CHAPTER 11

Embedding the JavaScript D3 Library in the IPython Notebook

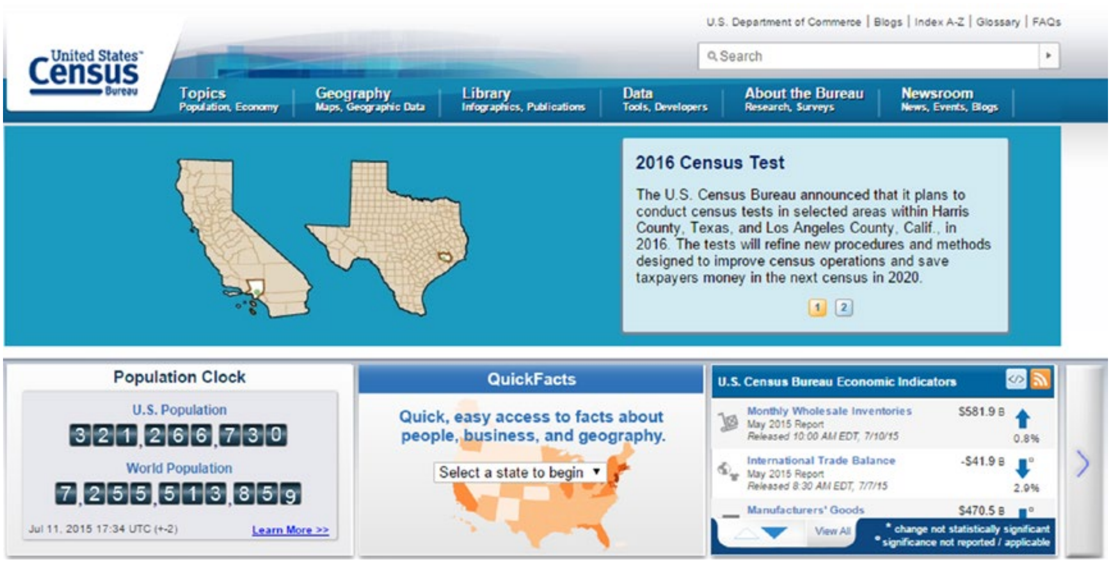
In this chapter, you will see how to extend the capabilities of the graphical representation including the JavaScript D3 library in your Jupyter Notebook. This library has enormous potential graphics and allows you to build graphical representations that even the matplotlib library cannot represent.

In the course of the various examples you will see how you can implement JavaScript code in a Python environment, using the large capacity of the integrative Jupyter Notebook. Also you'll see different ways to use the data contained in Pandas dataframes Pandas in representations based on JavaScript code.

The Open Data Source for Demographics

In this chapter, you will use demographic data as the dataset on which to perform the analysis. A good starting point is the one suggested in the Web article "Embedding Interactive Charts on an IPython Notebook," written by Agustin Barto (`http://www.machinalis.com/blog/embedding-interactive-charts-on-an-ipython-nb/`). This article suggested the site of the United States Census Bureau (`http://www.census.gov`) as the data source for demographics (see Figure 11-1).

F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-3913-1_11


U.S. Department of Commerce | Blogs | Index A-Z | Glossary | FAQs
United States Census Bureau
Search
Topics Population, Economy
Geography Maps, Geographic Data
Library Infographics, Publications
Data Tools, Developers
About the Bureau Research, Surveys
Newsroom News, Events, Blogs
1 2
Population Clock
U.S. Population
321,266,730
World Population
7,255,513,859
Jul 11, 2015 17:34 UTC (+-2)
Learn More >>
QuickFacts
Quick, easy access to facts about people, business, and geography.
Select a state to begin
U.S. Census Bureau Economic Indicators
Monthly Wholesale Inventories
May 2015 Report
Released 10:00 AM EDT, 7/10/15
$581.9 B
0.8%
International Trade Balance
May 2015 Report
Released 8:30 AM EDT, 7/7/15
-$41.9 B
2.9%
Manufacturers' Goods
$470.5 B
View All
* change not statistically significant
° significance not reported / applicable

2016 Census Test

The U.S. Census Bureau announced that it plans to conduct census tests in selected areas within Harris County, Texas, and Los Angeles County, Calif., in 2016. The tests will refine new procedures and methods designed to improve census operations and save taxpayers money in the next census in 2020.

Figure 11-1. *This is the home page of the United States Census Bureau*

The United States Census Bureau is part of the United States Department of Commerce, and is officially in charge of collecting demographic data on the U.S. population and reporting statistics about it. Its site provides a large amount of data as CSV files, which, as you have seen in previous chapters, are easily imported in the form of Pandas dataframes.

For the purposes of this chapter, you want the data that estimates the population of each state and counties in the United States. A CSV file that contains all of this information is `CO-EST2014-alldata.csv`.

So first, open Jupyter Notebook and in the first frame, import all aspects of the Python library that could later be needed in any page of IPython Notebook.

```
import numpy as np
import pandas as pd
import matplotlib.pyplot as plt
%matplotlib inline
```

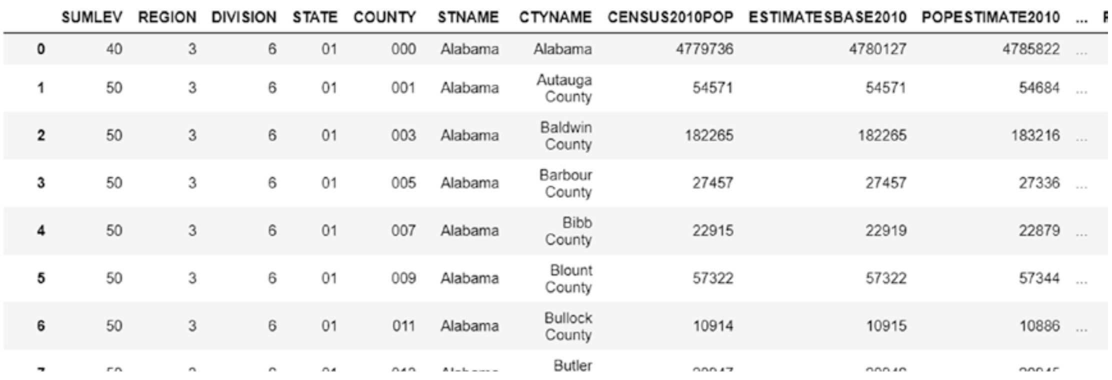
Now that you have all the necessary libraries, you can start by importing data from Census.gov in your Notebook. So you need to upload the `CO-EST2014-alldata.csv` file directly in the form of a Pandas dataframe. The `pd.read_csv()` function will convert tabular data contained in a CSV file to a Pandas dataframe, which you will name `pop2014`. Using the `dtype` option, you can force the interpretation of some fields that could be interpreted as numbers, as strings instead.

```
url = "https://raw.githubusercontent.com/dwdii/IS608-VizAnalytics/master/
FinalProject/Data/CO-EST2014-alldata.csv"
pop2014 =pd.read_csv(url,encoding='latin-1',dtype={'STATE': 'str',
'COUNTY': 'str'})
```

