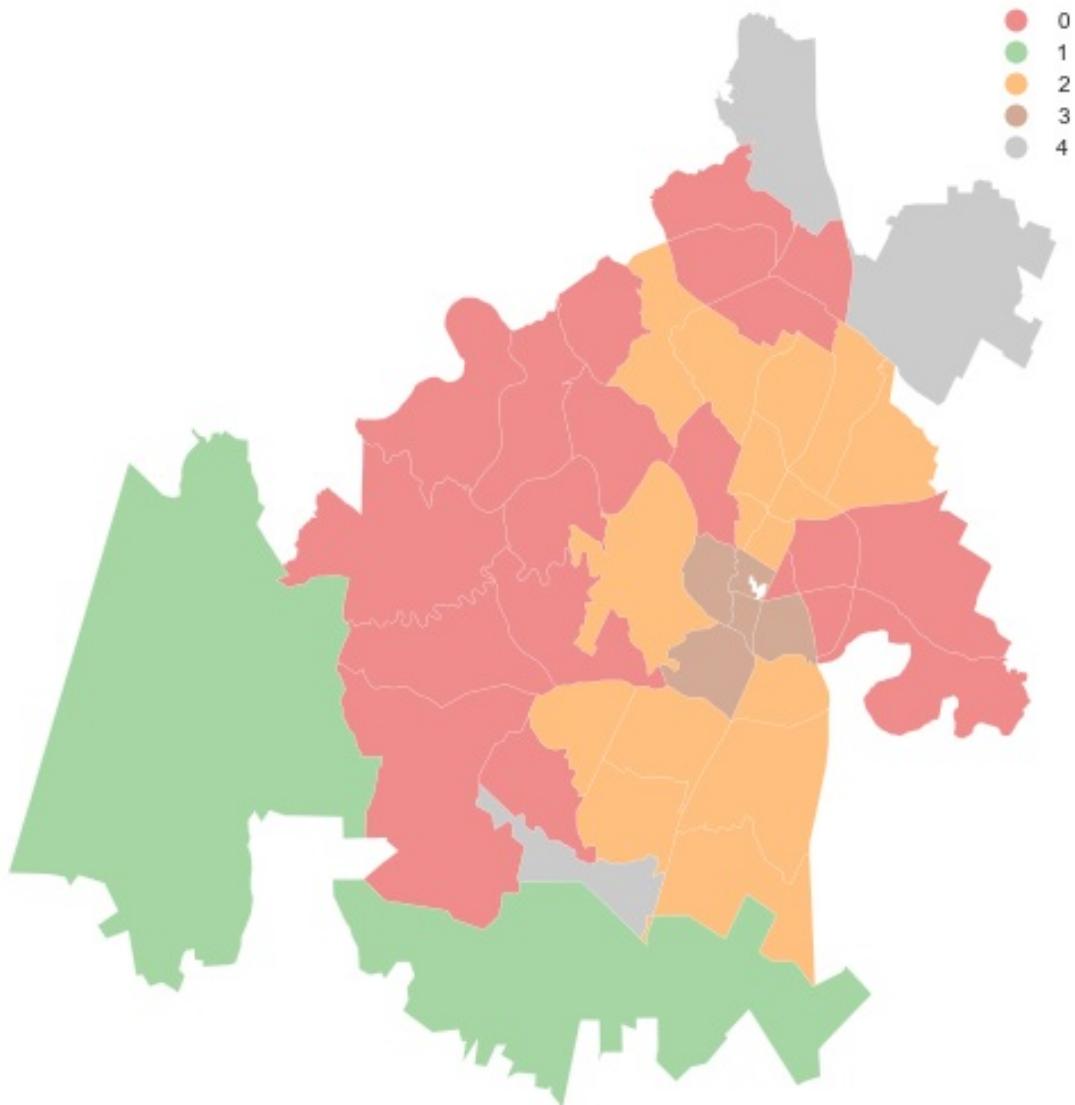
Now we can extract the classes and put them on a map:

```
f, ax = plt.subplots(1, figsize=(9, 9))

zdb.assign(cl=km5cls.labels_)\n    .plot(column='cl', categorical=True, legend=True,\n          linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```

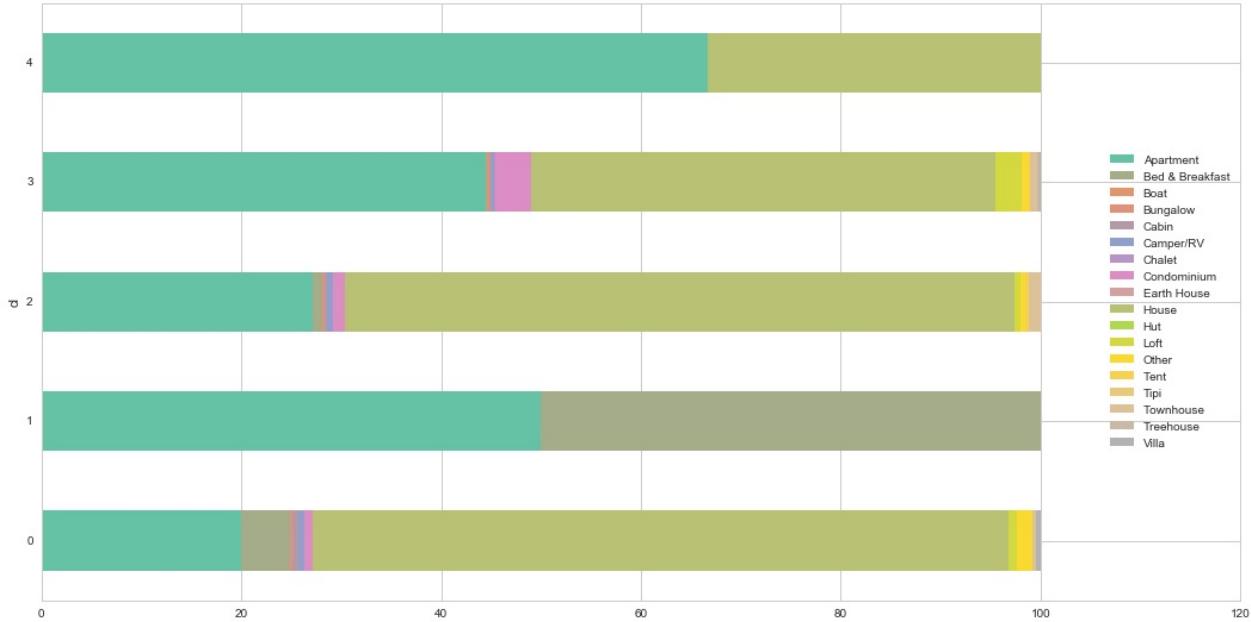


The map above shows a clear pattern: there is a class at the core of the city (number 0, in red), then two other ones in a sort of "urban ring" (number 1 and 3, in green and brown, respectively), and two peripheral sets of areas (number 2 and 4, yellow and green).

This gives us a good insight into the geographical structure, but does not tell us much about what are the defining elements of these groups. To do that, we can have a peek into the characteristics of the classes. For example, let us look at how the proportion of different types of properties are distributed across clusters:

```
c1_pcts = prop_types_pct.rename(lambda x: str(int(x)))\
    .reindex(zdb['zipcode'])\
    .assign(cl=km5cls.labels_)\
    .groupby('cl')\
    .mean()
```

```
f, ax = plt.subplots(1, figsize=(18, 9))
cl_pcts.plot(kind='barh', stacked=True, ax=ax, \
              cmap='Set2', linewidth=0)
ax.legend(ncol=1, loc="right");
```



A few interesting, albeit maybe not completely unsurprising, characteristics stand out. First, most of the locations we have in the dataset are either apartments or houses. However, how they are distributed is interesting. The urban core -cluster 0- distinctively has the highest proportion of condos and lofts. The suburban ring -clusters 1 and 3- is very consistent, with a large share of houses and less apartments, particularly so in the case of cluster 3. Class 4 has only two types of properties, houses and apartments, suggesting there are not that many places listed at Airbnb. Finally, class 3 arises as a more rural and leisure one: beyond apartments, it has a large share of bed & breakfasts.

Mini Exercise

What are the average number of beds, bedrooms and bathrooms for every class?

Regionalization analysis: building (meaningful) regions

In the case of analysing spatial data, there is a subset of methods that are of particular interest for many common cases in Geographic Data Science. These are the so-called regionalization techniques. Regionalization methods can take also many forms and faces but, at their core, they all involve statistical clustering of observations with the additional constraint that observations

need to be geographical neighbors to be in the same category. Because of this, rather than category, we will use the term area for each observation and region for each class or cluster - hence regionalization, the construction of regions from smaller areas.

As in the non-spatial case, there are many different algorithms to perform regionalization, and they all differ on details relating to the way they measure (dis)similarity, the process to regionalize, etc. However, same as above too, they all share a few common aspects. In particular, they all take a set of input attributes *and* a representation of space in the form of a binary spatial weights matrix. Depending on the algorithm, they also require the desired number of output regions into which the areas are aggregated.

In this example, we are going to create aggregations of zipcodes into groups that have areas where the Airbnb listed location have similar ratings. In other words, we will create delineations for the "quality" or "satisfaction" of Airbnb users. In other words, we will explore what are the boundaries that separate areas where Airbnb users tend to be satisfied about their experience versus those where the ratings are not as high. To do this, we will focus on the `review_scores_x` set of variables in the original dataset:

```
ratings = [i for i in lst if 'review_scores_' in i]
ratings
```

```
['review_scores_rating',
 'review_scores_accuracy',
 'review_scores_cleanliness',
 'review_scores_checkin',
 'review_scores_communication',
 'review_scores_location',
 'review_scores_value']
```

Similarly to the case above, we now bring this at the zipcode level. Note that, since they are all scores that range from 0 to 100, we can use averages and we do not need to standardize.

```
rt_av = lst.groupby('zipcode')[ratings]\n    .mean()\n    .rename(lambda x: str(int(x)))
```

And we link these to the geometries of zipcodes:

```
zrt = zc[['geometry', 'zipcode']].join(rt_av, on='zipcode')\n        .dropna()\n\nzrt.info()
```

```
<class 'geopandas.geodataframe.GeoDataFrame'>
Int64Index: 43 entries, 0 to 78
Data columns (total 9 columns):
geometry              43 non-null object
zipcode               43 non-null object
review_scores_rating   43 non-null float64
review_scores_accuracy 43 non-null float64
review_scores_cleanliness 43 non-null float64
review_scores_checkin   43 non-null float64
review_scores_communication 43 non-null float64
review_scores_location   43 non-null float64
review_scores_value      43 non-null float64
dtypes: float64(7), object(2)
memory usage: 3.4+ KB
```

In contrast to the standard clustering techniques, regionalization requires a formal representation of topology. This is so the algorithm can impose spatial constraints during the process of clustering the observations. We will use exactly the same approach as in the previous sections of this tutorial for this and build spatial weights objects `w` with `Pysal`. For the sake of this illustration, we will consider queen contiguity, but any other rule should work fine as long as there is a rational behind it. Weights constructors currently only work from shapefiles on disk, so we will write our `GeoDataFrame` first, then create the `w` object, and remove the files.

```
zrt.to_file('tmp')
w = ps.queen_from_shapefile('tmp/tmp.shp', idVariable='zipcode')
# NOTE: this might not work on Windows
! rm -r tmp
w
```

```
<pysal.weights.weights.W at 0x11bd5ff98>
```

Now we are ready to run the regionalization algorithm. In this case we will use the `max-p` ([Duque, Anselin & Rey, 2012](#)), which does not require a predefined number of output regions but instead it takes a target variable that you want to make sure a minimum threshold is met. In our case, since it is based on ratings, we will impose that every resulting region has at least 10% of the total number of reviews. Let us work through what that would mean:

```
n_rev = lst.groupby('zipcode')\
    .sum()\
    ['number_of_reviews']\
    .rename(lambda x: str(int(x)))\
    .reindex(zrt['zipcode'])
thr = np.round(0.1 * n_rev.sum())
thr
```

6271.0

This means we want every resulting region to be based on at least 6,271 reviews. Now we have all the pieces, let us glue them together through the algorithm:

```
# Set the seed for reproducibility
np.random.seed(1234)

z = zrt.drop(['geometry', 'zipcode'], axis=1).values
maxp = ps.region.Maxp(w, z, thr, n_rev.values[:, None], initial=1000)
```

We can check whether the solution is better (lower within sum of squares) than we would have gotten from a purely random regionalization process using the `c inference` method:

```
%%time
np.random.seed(1234)
maxp.c inference(nperm=999)
```

```
CPU times: user 26.2 s, sys: 185 ms, total: 26.4 s
Wall time: 32.1 s
```

Which allows us to obtain an empirical p-value:

```
maxp.cpvalue
```

0.022

Which gives us reasonably good confidence that the solution we obtain is more meaningful than pure chance.

With that out of the way, let us see what the result looks like on a map! First we extract the labels:

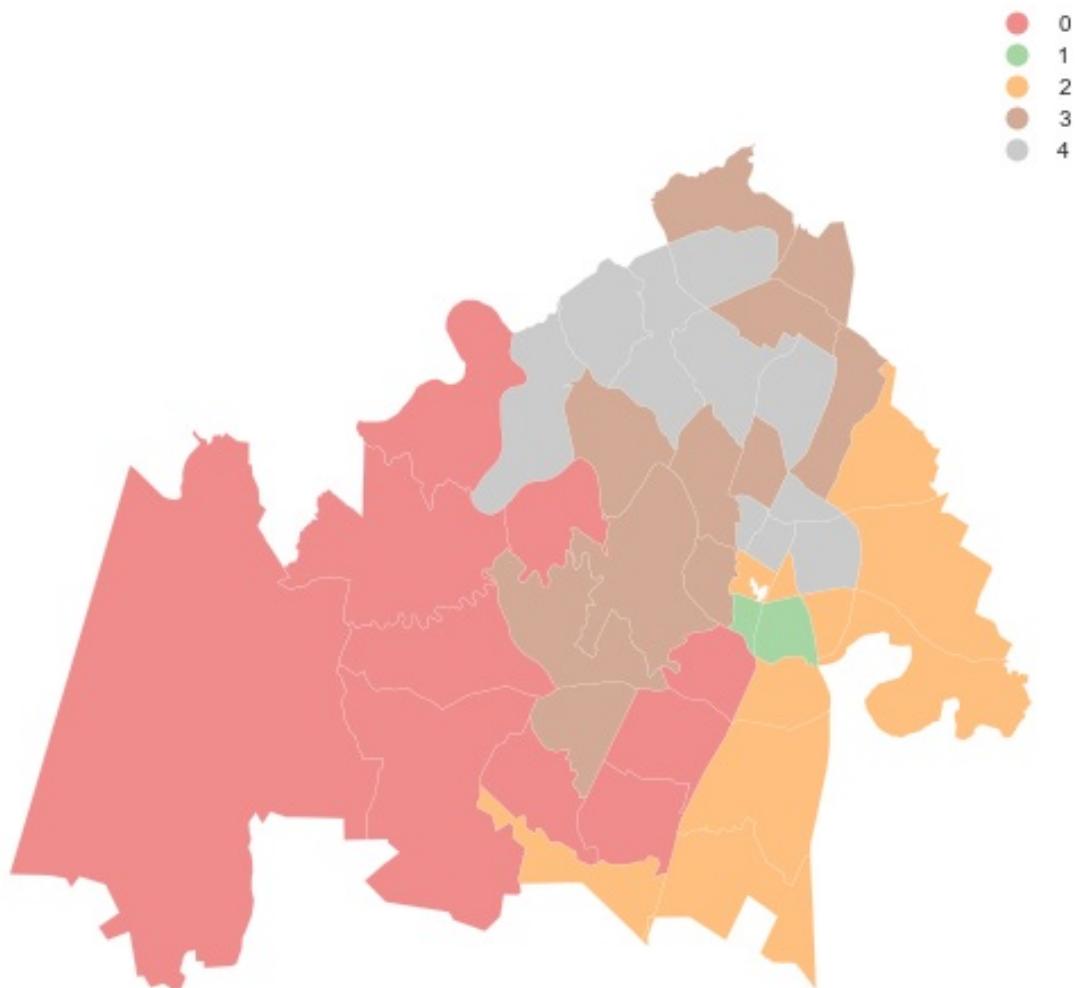
```
lbls = pd.Series(maxp.area2region).reindex(zrt['zipcode'])
```

```
f, ax = plt.subplots(1, figsize=(9, 9))

zrt.assign(cl=lbls.values)\n    .plot(column='cl', categorical=True, legend=True,\n          linewidth=0.1, edgecolor='white', ax=ax)

ax.set_axis_off()

plt.show()
```



The map shows a clear geographical pattern with a western area, another in the North and a smaller one in the East. Let us unpack what each of them is made of:

```
zrt[ratings].groupby(lbls.values).mean().T
```

	0	1	2	
review_scores_rating	96.911817	95.326614	92.502135	96.1747
review_scores_accuracy	9.767500	9.605032	9.548751	9.60745
review_scores_cleanliness	9.678277	9.558179	8.985408	9.59982
review_scores_checkin	9.922450	9.797086	9.765563	9.88992
review_scores_communication	9.932211	9.827390	9.794794	9.89878
review_scores_location	9.644754	9.548761	8.904775	9.59674
review_scores_value	9.678822	9.341224	9.491638	9.61418

Although very similar, there are some patterns to be extracted. For example, the East area seems to have lower overall scores.

Exercise

Obtain a geodemographic classification with eight classes instead of five and replicate the analysis above

Re-run the regionalization exercise imposing a minimum of 5% reviews per area

Spatial Regression

IPYNB

NOTE: some of this material has been ported and adapted from the Spatial Econometrics note in [Arribas-Bel \(2016b\)](#).

This notebook covers a brief and gentle introduction to spatial econometrics in Python. To do that, we will use a set of Austin properties listed in AirBnb.

The core idea of spatial econometrics is to introduce a formal representation of space into the statistical framework for regression. This can be done in many ways: by including predictors based on space (e.g. distance to relevant features), by splitting the datasets into subsets that map into different geographical regions (e.g. [spatial regimes](#)), by exploiting close distance to other observations to borrow information in the estimation (e.g. [kriging](#)), or by introducing variables that put in relation their value at a given location with those in nearby locations, to give a few examples. Some of these approaches can be implemented with standard non-spatial techniques, while others require bespoke models that can deal with the issues introduced. In this short tutorial, we will focus on the latter group. In particular, we will introduce some of the most commonly used methods in the field of spatial econometrics.

The example we will use to demonstrate this draws on hedonic house price modelling. This a well-established methodology that was developed by [Rosen \(1974\)](#) that is capable of recovering the marginal willingness to pay for goods or services that are not traded in the market. In other words, this allows us to put an implicit price on things such as living close to a park or in a neighborhood with good quality of air. In addition, since hedonic models are based on linear regression, the technique can also be used to obtain predictions of house prices.

Data

Before anything, let us load up the libraries we will use:

```
%matplotlib inline

import seaborn as sns
import matplotlib.pyplot as plt
import numpy as np
import pandas as pd
import pysal as ps
import geopandas as gpd

sns.set(style="whitegrid")
```

Let us also set the paths to all the files we will need throughout the tutorial, which is only the original table of listings:

```
# Adjust this to point to the right file in your computer
abb_link = '../data/listings.csv.gz'
```

And go ahead and load it up too:

```
lst = pd.read_csv(abb_link)
```

Baseline (nonspatial) regression

Before introducing explicitly spatial methods, we will run a simple linear regression model. This will allow us, on the one hand, set the main principles of hedonic modeling and how to interpret the coefficients, which is good because the spatial models will build on this; and, on the other hand, it will provide a baseline model that we can use to evaluate how meaningful the spatial extensions are.

Essentially, the core of a linear regression is to explain a given variable -the price of a listing P_i on AirBnb (P_i)- as a linear function of a set of other characteristics we will collectively call X_i :

$$\ln(P_i) = \alpha + \beta X_i + \epsilon_i$$

For several reasons, it is common practice to introduce the price in logarithms, so we will do so here. Additionally, since this is a probabilistic model, we add an error term ϵ_i that is assumed to be well-behaved (i.i.d. as a normal).

For our example, we will consider the following set of explanatory features of each listed property:

```
x = ['host_listings_count', 'bathrooms', 'bedrooms', 'beds', 'guests_included']
```

Additionally, we are going to derive a new feature of a listing from the `amenities` variable. Let us construct a variable that takes 1 if the listed property has a pool and 0 otherwise:

```
def has_pool(a):
    if 'Pool' in a:
        return 1
    else:
        return 0

lst['pool'] = lst['amenities'].apply(has_pool)
```

For convenience, we will re-package the variables:

```
yxs = lst.loc[:, x + ['pool', 'price']].dropna()
y = np.log(\n    yxs['price'].apply(lambda x: float(x.strip('$').replace(',', '')))\n    + 0.000001
)
```

To run the model, we can use the `spreg` module in `Pysal`, which implements a standard OLS routine, but is particularly well suited for regressions on spatial data. Also, although for the initial model we do not need it, let us build a spatial weights matrix that connects every observation to its 8 nearest neighbors. This will allow us to get extra diagnostics from the baseline model.

```
w = ps.knnw_from_array(lst.loc[\n    yxs.index, \n    ['longitude', 'latitude']]\\
].values)

w.transform = 'R'

w
```

```
<pysal.weights.weights.W at 0x11bdb5358>
```

At this point, we are ready to fit the regression:

```
m1 = ps.spreg.OLS(y.values[:, None], yxs.drop('price', axis=1).values, \
                    w=w, spat_diag=True, \
                    name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='ln(price)')
```

To get a quick glimpse of the results, we can print its summary:

```
print(m1.summary)
```

```
REGRESSION
-----
SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES
-----
Data set : unknown
Weights matrix : unknown
Dependent Variable : ln(price) Number of Observations: 57
67
Mean dependent var : 5.1952 Number of Variables :
7
S.D. dependent var : 0.9455 Degrees of Freedom : 57
60
R-squared : 0.4042
Adjusted R-squared : 0.4036
Sum squared residual: 3071.189 F-statistic : 651.39
58
Sigma-square : 0.533 Prob(F-statistic) :
0
S.E. of regression : 0.730 Log likelihood : -6366.1
62
Sigma-square ML : 0.533 Akaike info criterion : 12746.3
25
S.E of regression ML: 0.7298 Schwarz criterion : 12792.9
44
-----
--          Variable Coefficient Std.Error t-Statistic Probabilit
ty
--          CONSTANT 4.0976886 0.0223530 183.3171506 0.000000
00
host_listings_count -0.0000130 0.0001790 -0.0726772 0.94206
55
bathrooms 0.2947079 0.0194817 15.1273879 0.000000
00
bedrooms 0.3274226 0.0159666 20.5067654 0.000000
00
```

Spatial Regression

40	beds	0.0245741	0.0097379	2.5235601	0.01164
30	guests_included	0.0075119	0.0060551	1.2406028	0.21480
36	pool	0.0888039	0.0221903	4.0019209	0.00006

--					
REGRESSION DIAGNOSTICS					
MULTICOLLINEARITY CONDITION NUMBER				9.260	
TEST ON NORMALITY OF ERRORS					
TEST	DF	VALUE	PROB		
Jarque-Bera	2	1358479.047	0.0000		
DIAGNOSTICS FOR HETROSKEDEASTICITY					
RANDOM COEFFICIENTS					
TEST	DF	VALUE	PROB		
Breusch-Pagan test	6	1414.297	0.0000		
Koenker-Bassett test	6	36.756	0.0000		
DIAGNOSTICS FOR SPATIAL DEPENDENCE					
TEST	MI/DF	VALUE	PROB		
Lagrange Multiplier (lag)	1	255.796	0.0000		
Robust LM (lag)	1	13.039	0.0003		
Lagrange Multiplier (error)	1	278.752	0.0000		
Robust LM (error)	1	35.995	0.0000		
Lagrange Multiplier (SARMA)	2	291.791	0.0000		
===== END OF REPORT =====					
==					

Results are largely unsurprising, but nonetheless reassuring. Both an extra bedroom and an extra bathroom increase the final price around 30%. Accounting for those, an extra bed pushes the price about 2%. Neither the number of guests included nor the number of listings the host has in total have a significant effect on the final price.

Including a spatial weights object in the regression buys you an extra bit: the summary provides results on the diagnostics for spatial dependence. These are a series of statistics that test whether the residuals of the regression are spatially correlated, against the null of a random distribution over space. If the latter is rejected a key assumption of OLS, independently distributed error terms, is violated. Depending on the structure of the spatial pattern, different strategies have been defined within the spatial econometrics literature to deal with them. If you are interested in this, a very recent and good resource to check out is [Anselin & Rey \(2015\)](#). The main summary from the diagnostics for spatial dependence is that there is clear evidence to reject the null of spatial randomness in the residuals, hence an explicitly spatial approach is warranted.

Spatially lagged exogenous regressors (wX)

The first and most straightforward way to introduce space is by "spatially lagging" one of the explanatory variables. Mathematically, this can be expressed as follows:

$$\ln(P_i) = \alpha + \beta X_i + \delta \sum_j w_{ij} X'_i + \epsilon_i$$

where X'_i is a subset of X_i , although it could encompass all of the explanatory variables, and w_{ij} is the ij -th cell of a spatial weights matrix W . Because W assigns non-zero values only to spatial neighbors, if W is row-standardized (customary in this context), then $\sum_j w_{ij} X'_i$ captures the average value of X'_i in the surroundings of location i . This is what we call the *spatial lag* of X_i . Also, since it is a spatial transformation of an explanatory variable, the standard estimation approach -OLS- is sufficient: spatially lagging the variables does not violate any of the assumptions on which OLS relies.

Usually, we will want to spatially lag variables that we think may affect the price of a house in a given location. For example, one could think that pools represent a visual amenity. If that is the case, then listed properties surrounded by other properties with pools might, everything else equal, be more expensive. To calculate the number of pools surrounding each property, we can build an alternative weights matrix that we do not row-standardize:

```
w_pool = ps.knnW_from_array(lst.loc[\n    yxs.index, \n    ['longitude', 'latitude']]\\n    ].values)\n\nyxs_w = yxs.assign(w_pool=ps.lag_spatial(w_pool, yxs['pool'].values))
```

And now we can run the model, which has the same setup as `m1`, with the exception that it includes the number of AirBnb properties with pools surrounding each house:

```
m2 = ps.spreg.OLS(y.values[:, None], \n    yxs_w.drop('price', axis=1).values, \\n    w=w, spat_diag=True, \\n    name_x=yxs_w.drop('price', axis=1).columns.tolist(), name_y='ln(price)')\n\nprint(m2.summary)
```

REGRESSION

SUMMARY OF OUTPUT: ORDINARY LEAST SQUARES					
Data set	:	unknown			
Weights matrix	:	unknown			
Dependent Variable	:	ln(price)	Number of Observations:	57	
67					
Mean dependent var	:	5.1952	Number of Variables :		
8					
S.D. dependent var	:	0.9455	Degrees of Freedom :	57	
59					
R-squared	:	0.4044			
Adjusted R-squared	:	0.4037			
Sum squared residual:	3070.363		F-statistic :	558.61	
39					
Sigma-square	:	0.533	Prob(F-statistic) :		
0					
S.E. of regression	:	0.730	Log likelihood :	-6365.3	
87					
Sigma-square ML	:	0.532	Akaike info criterion :	12746.7	
73					
S.E. of regression ML:	0.7297		Schwarz criterion :	12800.0	
53					
<hr/>					
--	Variable	Coefficient	Std.Error	t-Statistic	Probability
ty					
--	CONSTANT	4.0906444	0.0230571	177.4134022	0.00000
00					
host_listings_count	-0.0000108	0.0001790	-0.0603617	0.95186	
97					
bathrooms	0.2948787	0.0194813	15.1365024	0.00000	
00					
bedrooms	0.3277450	0.0159679	20.5252404	0.00000	
00					
beds	0.0246650	0.0097377	2.5329419	0.01133	
73					
guests_included	0.0076894	0.0060564	1.2696250	0.20426	
95					
pool	0.0725756	0.0257356	2.8200486	0.00481	
81					
w_pool	0.0188875	0.0151729	1.2448141	0.21325	
08					
--					
REGRESSION DIAGNOSTICS					
MULTICOLLINEARITY CONDITION NUMBER			9.605		

TEST ON NORMALITY OF ERRORS			
TEST	DF	VALUE	PROB
Jarque-Bera	2	1368880.320	0.0000
DIAGNOSTICS FOR HETROSKEDEASTICITY			
RANDOM COEFFICIENTS			
TEST	DF	VALUE	PROB
Breusch-Pagan test	7	1565.566	0.0000
Koenker-Bassett test	7	40.537	0.0000
DIAGNOSTICS FOR SPATIAL DEPENDENCE			
TEST	MI/DF	VALUE	PROB
Lagrange Multiplier (lag)	1	255.124	0.0000
Robust LM (lag)	1	13.448	0.0002
Lagrange Multiplier (error)	1	276.862	0.0000
Robust LM (error)	1	35.187	0.0000
Lagrange Multiplier (SARMA)	2	290.310	0.0000
===== END OF REPORT =====			
==			

Results are largely consistent with the original model. Also, incidentally, the number of pools surrounding a property does not appear to have any significant effect on the price of a given property. This could be for a host of reasons: maybe Airbnb customers do not value the number of pools surrounding a property where they are looking to stay; but maybe they do but our dataset only allows us to capture the number of pools in *other* Airbnb properties, which is not necessarily a good proxy of the number of pools in the immediate surroundings of a given property.

Spatially lagged endogenous regressors (wY)

In a similar way to how we have included the spatial lag, one could think the prices of houses surrounding a given property also enter its own price function. In math terms, this implies the following:

$$\ln(P_i) = \alpha + \lambda \sum_j w_{ij} \ln(P_j) + \beta X_i + \epsilon_i$$

This is essentially what we call a *spatial lag* model in spatial econometrics. Two calls for caution:

1. Unlike before, this specification *does* violate some of the assumptions on which OLS relies. In particular, it is including an endogenous variable on the right-hand side. This means we need a new estimation method to obtain reliable coefficients. The technical details of this go well beyond the scope of this workshop (although, if you are interested, go check [Anselin &](#)

Rey, 2015). But we can offload those to `PysAL` and use the `GM_Lag` class, which implements the state-of-the-art approach to estimate this model.

2. A more conceptual *gotcha*: you might be tempted to read the equation above as the effect of the price in neighboring locations j on that of location i . This is not exactly the exact interpretation. Instead, we need to realize this is all assumed to be a "joint decision": rather than some houses setting their price first and that having a subsequent effect on others, what the equation models is an interdependent process by which each owner sets her own price *taking into account* the price that will be set in neighboring locations. This might read a bit like a technical subtlety and, to some extent, it is; but it is important to keep it in mind when you are interpreting the results.

Let us see how you would run this using `PysAL` :

