2-1, Lesson 3 - Scales, Axes, and Text

3.1 Number Scales

So far, we have seen how D3 uses data to dynamically draw and style markup elements. But what if you want to manipulate the *data* itself? Sometimes it is necessary to derive output values *as a function of* your input data—or, put another way, to map a set of input values to a different set of output values. For these scenarios, D3 provides **scales** (https://github.com/d3/d3/blob/master/API.md#scales-d3-scale) . There are five types of scales in D3 v4: continuous, sequential, quantize, threshold, and ordinal. The first four types of scales have a continuous input **domain**, or set of values of the independent variable (x) of the scale function. **Continuous scales** (https://github.com/d3/d3-scale/blob/master/README.md#continuous-scales) map the domain to a continuous **range** of output (y) values; these are useful for linear, power, and log scales (see below), axes, and **time scales** (https://github.com/d3/d3-scale/blob/master/README.md#time-scales) . **Sequential scales** (https://github.com/d3/d3-scale/blob/master/README.md#sequential-scales) are similar, but map the domain to an interpolator, or specific range function. These are most useful for creating color ramps. **Quantize scales** (https://github.com/d3/d3-scale/blob/master/README.md#sequential-scales) have a discrete range, or set of specific output values; we'll use those to create classed choropleths in the next module. Threshold scales subdivide the continuous domain according to specified class breaks and map the subsets to discreet range values. **Ordinal scales** (https://github.com/d3/d3-scale/blob/master/README.md#ordinal-scales) have a discreet domain, such as names or categories, and a discreet range.

For now, we will focus on continuous scales. D3 offers several kinds of continuous scales, including linear scales, power scales, log scales, and others. You can explore these on the API documentation page linked above. The most used type of continuous scale is the **linear scale** (https://github.com/d3/d3-scale/blob/master/README.md#linear-scales) , which simply interpolates values using linear algebra. It is important to note that there is nothing inherently *visual* about a scale; it is merely a mathematical function used to derive a new data value from an input data value (Figure 3.1). If this is difficult to understand, **Chapter 7** (http://chimera.labs.oreilly.com/books/1230000000345/ch07.html) of Murray, 2013 provides a more thorough explanation.