Once you have acquired and collected data in the `pop2014` dataframe, you can see how they are structured by simply writing:

```
pop2014
```

You will obtain an image like that shown in Figure 11-2.



	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	[illegible]	Butler	[illegible]	[illegible]	[illegible]	

Figure 11-2. *The pop2014 dataframe contains all demographics for the years from 2010 to 2014*

Carefully analyzing the nature of the data, you can see how they are organized within the dataframe. The `SUMLEV` column contains the geographic level of the data; for example, 40 indicates a state and 50 indicates data covering a single county.

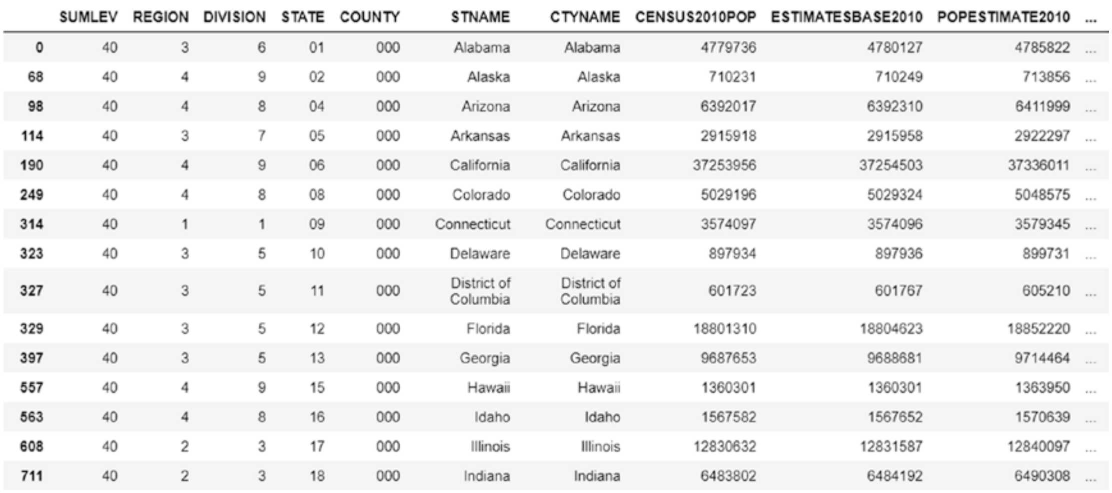
The `REGION`, `DIVISION`, `STATE`, and `COUNTY` columns contain hierarchical subdivisions of all areas in which the U.S. territory has been divided. `STNAME` and `CTYNAME` indicate the name of the state and the county, respectively. The following columns contain the data on population. `CENSUS2010POP` is the column that contains the actual data on the population, that is, the data that were collected by the 2010 census. Following that are other columns with the population estimates calculated for each year. In this example, you can see 2010 (2011, 2012, 2013, and 2014 are also in the dataframe but not shown in Figure 11-2).

You will use these values of population estimates as data to be represented in the examples discussed in this chapter.

The `pop2014` dataframe contains a large number of columns and rows that you are not interested in, so it is smart to eliminate unnecessary information. First, you are interested in the values of the people who relate to entire states, and so you extract only the rows with `SUMLEV` equal to 40. Collect these data within the `pop2014_by_state` dataframe.

```
pop2014_by_state = pop2014[pop2014.SUMLEV == 40]
pop2014_by_state
```

You get a dataframe like the one shown in Figure 11-3.



	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
0	40	3	6	01	000	Alabama	Alabama	4779736	4780127	4785822	...
68	40	4	9	02	000	Alaska	Alaska	710231	710249	713856	...
98	40	4	8	04	000	Arizona	Arizona	6392017	6392310	6411999	...
114	40	3	7	05	000	Arkansas	Arkansas	2915918	2915958	2922297	...
190	40	4	9	06	000	California	California	37253956	37254503	37336011	...
249	40	4	8	08	000	Colorado	Colorado	5029196	5029324	5048575	...
314	40	1	1	09	000	Connecticut	Connecticut	3574097	3574096	3579345	...
323	40	3	5	10	000	Delaware	Delaware	897934	897936	899731	...
327	40	3	5	11	000	District of Columbia	District of Columbia	601723	601767	605210	...
329	40	3	5	12	000	Florida	Florida	18801310	18804623	18852220	...
397	40	3	5	13	000	Georgia	Georgia	9687653	9688681	9714464	...
557	40	4	9	15	000	Hawaii	Hawaii	1360301	1360301	1363950	...
563	40	4	8	16	000	Idaho	Idaho	1567582	1567652	1570639	...
608	40	2	3	17	000	Illinois	Illinois	12830632	12831587	12840097	...
711	40	2	3	18	000	Indiana	Indiana	6483802	6484192	6490308	...