```
m3 = ps.spreg.GM_Lag(y.values[:, None], yxs.drop('price', axis=1).values, \
                      w=w, spat_diag=True, \
                      name_x=yxs.drop('price', axis=1).columns.tolist(), name_y='ln(pr\
ice)')  
  
print(m3.summary)
```

```

REGRESSION
-----
SUMMARY OF OUTPUT: SPATIAL TWO STAGE LEAST SQUARES
-----
Data set : unknown
Weights matrix : unknown
Dependent Variable : ln(price) Number of Observations: 57
67
Mean dependent var : 5.1952 Number of Variables :
8
S.D. dependent var : 0.9455 Degrees of Freedom :
59
Pseudo R-squared : 0.4224
Spatial Pseudo R-squared: 0.4056

-----
-- Variable Coefficient Std.Error z-Statistic Probabilit
ty
-----
-- CONSTANT 3.7085715 0.1075621 34.4784213 0.00000
00
host_listings_count -0.0000587 0.0001765 -0.3324585 0.73954
30
bathrooms 0.2857932 0.0193237 14.7897969 0.00000
00
bedrooms 0.3272598 0.0157132 20.8270544 0.00000
00
beds 0.0239548 0.0095848 2.4992528 0.01244
55
guests_included 0.0065147 0.0059651 1.0921407 0.27477
13
pool 0.0891100 0.0218383 4.0804521 0.00004
49
W_ln(price) 0.0785059 0.0212424 3.6957202 0.00021
93
-----
-- Instrumented: W_ln(price)
Instruments: W_bathrooms, W_bedrooms, W_beds, W_guests_included,
W_host_listings_count, W_pool

DIAGNOSTICS FOR SPATIAL DEPENDENCE
TEST MI/DF VALUE PROB
Anselin-Kelejian Test 1 31.545 0.0000
===== END OF REPORT =====
==
```

As we can see, results are again very similar in all the other variable. It is also very clear that the estimate of the spatial lag of price is statistically significant. This points to evidence that there are processes of spatial interaction between property owners when they set their price.

Prediction performance of spatial models

Even if we are not interested in the interpretation of the model to learn more about how alternative factors determine the price of an Airbnb property, spatial econometrics can be useful. In a purely predictive setting, the use of explicitly spatial models is likely to improve accuracy in cases where space plays a key role in the data generating process. To have a quick look at this issue, we can use the mean squared error (MSE), a standard metric of accuracy in the machine learning literature, to evaluate whether explicitly spatial models are better than traditional, non-spatial ones:

```
from sklearn.metrics import mean_squared_error as mse

mses = pd.Series({'OLS': mse(y, m1.predy.flatten()), \
                  'OLS+W': mse(y, m2.predy.flatten()), \
                  'Lag': mse(y, m3.predy_e)})
mses.sort_values()
```

Lag	0.531327
OLS+W	0.532402
OLS	0.532545
<code>dtype</code> :	<code>float64</code>

We can see that the inclusion of the number of surrounding pools (which was insignificant) only marginally reduces the MSE. The inclusion of the spatial lag of price, however, does a better job at improving the accuracy of the model.

Exercise

Run a regression including both the spatial lag of pools and of the price. How does its predictive performance compare?

Development workflow

Dependencies

In addition to the packages required to run the tutorial (see the [install guide](#) for more detail), you will need the following libraries:

- `npm` and `node.js`
- `gitbook`
- `make`
- `cp`, `rm`, and `zip` Unix utilities.

Workflow

The overall structure of the workflow is as follows:

1. Develop material on Jupyter notebooks and place them under the `content/` folder.
2. When you want to build the website with the new content run on, the root folder:

```
> make notebooks
```

3. When you want to obtain a new version of the pdf or ebook formats, run on the root folder:

```
> make book
```

4. When you want to push a new version to the website to Github Pages, make sure to commit all your changes first on the `master` branch (assuming your remote is named as `origin`):

```
> git add .
> git commit -m "commit message"
> git push origin master
```

Then you can run:

```
> make website
```

This will compile a new version of the website, pdf, epub and mobi files, check them in, switch to the `gh-pages` branch, check the new version of the website and push it to Github.