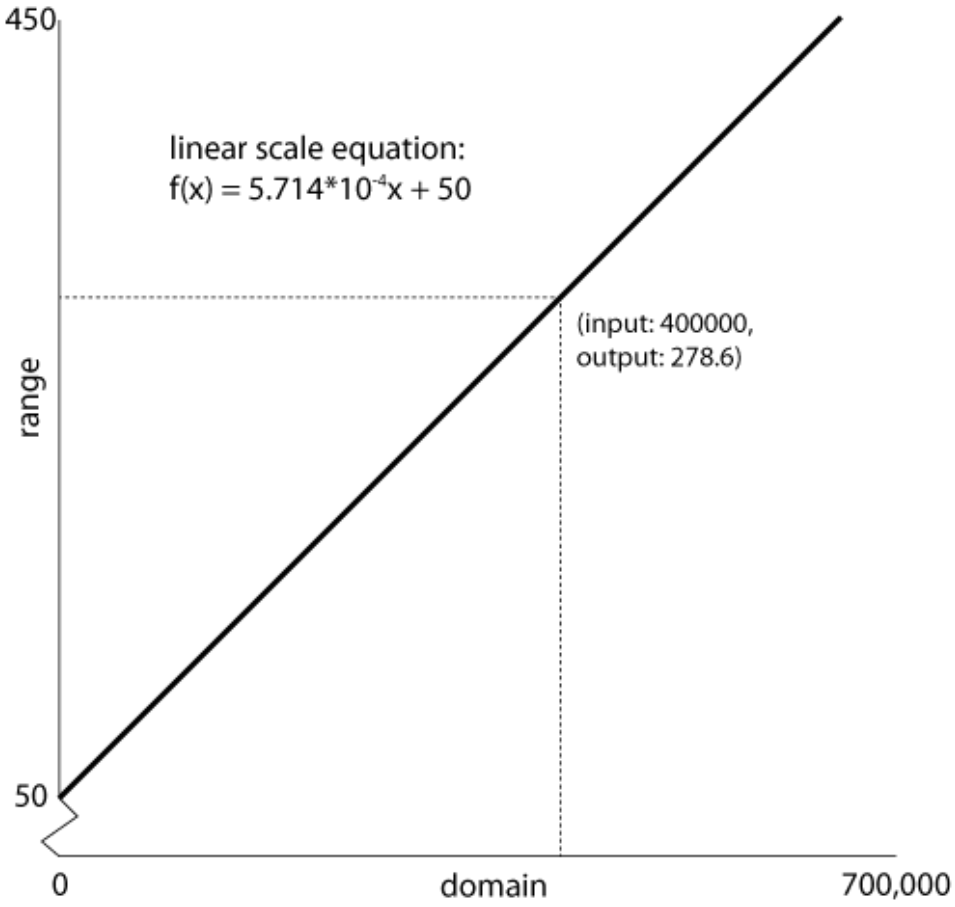

Figure 3.1: An example linear scale function with an input domain of [0, 700000] and output range of [50, 450]

How would we apply a scale? let's say we wanted to space our circles more evenly along the horizontal axis of the bubble chart. As our script stands, we are using a simple mathematical equation with the array index value as an input domain to return each circle's center x position (Example 2.8 line 36). Instead, we can create a linear scale with an array of the minimum and maximum index values as the domain and an array with our desired minimum and maximum x coordinates as the range. Note that we must create our scale *above* the `circles` block and assign it to a variable to use it in the block (Example 3.1):

Example 3.1: x coordinate linear scale in *main.js*

JavaScript

```
//above Example 2.8 line 20
var x = d3.scaleLinear()  //create the scale
    .range([90, 810]) //output min and max
    .domain([0, 3]); //input min and max
```

The operand of `x` is not a single value, object, or array. Instead, `d3.scaleLinear()` method creates what is called a **generator**. This is a *custom function* that will be used to decide where in the range each output value lies based on each input datum sent to it. We can see this function if we add the statement `console.log(x)` below the block (Figure 3.2):

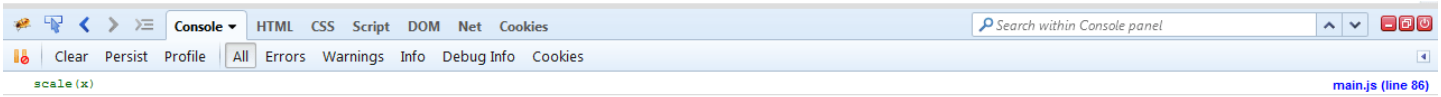

Figure 3.2: scale generator created by `d3.scaleLinear()`

When passed a value, the scale generator will determine where that value lies in the scale's domain, and interpolate between the minimum and maximum range values to generate an output value proportional to the input. The way we apply our scale, then, is by calling its variable (`x`) like a function and passing in each datum as a parameter (Example 3.2):

Example 3.2: applying the x scale to return the circles' center x coordinates in *main.js*

JavaScript

```
//Example 2.8 line 34
 .attr("cx", function(d, i){
     //use the scale generator with the index to place each circle horizontally
     return x(i);
 })
```

We can do the same sort of thing with the center y coordinate of the circles. The difference here is that we have written our equation for `cy` to return a value based on each city's population. Therefore, to create a scale for `cy`, we need to determine the minimum and maximum populations of our dataset for our input domain. While you could write a complicated custom function to pull out these values, a much simpler way to do it is to make use of D3's `.min()` (https://github.com/d3/d3-array/blob/master/README.md#min) and `.max()` (https://github.com/d3/d3-array/blob/master/README.md#max) methods. These methods take up to two parameters: the array first, and then an accessor function that tells each method where to look for the values to compare. Once we have found these values and stored them in variables, we can apply them to the domain of our `y` scale (Example 3.3):

