CHAPTER 11

# Embedding the JavaScript D3 Library in the IPython Notebook

In this chapter, you will learn 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 to implement JavaScript code in a Python environment, using the large capacity of the integrative Jupyter Notebook. You'll also see different ways to use the data contained in pandas dataframes and representations based on JavaScript code.

## The Open Data Source for Demographics

In this chapter, you use demographic data as the dataset on which to perform the analysis. This chapter uses the United States Census Bureau site (`www.census.gov`) as the data source for the demographics (see Figure 11-1).


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


F. Nelli, *Python Data Analytics*, https://doi.org/10.1007/978-1-4842-9532-8_11

The United States Census Bureau is part of the United States Department of Commerce, and it 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 the states and counties in the United States. On the site there is a series of datasets made available for studies at the link `www2.census.gov/programs-surveys/popest/datasets/`. Among the available datasets, look for the most recent one and download it to your computer. This example uses the CSV file called `co-est2022-alldata.csv`.

Now, open a Jupyter Notebook and import all the necessary libraries for this kind of analysis in the first cell.

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

You can start by importing data from Census.gov in your Notebook. You need to upload the `co-est2022--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 should name `pop2022`. Using the `dtype` option, you can force some fields that could be interpreted as numbers to be interpreted as strings instead.

```
pop2022 =pd.read_csv('co-est2022-alldata.csv' ,encoding='latin-1',dtype={'STATE': 'str',
'COUNTY': 'str'})
```

Once you have acquired and collected data in the `pop2022` dataframe, you can see how the data are structured by simply writing:

```
pop2022
```

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

| | SUMLEV | REGION | DIVISION | STATE | COUNTY | STNAME | CTYNAME | ESTIMATESBASE2020 | POPESTIMATE2020 | POPESTIMATE2021 | ... | RDEATH2021 | RDEATH2022 | RNATURALCHG2021 |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 40 | 3 | 6 | 01 | 000 | Alabama | Alabama | 5024356 | 5031362 | 5049846 | ... | 13.699945 | 13.210008 | -2.369557 |
| 1 | 50 | 3 | 6 | 01 | 001 | Alabama | Autauga County | 58802 | 58902 | 59210 | ... | 11.548361 | 11.313872 | 0.203197 |
| 2 | 50 | 3 | 6 | 01 | 003 | Alabama | Baldwin County | 231761 | 233219 | 239361 | ... | 12.882475 | 11.976220 | -2.865123 |
| 3 | 50 | 3 | 6 | 01 | 005 | Alabama | Barbour County | 25224 | 24960 | 24539 | ... | 15.475060 | 15.108133 | -4.323320 |
| 4 | 50 | 3 | 6 | 01 | 007 | Alabama | Bibb County | 22300 | 22183 | 22370 | ... | 14.275133 | 14.467606 | -3.321886 |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... |
| 3190 | 50 | 4 | 8 | 56 | 037 | Wyoming | Sweetwater County | 42267 | 42190 | 41582 | ... | 10.122714 | 10.515272 | 0.716230 |
| 3191 | 50 | 4 | 8 | 56 | 039 | Wyoming | Teton County | 23323 | 23377 | 23622 | ... | 3.957531 | 4.476753 | 5.404370 |
| 3192 | 50 | 4 | 8 | 56 | 041 | Wyoming | Uinta County | 20446 | 20457 | 20655 | ... | 8.999805 | 9.621196 | 2.043199 |
| 3193 | 50 | 4 | 8 | 56 | 043 | Wyoming | Washakie County | 7682 | 7658 | 7712 | ... | 14.313598 | 14.905061 | -4.814574 |
| 3194 | 50 | 4 | 8 | 56 | 045 | Wyoming | Weston County | 6840 | 6818 | 6766 | ... | 12.956419 | 14.677822 | -4.564193 |

3195 rows × 51 columns

***Figure 11-2.*** *The pop2022 dataframe contains all demographics for the years 2020, 2021, and 2022*

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. POPESTIMATE2020 is the column that contains the population estimate for 2020, followed by those for 2021 and 2022.