***Figure 11-3.** The pop2014_by_state dataframe contains all demographics related to the states*

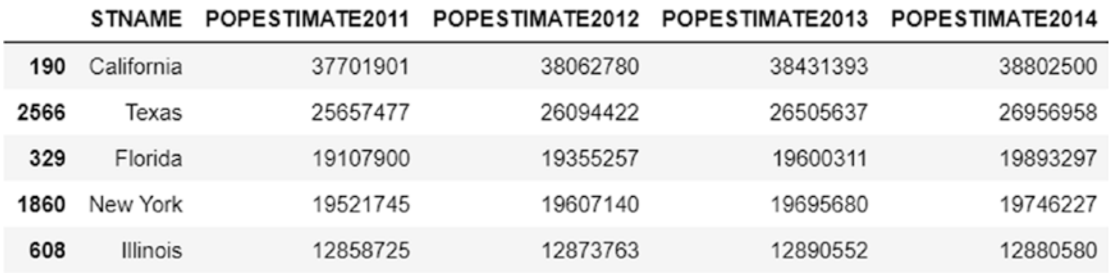
However, the dataframe just obtained still contains too many columns with unnecessary information. Given the high number of columns, instead of carrying out their removal with the `drop()` function, it is more convenient to perform an extraction.

```
states = pop2014_by_state[['STNAME','POPESTIMATE2011', 'POPESTIMATE2012',
'POPESTIMATE2013','POPESTIMATE2014']]
```

Now that you have the essential information needed, you can start to make graphical representations. For example, you could determine the five most populated states in the country.

```
states.sort_values(['POPESTIMATE2014'], ascending=False)[:5]
```

Listing them in descending order, you will receive a dataframe as shown in Figure 11-4.



	STNAME	POPESTIMATE2011	POPESTIMATE2012	POPESTIMATE2013	POPESTIMATE2014
190	California	37701901	38062780	38431393	38802500
2566	Texas	25657477	26094422	26505637	26956958
329	Florida	19107900	19355257	19600311	19893297
1860	New York	19521745	19607140	19695680	19746227
608	Illinois	12858725	12873763	12890552	12880580

Figure 11-4. *The five most populous states in the United States*

For example, you could use a bar chart to represent the five most populous states in descending order. This work is easily achieved using matplotlib, but in this chapter, you will take advantage of this simple representation to see how you can use the JavaScript D3 library to create the same representation.

The JavaScript D3 Library

D3 is a JavaScript library that allows direct inspection and manipulation of the DOM object (HTML5), but it is intended solely for data visualization and it does its job excellently. In fact, the name D3 is derived from the three Ds contained in "data-driven documents." D3 was entirely developed by Mike Bostock.

This library is proving to be very versatile and powerful, thanks to the technologies upon which it is based: JavaScript, SVG, and CSS. D3 combines powerful visualization components with a data-driven approach to the DOM manipulation. In so doing, D3 takes full advantage of the capabilities of the modern browser.

Given that even Jupyter Notebooks are Web objects and use the same technologies that are the basis of the current browser, the idea of using this library, although JavaScript, within the notebook is not as preposterous as it may seem at first.

For those not familiar with the JavaScript D3 library and want to know more about this topic, I recommend reading another book, entitled “Create Web Charts with D3,” by F. Nelli (Apress, 2014).

Indeed, Jupyter Notebook has the magic function `%% javascript` to integrate JavaScript code within the Python code.

But the JavaScript code, in a manner similar to Python, requires you to import some libraries. The libraries are available online and must be loaded each time you launch the execution. In HTML, the process of importing library has a particular construct:

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js">
</script>
```

This is an HTML tag. To make the import within an Jupyter Notebook, you should instead use this different construct:

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min'
    }
});
```

Using `require.config()`, you can import all the necessary JavaScript libraries.

In addition, if you are familiar with HTML code, you will know for sure that you need to define CSS styles if you want to strengthen the capacity of visualization of an HTML page. In parallel, also in the Jupyter Notebook, you can define a set of CSS styles. To do this you can write HTML code, thanks to the `HTML()` function belonging to the `IPython.core.display` module. Therefore, make the appropriate CSS definitions as follows:

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.bar {
   fill: steelblue;
}
.bar:hover{
   fill: brown;
}

.axis {
   font: 10px sans-serif;
}

.axis path,

.axis line {
   fill: none;
   stroke: #000;
}

.x.axis path {
   display: none;
}

</style>
<div id="chart_d3" />
"""))
```

At the bottom of the previous code, you notice that the `<div>` HTML tag is identified as `chart_d3`. This tag identifies the location where it will be represented.

Now you have to write the JavaScript code by using the functions provided by the D3 library. Using the `Template` object provided by the `Jinja2` library, you can define dynamic JavaScript code where you can replace the text depending on the values contained in a dataframe Pandas.

If there is still not a Jinja2 library installed on your system, you can always install it with Anaconda.

```
conda install jinja2
```

Or by using

```
pip install jinja2
```

After you have installed this library, you can define the template.

```
import jinja2

myTemplate = jinja2.Template("""

require(["d3"], function(d3){

   var data = []

   {% for row in data %}
   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }}  });
   {% endfor %}

d3.select("#chart_d3 svg").remove()

    var margin = {top: 20, right: 20, bottom: 30, left: 40},
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var x = d3.scale.ordinal()
        .rangeRoundBands([0, width], .25);

    var y = d3.scale.linear()
        .range([height, 0]);
    var xAxis = d3.svg.axis()
        .scale(x)
        .orient("bottom");

    var yAxis = d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(10)
        .tickFormat(d3.format('.1s'));

    var svg = d3.select("#chart_d3").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
       .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
    x.domain(data.map(function(d) { return d.state; }));
    y.domain([0, d3.max(data, function(d) { return d.population; })]);

    svg.append("g")
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);

    svg.append("g")
        .attr("class", "y axis")
        .call(yAxis)
        .append("text")
        .attr("transform", "rotate(-90)")
        .attr("y", 6)
        .attr("dy", ".71em")
        .style("text-anchor", "end")
        .text("Population");
    svg.selectAll(".bar")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar")
        .attr("x", function(d) { return x(d.state); })
        .attr("width", x.rangeBand())
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
});
""");
```

You aren’t finished. Now is the time to launch the representation of this D3 chart you have just defined. You also need to write the commands needed to pass data contained in the Pandas dataframe to the template, so they can be directly integrated into the JavaScript code written previously. The representation of JavaScript code, or rather the template just defined, will be executed by launching the `render()` function.