Example 3.3: determining maximum and minimum population values in *main.js*

JavaScript

```
//above Example 2.8 line 20
//find the minimum value of the array
var minPop = d3.min(cityPop, function(d){
    return d.population;
});

//find the maximum value of the array
var maxPop = d3.max(cityPop, function(d){
    return d.population;
});

//scale for circles center y coordinate
var y = d3.scaleLinear()
    .range([440, 95])
    .domain([
        minPop,
```

```
        maxPop
    ]);
```

Note that the range is flipped, with a "minimum" value of `440` and a "maximum" of `95`. Like the subtraction in our prior equation, this ensures that higher values are associated with "up" rather than "down", since the [0,0] coordinate of the SVG is its upper-left corner. We've chosen these values because they spread our circles most evenly over the inner rectangle of the chart without overflowing it. We can now apply our scale to our population values in the `cy` anonymous function (Example 3.4):

Example 3.4: applying the y scale to return the circles' center y coordinates in *main.js*

JavaScript

```
    //Example 2.8 line 38
    .attr("cy", function(d){
        return y(d.population);
    });
```

Here are our much more evenly-spaced circles (Figure 3.3):

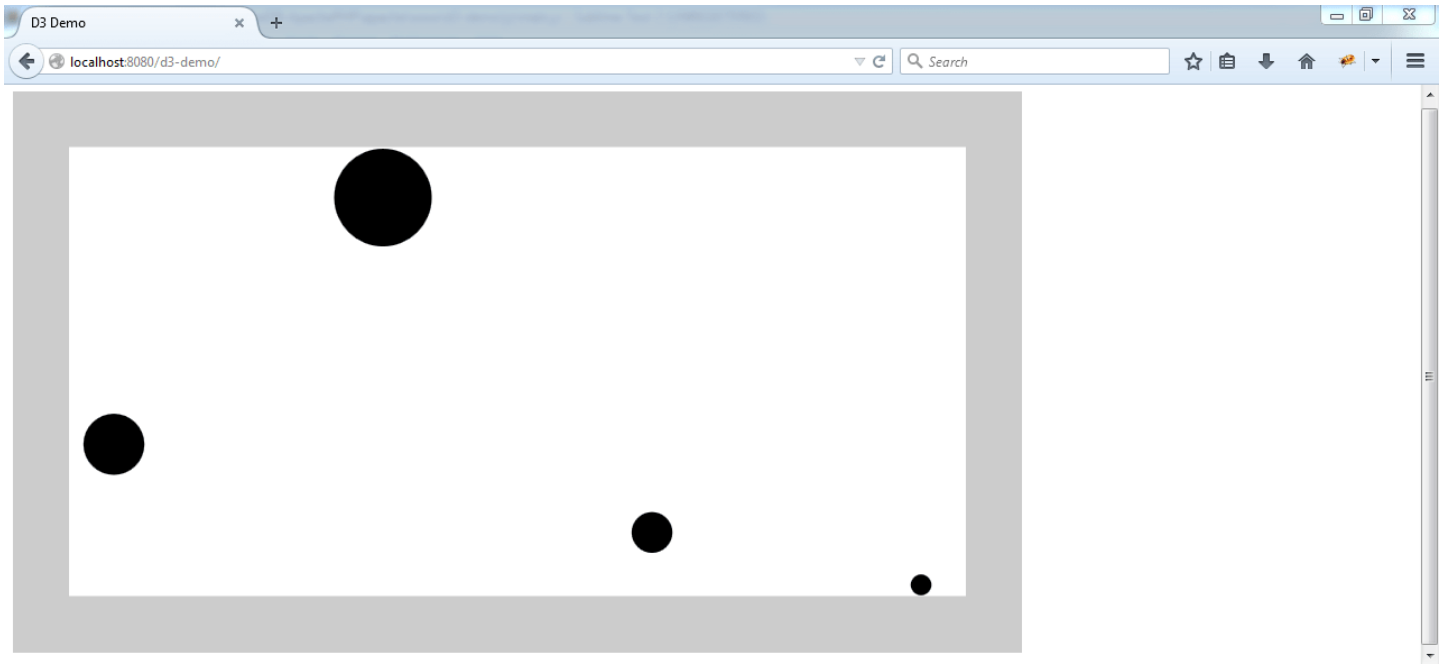

Figure 3.3: the bubble chart with scales applied

3.2 Color Scales

One nice feature of D3 scales is that they support interpolation for just about any kind of value that can be interpolated—including color. For your D3 Lab assignment, you will be creating a choropleth map, which will require the use of a color scale. Let's try out an easy one for the circles on our bubble chart, with color value corresponding to population size. We will again make use of `d3.scaleLinear()` with the same domain as the `y` scale, but this time the range will be colors (Example 3.5):

Example 3.5: implementing a color scale in *main.js*

JavaScript

```
    //above Example 2.8 line 20
    //color scale generator
    var color = d3.scaleLinear()
        .range([
            "#FDBE85",
            "#D94701"
        ])
        .domain([
            minPop,
            maxPop
        ]);

    ...

        //Example 3.4 line 1
        .attr("cy", function(d){
            return y(d.population);
        })
        .style("fill", function(d, i){ //add a fill based on the color scale generator
            return color(d.population);
        })
        .style("stroke", "#000"); //black circle stroke
```

Since our color scale generator uses only two color values for the range (lines 4-7), the result will be an unclassed color scheme, with each color derived from interpolation between the two range colors. For a classed color scheme (such as you will use in the D3 lab), you simply need to provide an array with each of the colors used for the classes as the range.

Here is the output of our simple unclassed color scale (Figure 3.4):