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

The pop2022 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 can extract only the rows with SUMLEV equal to 40. Collect these data within the pop2022_by_state dataframe.

```
pop2022_by_state = pop2022[pop2022.SUMLEV == 40]
pop2022_by_state
```

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

| | SUMLEV | REGION | DIVISION | STATE | COUNTY | STNAME | CTYNAME | ESTIMATESBASE2020 | POPESTIMATE2020 | POPESTIMATE2021 | ... | RDEATH2021 | RDEATH2022 | RNATURALCHG |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 0 | 40 | 3 | 6 | 01 | 000 | Alabama | Alabama | 5024356 | 5031362 | 5049846 | ... | 13.699945 | 13.210008 | -2.3 |
| 68 | 40 | 4 | 9 | 02 | 000 | Alaska | Alaska | 733378 | 732923 | 734182 | ... | 7.312360 | 8.715292 | 5.5 |
| 99 | 40 | 4 | 8 | 04 | 000 | Arizona | Arizona | 7151507 | 7179943 | 7264877 | ... | 11.136172 | 10.849097 | -0.6 |
| 115 | 40 | 3 | 7 | 05 | 000 | Arkansas | Arkansas | 3011555 | 3014195 | 3028122 | ... | 13.297548 | 13.232991 | -1.6 |
| 191 | 40 | 4 | 9 | 06 | 000 | California | California | 39538245 | 39501653 | 39142991 | ... | 8.787630 | 8.148586 | 1.6 |
| 250 | 40 | 4 | 8 | 08 | 000 | Colorado | Colorado | 5773733 | 5784865 | 5811297 | ... | 8.044731 | 8.476535 | 2.6 |
| 315 | 40 | 1 | 1 | 09 | 000 | Connecticut | Connecticut | 3605942 | 3597362 | 3623355 | ... | 9.597108 | 9.628722 | -0.2 |
| 325 | 40 | 3 | 5 | 10 | 000 | Delaware | Delaware | 989957 | 992114 | 1004807 | ... | 11.247315 | 11.330549 | -1.0 |
| 329 | 40 | 3 | 5 | 11 | 000 | District of Columbia | District of Columbia | 689546 | 670868 | 668791 | ... | 8.706693 | 8.288863 | 3.6 |

***Figure 11-3.*** *The pop2022_by_state dataframe contains all demographics related to the states*

The dataframe just obtained still contains too many columns with unnecessary information. Given the large number of columns, instead of removing them with the drop() function, it is more convenient to perform an extraction.

```
states = pop2022_by_state[['STNAME','POPESTIMATE2020', 'POPESTIMATE2021',
'POPESTIMATE2022']]
```

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

```
states.sort_values(['POPESTIMATE2022'], ascending=False)[:5]
```

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

| | STNAME | POPESTIMATE2020 | POPESTIMATE2021 | POPESTIMATE2022 |
|---|---|---|---|---|
| **191** | California | 39501653 | 39142991 | 39029342 |
| **2568** | Texas | 29232474 | 29558864 | 30029572 |
| **331** | Florida | 21589602 | 21828069 | 22244823 |
| **1862** | New York | 20108296 | 19857492 | 19677151 |
| **2284** | Pennsylvania | 12994440 | 13012059 | 12972008 |

***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 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 in a notebook is not as preposterous as it may seem at first, even though it's a JavaScript library.

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 called `%% javascript` that integrates JavaScript code into 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 a 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 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.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, note 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 pandas dataframe.

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 this

```
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[4] }}'  });
       {% 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 the D3 chart you 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, is executed by launching the `render()` function.

```
display(Javascript(myTemplate.render(
    data=states.sort_values(['POPESTIMATE2022'], 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 2022.