```
display(Javascript(myTemplate.render(
    data=states.sort_values(['POPESTIMATE2012'], ascending=False)[:10].
    itertuples()
)))
```

The bar chart will appear in the previous frame in which the `<div>` was placed, as shown in Figure 11-5, which shows all the population estimates for the year 2014.

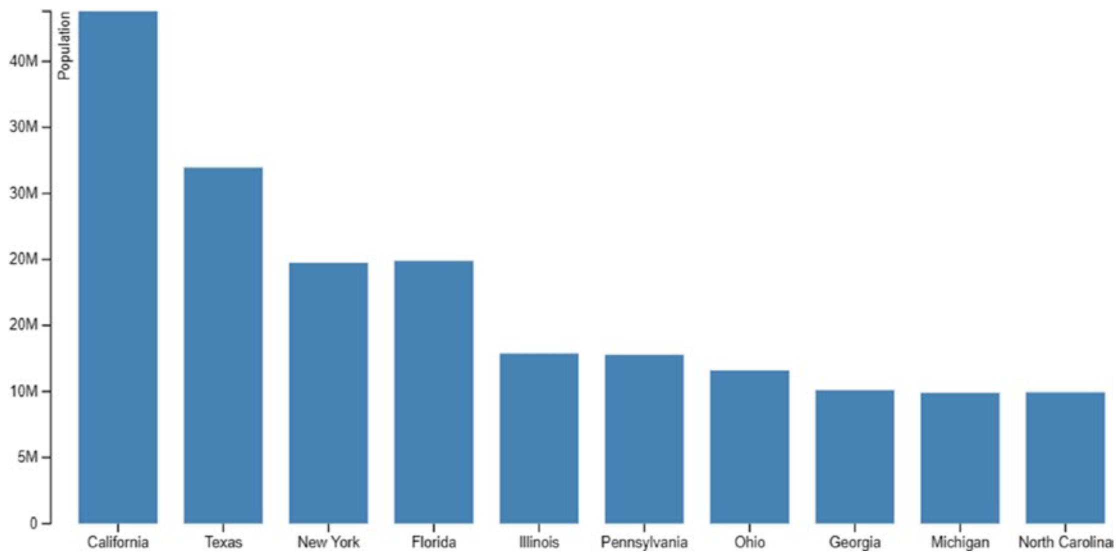


***Figure 11-5.** The five most populous states of the United States represented by a bar chart relative to 2014*

Drawing a Clustered Bar Chart

So far you have relied broadly on what had been described in the fantastic article written by Barto. However, the type of data that you extracted has given you the trend of population estimates in the last four years for the United States. A more useful chart for visualizing data would be to show the trend of the population of each state over time.

To do that, a good choice is to use a clustered bar chart, where each cluster is one of the five most populous states and each cluster will have four bars to represent the population in a given year.

At this point you can modify the previous code or write code again in your Jupyter Notebook.

```
display(HTML("""
<style>

.bar2011 {
   fill: steelblue;
}
.bar2012 {
   fill: red;
}

.bar2013 {
   fill: yellow;
}

.bar2014 {
   fill: green;
}
.axis {
   font: 10px sans-serif;
}

.axis path,

.axis line {
   fill: none;
   stroke: #000;
}

.x.axis path {
   display: none;
}

</style>
<div id="chart_d3" />
"""))
```

You have to modify the template as well, by adding the other three sets of data corresponding to the years 2011, 2012, and 2013. These years will be represented by a different color on the clustered bar chart.

```
import jinja2

myTemplate = jinja2.Template("""

require(["d3"], function(d3){
   var data = []
   var data2 = []
   var data3 = []
   var data4 = []

   {% for row in data %}
   data.push ({ 'state': '{{ row[1] }}', 'population': {{ row[2] }}  });
   data2.push({ 'state': '{{ row[1] }}', 'population': {{ row[3] }}  });
   data3.push({ 'state': '{{ row[1] }}', 'population': {{ row[4] }}  });
   data4.push({ 'state': '{{ row[1] }}', 'population': {{ row[5] }}  });
   {% endfor %}

d3.select("#chart_d3 svg").remove()

    var margin = {top: 20, right: 20, bottom: 30, left: 40},
        width = 800 - margin.left - margin.right,
        height = 400 - margin.top - margin.bottom;

    var x = d3.scale.ordinal()
        .rangeRoundBands([0, width], .25);

    var y = d3.scale.linear()
        .range([height, 0]);

    var xAxis = d3.svg.axis()
        .scale(x)
        .orient("bottom");

    var yAxis = d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(10)
        .tickFormat(d3.format('.1s'));

    var svg = d3.select("#chart_d3").append("svg")
        .attr("width", width + margin.left + margin.right)
        .attr("height", height + margin.top + margin.bottom)
        .append("g")
        .attr("transform", "translate(" + margin.left + "," +
         margin.top + ")");
```

```
x.domain(data.map(function(d) { return d.state; }));
y.domain([0, d3.max(data, function(d) { return d.population; })]);

svg.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

svg.append("g")
    .attr("class", "y axis")
    .call(yAxis)
    .append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Population");

svg.selectAll(".bar2011")
    .data(data)
    .enter().append("rect")
    .attr("class", "bar2011")
    .attr("x", function(d) { return x(d.state); })
    .attr("width", x.rangeBand()/4)
    .attr("y", function(d) { return y(d.population); })
    .attr("height", function(d) { return height - y(d.population); });

svg.selectAll(".bar2012")
    .data(data2)
    .enter().append("rect")
    .attr("class", "bar2012")
    .attr("x", function(d) { return (x(d.state)+x.rangeBand()/4); })
    .attr("width", x.rangeBand()/4)
    .attr("y", function(d) { return y(d.population); })
    .attr("height", function(d) { return height - y(d.population); });
    svg.selectAll(".bar2013")
        .data(data3)
        .enter().append("rect")
        .attr("class", "bar2013")
        .attr("x", function(d) { return (x(d.state)+2*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

    svg.selectAll(".bar2014")
        .data(data4)
        .enter().append("rect")
        .attr("class", "bar2014")
        .attr("x", function(d) { return (x(d.state)+3*x.rangeBand()/4); })
        .attr("width", x.rangeBand()/4)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });

});
""");
```

The series of data to be passed from the dataframe to the template are now four, so you have to refresh the data and the changes that you have just made to the code. Therefore, you need to rerun the code of the `render()` function.