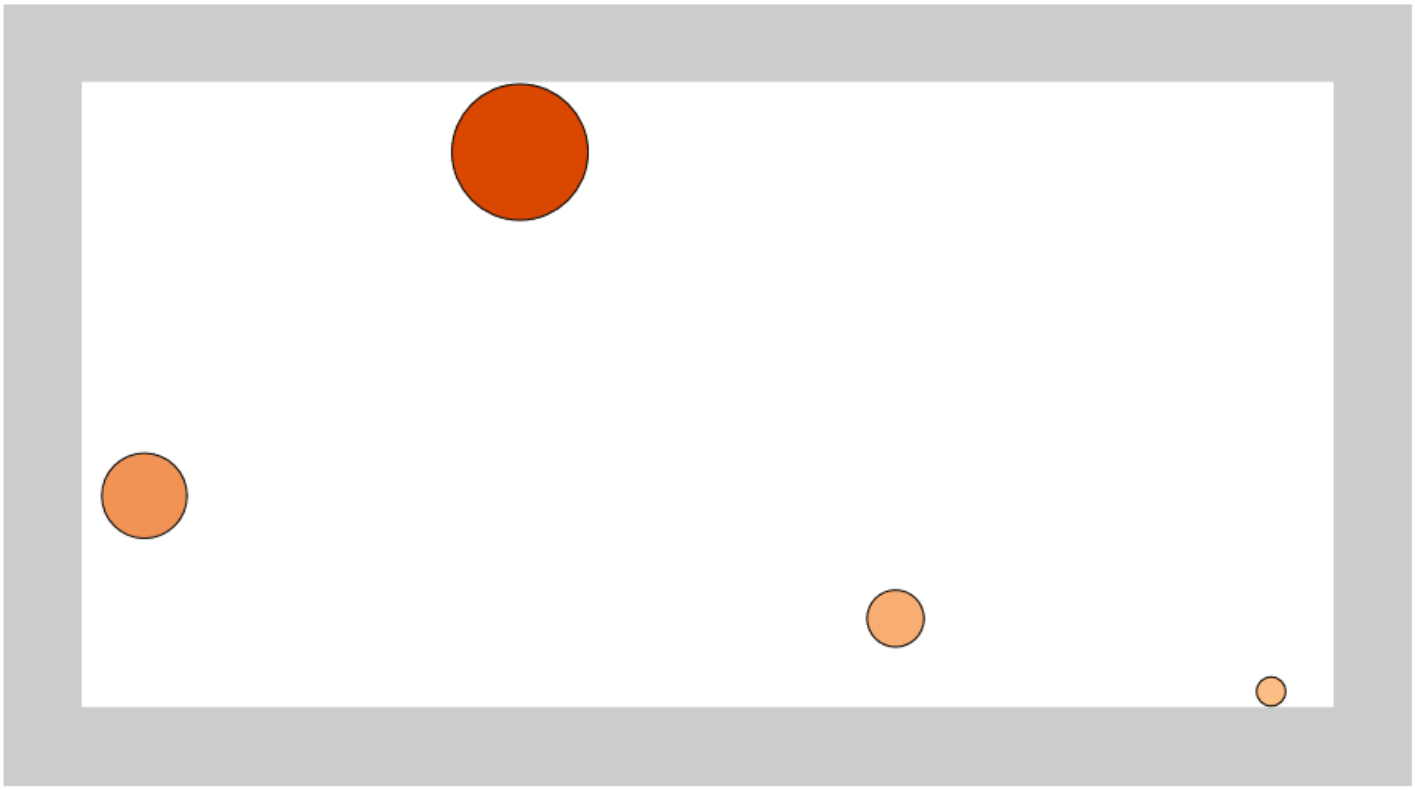

Figure 3.4: colored circles

> **Practice!** Create and apply D3 scale generators for the center x coordinate, center y coordinate, and fill color of your circles.

3.3 Axes

With four colored, proportionally-sized and -positioned circles, our bubble chart is looking pretty good—except that nobody looking at it would know how to read it! In order to make a data graphic useful, you need to give users **affordances**, or handles to contextualize the information they are seeing. One important affordance is an **axis**, the familiar reference line with tick marks and numbers that makes the graphic's scale in any one dimension visible to the user. D3 includes a module for automatically drawing axes, although it can be a bit of a trick to apply properly.

For our bubble chart, the horizontal scale is basically meaningless; it uses the data array index values as inputs and only functions to separate our circles evenly. Our vertical scale, on the other hand, makes meaningful use of our population data. Thus, it makes sense to provide the user with a vertical axis as a visual affordance for the information encoded by each circle's `"cy"` attribute. We create a vertical access on the left side of the chart based on our vertical scale using `d3.axisLeft(y)` (Example 3.6):

Example 3.6: creating the y axis generator in *main.js*

JavaScript

```
    //below Example 3.5...create y axis generator
    var yAxis = d3.axisLeft(y);