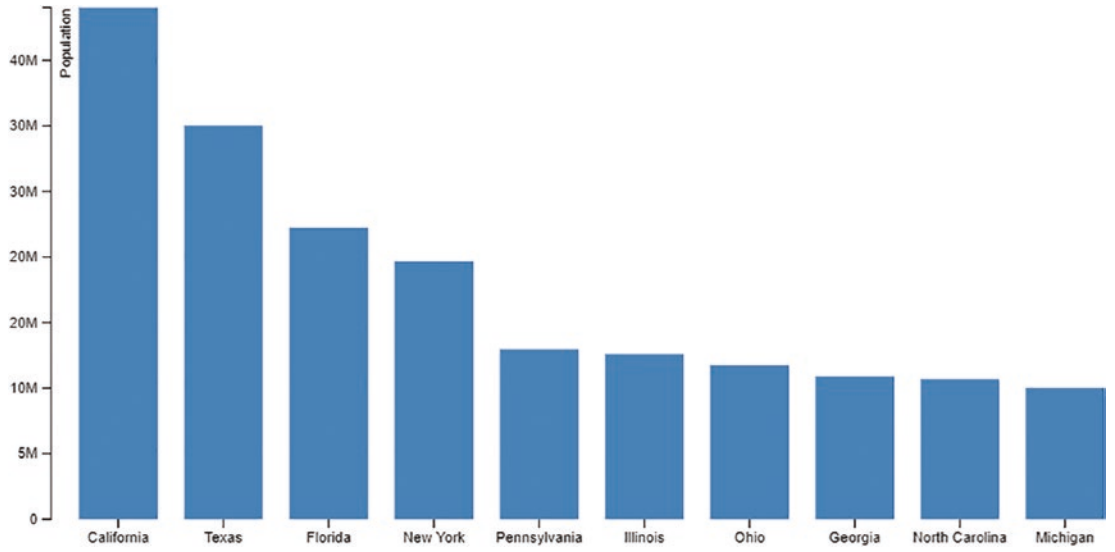


***Figure 11-5.*** *The five most populous states of the United States represented by a bar chart relative to 2022*

# 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 that represent the population in a given year.

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

```
display(HTML("""
<style>
.bar2020 {
   fill: steelblue;
}
```

```
.bar2021 {
   fill: red;
}
.bar2022 {
   fill: yellow;
}
.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 2020 and 2021. 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 = []

   {% 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] }}'  });
   {% 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(".bar2020")
        .data(data)
        .enter().append("rect")
        .attr("class", "bar2020")
        .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(".bar2021")
        .data(data2)
        .enter().append("rect")
        .attr("class", "bar2021")
        .attr("x", function(d) { return (x(d.state)+x.rangeBand()/3); })
        .attr("width", x.rangeBand()/3)
        .attr("y", function(d) { return y(d.population); })
        .attr("height", function(d) { return height - y(d.population); });
    svg.selectAll(".bar2022")
        .data(data3)
        .enter().append("rect")
        .attr("class", "bar2022")
        .attr("x", function(d) { return (x(d.state)+2*x.rangeBand()/3); })
        .attr("width", x.rangeBand()/3)
        .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 just made to the code. Therefore, you need to rerun the code of the `render()` function.

```
display(Javascript(myTemplate.render(
    data=states.sort_values(['POPESTIMATE2022'], ascending=False)[:5].itertuples()
)))
```

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

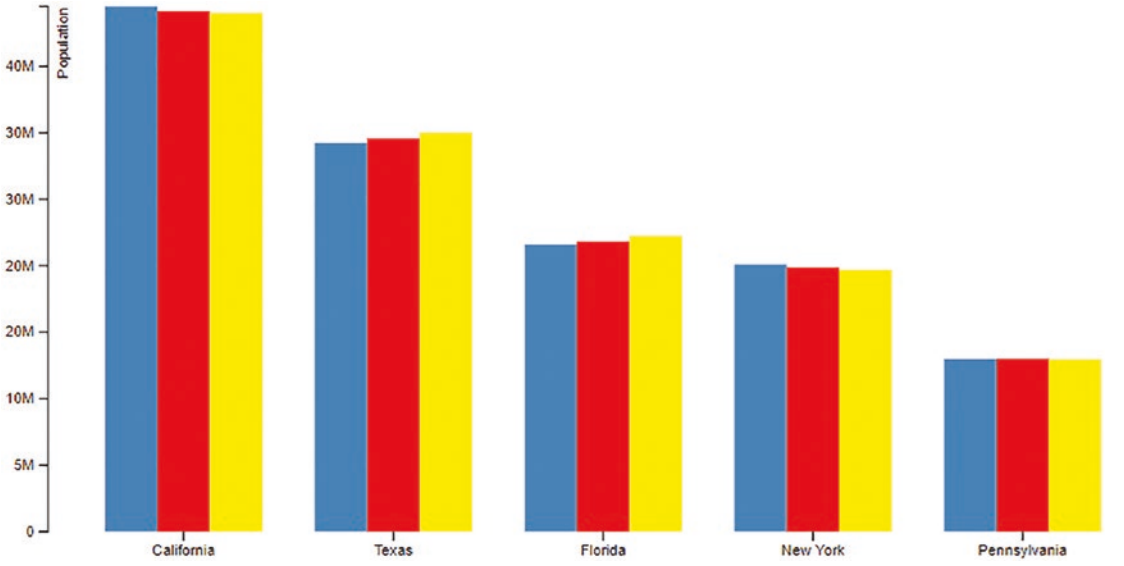


***Figure 11-6.*** *A clustered bar chart representing the populations of the five most populous states from 2020 to 2022*

# 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`. This section illustrates 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 for representing 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, 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 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, you need to import the 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>
```