```
display(Javascript(myTemplate.render(
    data=states.sort_values(['POPESTIMATE2014'], ascending=False)[:5].
    itertuples()
)))
```

Once you have launched the `render()` function again, you get a chart like the one shown in Figure 11-6.

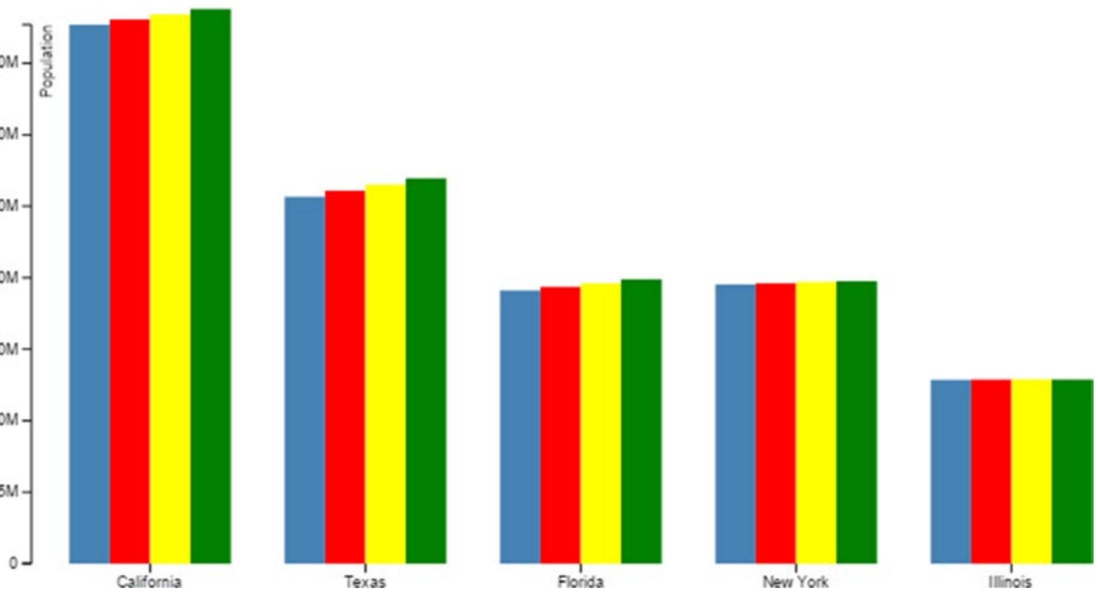


Figure 11-6. *A clustered bar chart representing the populations of the five most populous states from 2011 to 2014*

The Choropleth Maps

In the previous sections you saw how to use JavaScript code and the D3 library to represent a bar chart. Well, these achievements would have been easy with matplotlib and perhaps implemented in an even better way. The purpose of the previous code was only for educational purposes.

Something quite different is the use of much more complex views that are unobtainable by matplotlib. So now we will put in place the true potential made available by the D3 library. The *choropleth maps* are very complex types of representations.

The choropleth maps are geographical representations where the land areas are divided into portions characterized by different colors. The colors and the boundaries between a portion geographical and another are themselves representations of data.

This type of representation is very useful to represent the results of data analysis carried out on demographic or economic information, and this is also the case for data that correlates to their geographical distributions.

The representation of choropleth is based on a particular file called TopoJSON. This type of file contains all the inside information representing a choropleth map such as the United States (see Figure 11-7).

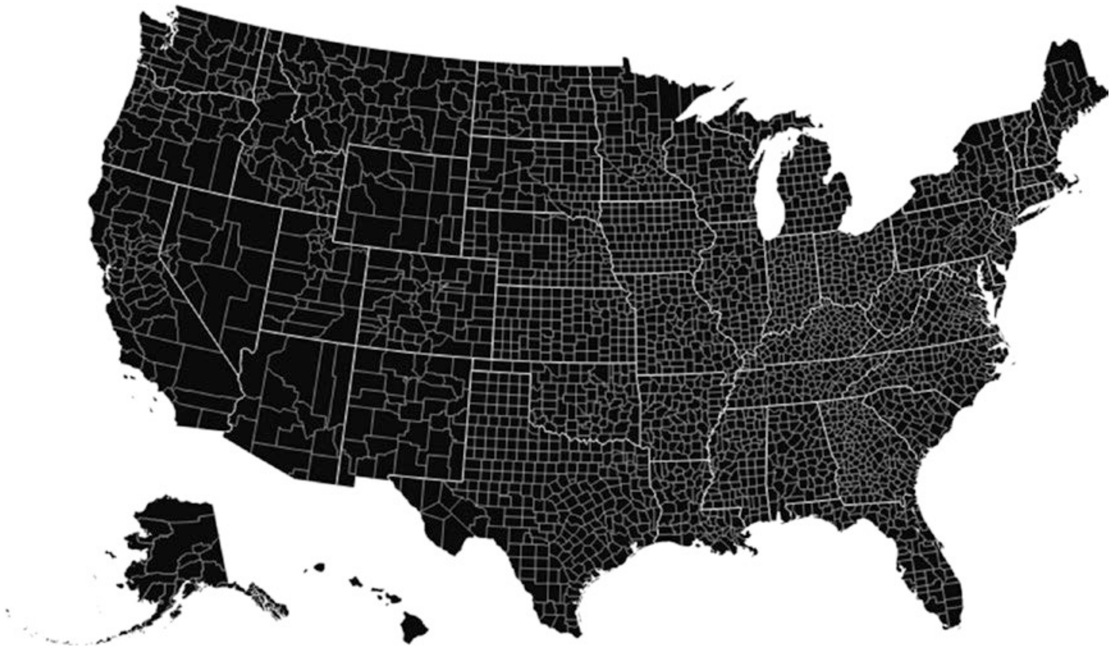


***Figure 11-7.** The representation of a choropleth map of U.S. territories with no value related to each county or state*

A good link to find such material is the U.S. Atlas TopoJSON (`https://github.com/mbostock/us-atlas`), but a lot of literature about it is available online.

A representation of this kind is not only possible but is also customizable. Thanks to the D3 library, you can correlate the geographic portions based on the value of particular columns contained in a dataframe.

First, let's start with an example already on the Internet, in the D3 library, `http://bl.ocks.org/mbostock/4060606`, but fully developed in HTML. So now you will learn how to adapt a D3 example in HTML in an IPython Notebook.

If you look at the code shown on the web page of the example, you can see that there are three necessary JavaScript libraries. This time, in addition to the D3 library, we need to import queue and TopoJSON libraries.

```
<script src="https://cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min.js">
</script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/
queue.min.js"></script>
<script src="https://cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/
topojson.min.js"></script>
```

So you have to use `require.config()` as you did in the previous sections.