```

Just as a scale starts with creating a scale generator function, `d3.axisLeft(y)` creates an axis generator function. The `.scale()` operator (Example 3.6 line 3) feeds in our `y` scale. The `.orient()` operator (line 4) tells D3 which direction to orient the axis: bottom, top, left, or right. A left-oriented scale is a vertical scale that will appear on the left side of the SVG.

The next step is to create a new SVG element to hold the axis. The axis generator will automatically draw several new child elements, so the best thing to put them in is a `<g>` (group) element (Example 3.7):

Example 3.7: adding the y axis in *main.js*

JavaScript

```
//Example 3.6 line 1...create y axis generator
var yAxis = d3.axisLeft(y);
    .scale(y)
    .orient("left")

//create axis g element and add axis
var axis = container.append("g")
    .attr("class", "axis")
    .call(yAxis);
```

Note that we use the `.call()` method to invert the order of the code, feeding the `axis` selection to the `yAxis`. This is a useful shorthand for generator that doesn't return anything; it is functionally equivalent to (Example 3.8):

Example 3.8: Inverting `.call(yAxis)` in *main.js*

JavaScript

```
//Example 3.7 line 6...create axis g element and add axis
var axis = container.append("g")
    .attr("class", "axis");

yAxis(axis);
```

If you now refresh your browser, you won't see anything different unless you hunt for the axis in the developer tools HTML tab (Figure 3.5):