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 the CSS is shown again, all within the `HTML()` function.

```
from IPython.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){
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 the 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).

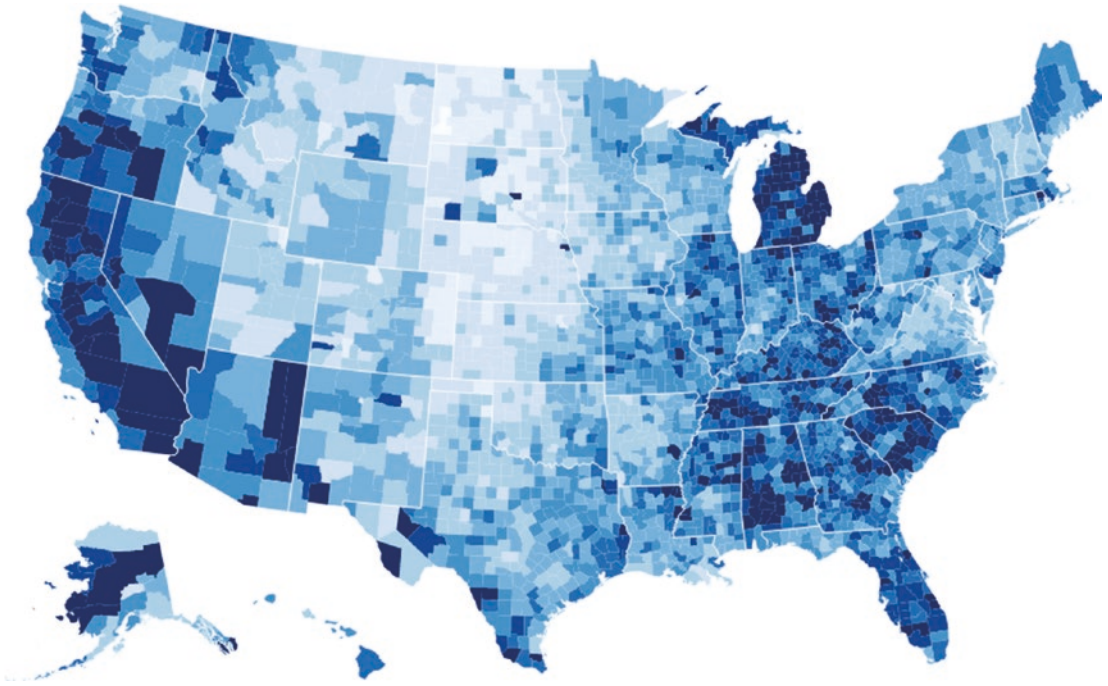

***Figure 11-8.*** *The choropleth map of the United States with the coloring of the counties based on the values contained in the file TSV*

# The Choropleth Map of the U.S. Population in 2022

Now that you have seen how to extract demographic information from the U.S. Census Bureau and you can create a 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 about the states using the `pop2022` 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. Therefore, you have to take out a new dataframe by taking `pop2022` using only lines with a `SUMLEV` of 50.

You must instead select the rows to level 50.

```
pop2022_by_county = pop2022[pop2022.SUMLEV == 50]
pop2022_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 | ESTIMATESBASE2020 | POPESTIMATE2020 | POPESTIMATE2021 | ... | RDEATH2021 | RI |
|---|---|---|---|---|---|---|---|---|---|---|---|---|---|
| 1 | 50 | 3 | 6 | 01 | 001 | Alabama | Autauga County | 58802 | 58902 | 59210 | ... | 11.548361 | |
| 2 | 50 | 3 | 6 | 01 | 003 | Alabama | Baldwin County | 231761 | 233219 | 239361 | ... | 12.882475 | |
| 3 | 50 | 3 | 6 | 01 | 005 | Alabama | Barbour County | 25224 | 24960 | 24539 | ... | 15.475060 | |
| 4 | 50 | 3 | 6 | 01 | 007 | Alabama | Bibb County | 22300 | 22183 | 22370 | ... | 14.275133 | |
| 5 | 50 | 3 | 6 | 01 | 009 | Alabama | Blount County | 59130 | 59102 | 59085 | ... | 14.637820 | |
| ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | ... | |
| 3190 | 50 | 4 | 8 | 56 | 037 | Wyoming | Sweetwater County | 42267 | 42190 | 41582 | ... | 10.122714 | |
| 3191 | 50 | 4 | 8 | 56 | 039 | Wyoming | Teton County | 23323 | 23377 | 23622 | ... | 3.957531 | |
| 3192 | 50 | 4 | 8 | 56 | 041 | Wyoming | Uinta County | 20446 | 20457 | 20655 | ... | 8.999805 | |
| 3193 | 50 | 4 | 8 | 56 | 043 | Wyoming | Washakie County | 7682 | 7658 | 7712 | ... | 14.313598 | |
| 3194 | 50 | 4 | 8 | 56 | 045 | Wyoming | Weston County | 6840 | 6818 | 6766 | ... | 12.956419 | |