```
%%javascript
require.config({
    paths: {
        d3: '//cdnjs.cloudflare.com/ajax/libs/d3/3.5.5/d3.min',
        queue: '//cdnjs.cloudflare.com/ajax/libs/queue-async/1.0.7/queue.min',
        topojson: '//cdnjs.cloudflare.com/ajax/libs/topojson/1.6.19/
                   topojson.min'
    }
});
```

The pertinent part of CSS is shown again, all within the `HTML()` function.

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
  fill: none;
}

.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth" />
"""))
```

Here is the new template that mirrors the code shown in the Bostock example, with some changes:

```
import jinja2

choropleth = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

//   var data = []

//   {% for row in data %}
//   data.push({ 'state': '{{ row[1] }}', 'population': {{ row[2] }}  });
//   {% endfor %}

d3.select("#choropleth svg").remove()

var width = 960,
    height = 600;

var rateById = d3.map();

var quantize = d3.scale.quantize()
    .domain([0, .15])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

var path = d3.geo.path()
```

```
    .projection(projection);

//row to modify
var svg = d3.select("#choropleth").append("svg")
    .attr("width", width)
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.tsv, "unemployment.tsv", function(d) { rateById.set(d.id,
    +d.rate); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a
      !== b; }))
      .attr("class", "states")
      .attr("d", path);
}
});
""");
```

Now you launch the representation, this time without any value for the template, since all values are contained in the `us.json` and `unemployment.tsv` files (you can find them in the source code of this book).

```
display(Javascript(choropleth.render()))
```

The results are identical to those shown in the Bostock example (see Figure 11-8).

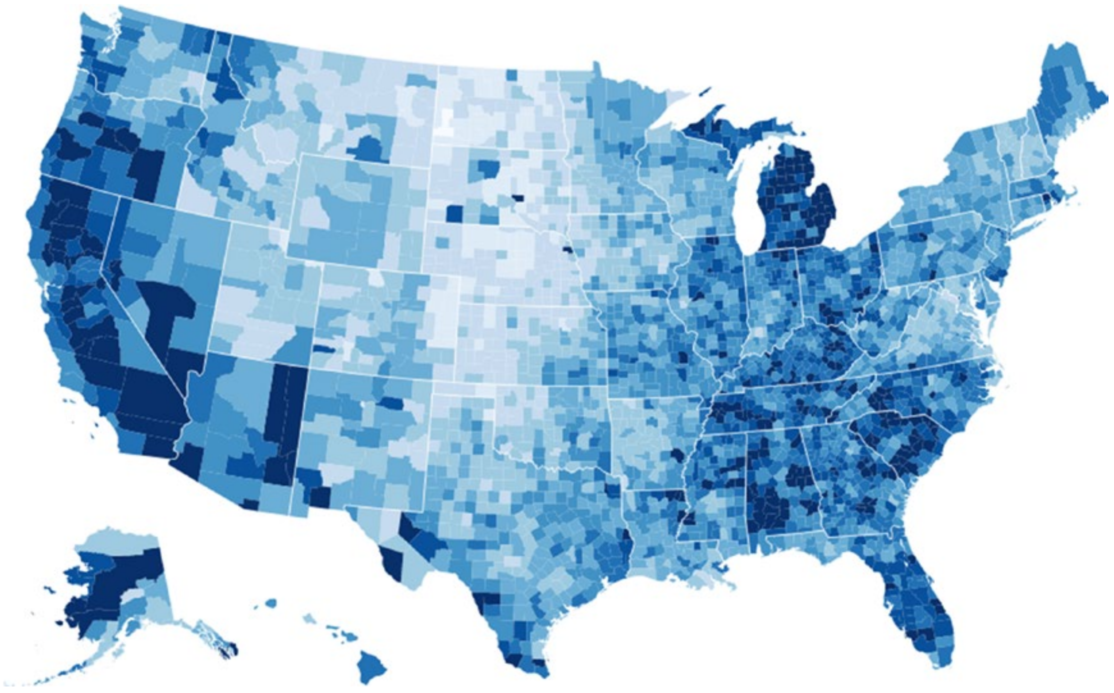


Figure 11-8. *The choropleth map of the United States with the coloring of the counties based on the values contained in the file TSV*

The Choropleth Map of the U.S. Population in 2014

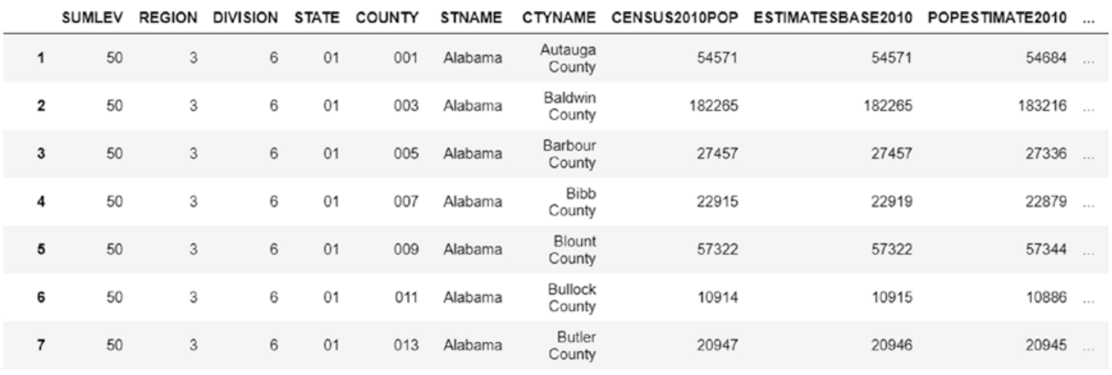
Now that you have seen how to extract demographic information from the U.S. Census Bureau and you can create the choropleth map, you can unify both things to represent a choropleth map showing the population values. The more populous the county, the deeper blue it will be. In counties with very low population levels, the hue will tend toward white.

In the first section of the chapter, you extracted information on the states by the `pop2014` dataframe. This was done by selecting the rows of the dataframe with `SUMLEV` values equal to 40. In this example, you instead need the values of the populations of each county and so you have to take out a new dataframe by taking `pop2014` using only lines with a `SUMLEV` of 50.

You must instead select the rows to level 50.

```
pop2014_by_county = pop2014[pop2014.SUMLEV == 50]
pop2014_by_county
```

You get a dataframe that contains all U.S. counties, as shown in Figure 11-9.



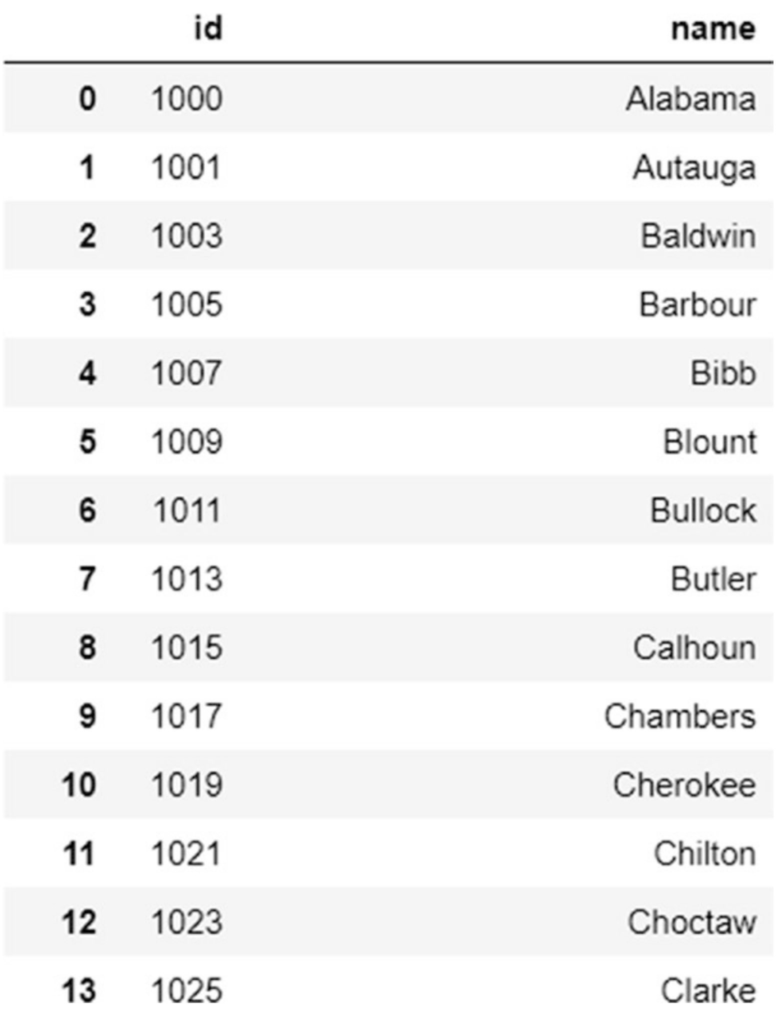
	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
1	50	3	6	01	001	Alabama	Autauga County	54571	54571	54684	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
3	50	3	6	01	005	Alabama	Barbour County	27457	27457	27336	...
4	50	3	6	01	007	Alabama	Bibb County	22915	22919	22879	...
5	50	3	6	01	009	Alabama	Blount County	57322	57322	57344	...
6	50	3	6	01	011	Alabama	Bullock County	10914	10915	10886	...
7	50	3	6	01	013	Alabama	Butler County	20947	20946	20945	...

Figure 11-9. *The pop2014_by_county dataframe contains all demographics of all U.S. counties*

You must use your data instead of the TSV previously used. Inside it, there are the ID numbers corresponding to the various counties. You can use a file on the Web to determine their names. You can download it and turn it into a dataframe.

```
USJSONnames = pd.read_table('us-county-names.tsv')
USJSONnames
```

Thanks to this file, you see the codes with the corresponding counties (see Figure 11-10).



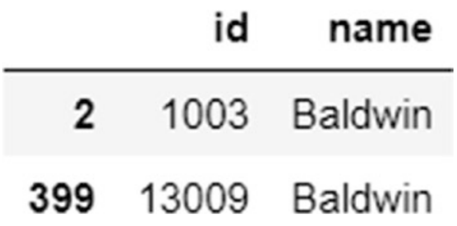
	id	name
0	1000	Alabama
1	1001	Autauga
2	1003	Baldwin
3	1005	Barbour
4	1007	Bibb
5	1009	Blount
6	1011	Bullock
7	1013	Butler
8	1015	Calhoun
9	1017	Chambers
10	1019	Cherokee
11	1021	Chilton
12	1023	Choctaw
13	1025	Clarke

Figure 11-10. *The codes of the counties are contained in the file TSV*

If you take for example the Baldwin county”

```
USJSONnames[USJSONnames['name'] == 'Baldwin']
```

You can see that there are actually two counties with the same name, but they are identified by two different identifiers (Figure 11-11).

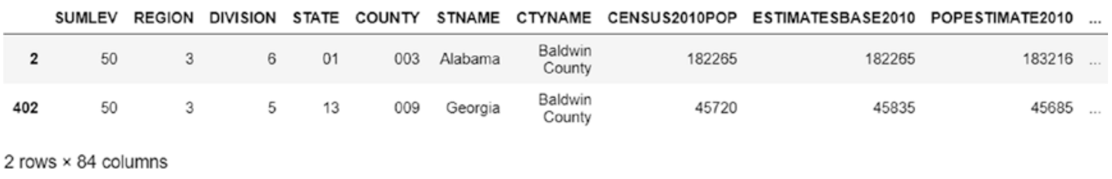


	id	name
2	1003	Baldwin
399	13009	Baldwin

Figure 11-11. *There are two Baldwin counties*

You get a table and see that there are two counties and two different codes. Now you see this in your dataframe with data taken from the data source `census.gov` (see Figure 11-12).

`pop2014_by_county[pop2014_by_county['CTYNAME'] == 'Baldwin County']`



	SUMLEV	REGION	DIVISION	STATE	COUNTY	STNAME	CTYNAME	CENSUS2010POP	ESTIMATESBASE2010	POPESTIMATE2010	...
2	50	3	6	01	003	Alabama	Baldwin County	182265	182265	183216	...
402	50	3	5	13	009	Georgia	Baldwin County	45720	45835	45685	...