3144 rows × 51 columns

***Figure 11-9.*** *The pop2022_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 |
| ... | ... | ... |
| 3349 | 78222 | Ngardmau |
| 3350 | 78224 | Ngatpang |
| 3351 | 78226 | Ngchesar |
| 3352 | 78350 | Peleliu |
| 3353 | 78370 | Sonsorol |

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

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 at `census.gov` (see Figure 11-12).

```
pop2022_by_county[pop2022_by_county['CTYNAME'] == 'Baldwin County']
```

| | SUMLEV | REGION | DIVISION | STATE | COUNTY | STNAME | CTYNAME | ESTIMATESBASE2020 | POPESTIMATE2020 | POPESTIMATE2021 | ... |
|---|---|---|---|---|---|---|---|---|---|---|---|
| 2 | 50 | 3 | 6 | 01 | 003 | Alabama | Baldwin County | 231761 | 233219 | 239361 | ... |
| 404 | 50 | 3 | 5 | 13 | 009 | Georgia | Baldwin County | 43795 | 43794 | 43671 | ... |

***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, 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 = pop2022_by_county[['STATE','COUNTY','POPESTIMATE2022']]
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.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" />
"""))
```

You also 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.

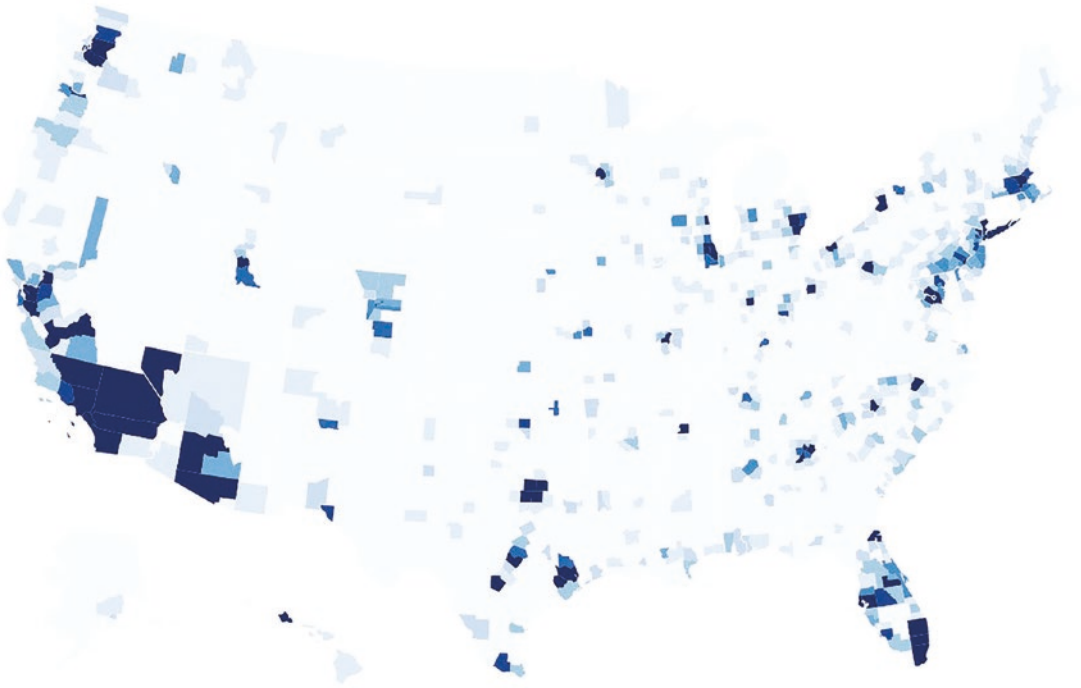

***Figure 11-13.*** *A choropleth map of the United States showing the density of the population of all counties*

# Conclusions

In this chapter, you learned 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 your work.

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