2 rows × 84 columns

Figure 11-12. *The ID codes in the TSV files correspond to the combination of the values contained in the STATE and COUNTY columns*

You can recognize that there is a match. The ID contained in `TOPOJSON` matches the numbers in the `STATE` and `COUNTY` columns if combined together, but removing the 0 when it is the digit at the beginning of the code. So now you can reconstruct all the data needed to replicate the TSV example of choropleth from the `counties` dataframe. The file will be saved as `population.csv`.

```
counties = pop2014_by_county[['STATE','COUNTY','POPESTIMATE2014']]
counties.is_copy = False
counties['id'] = counties['STATE'].str.lstrip('0') + "" +
counties['COUNTY']
del counties['STATE']
del counties['COUNTY']
```

```
counties.columns = ['pop','id']
counties = counties[['id','pop']]
counties.to_csv('population.csv')
```

Now you rewrite the contents of the `HTML()` function by specifying a new `<div>` tag with the ID as `choropleth2`.

```
from IPython.core.display import display, Javascript, HTML

display(HTML("""
<style>

.counties {
  fill: none;
}

.states {
  fill: none;
  stroke: #fff;
  stroke-linejoin: round;
}

.q0-9 { fill:rgb(247,251,255); }
.q1-9 { fill:rgb(222,235,247); }
.q2-9 { fill:rgb(198,219,239); }
.q3-9 { fill:rgb(158,202,225); }
.q4-9 { fill:rgb(107,174,214); }
.q5-9 { fill:rgb(66,146,198); }
.q6-9 { fill:rgb(33,113,181); }
.q7-9 { fill:rgb(8,81,156); }
.q8-9 { fill:rgb(8,48,107); }

</style>
<div id="choropleth2" />
"""))
```

Finally, you have to define a new Template object.

```
choropleth2 = jinja2.Template("""

require(["d3","queue","topojson"], function(d3,queue,topojson){

   var data = []

d3.select("#choropleth2 svg").remove()

var width = 960,
    height = 600;

var rateById = d3.map();

var quantize = d3.scale.quantize()
    .domain([0, 1000000])
    .range(d3.range(9).map(function(i) { return "q" + i + "-9"; }));

var projection = d3.geo.albersUsa()
    .scale(1280)
    .translate([width / 2, height / 2]);

var path = d3.geo.path()
    .projection(projection);
var svg = d3.select("#choropleth2").append("svg")
    .attr("width", width)
    .attr("height", height);

queue()
    .defer(d3.json, "us.json")
    .defer(d3.csv,"population.csv", function(d) { rateById.set(d.id,
    +d.pop); })
    .await(ready);

function ready(error, us) {
  if (error) throw error;

  svg.append("g")
      .attr("class", "counties")
    .selectAll("path")
      .data(topojson.feature(us, us.objects.counties).features)
    .enter().append("path")
      .attr("class", function(d) { return quantize(rateById.get(d.id)); })
      .attr("d", path);

  svg.append("path")
      .datum(topojson.mesh(us, us.objects.states, function(a, b) { return a
       !== b; }))
      .attr("class", "states")
      .attr("d", path);
}

});

""");
```

Finally, you can execute the render() function to get the chart.

```
display(JavaScript(choropleth2.render()))
```

The choropleth map will be shown with the counties differently colored depending on their population, as shown in Figure 11-13.

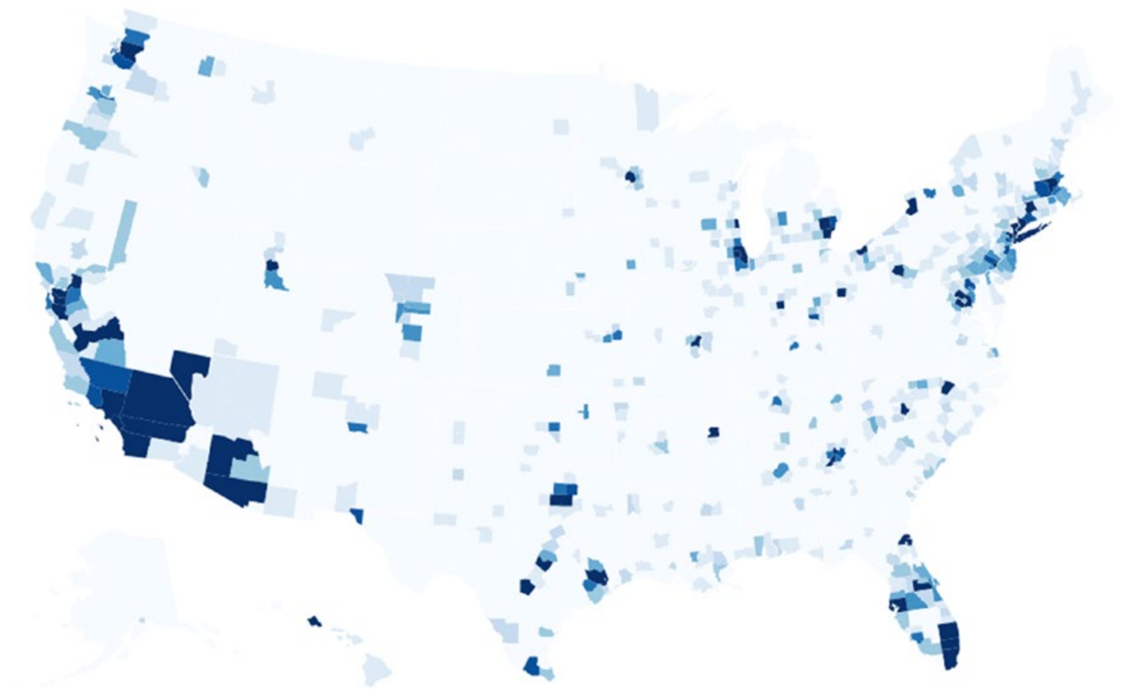


Figure 11-13. *The choropleth map of the United States shows the density of the population of all counties*

Conclusions

In this chapter, you have seen how it is possible to further extend the ability to display data using a JavaScript library called D3. Choropleth maps are just one of many examples of advanced graphics that are used to represent data. This is also a very good way to see the Jupyter Notebook in action. The world does not revolve around Python alone, but Python can provide additional capabilities for our work.

In the next chapter, you will see how to apply data analysis to images. You'll see how easy it is to build a model that can recognize handwritten numbers.