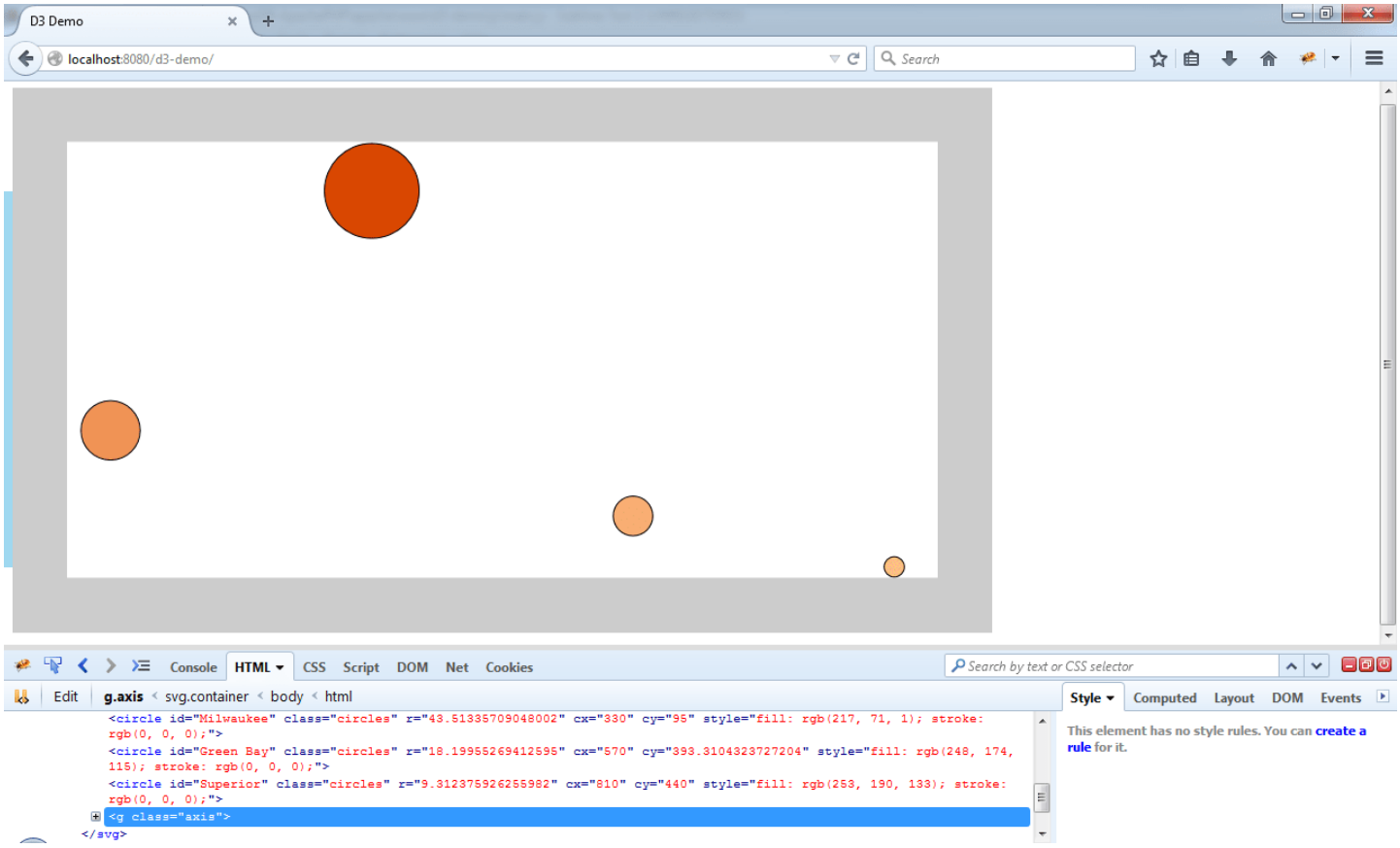

Figure 3.5: the hidden axis

Figure 3.5 shows that the axis in the DOM, but way off to the left, outside the bounds of our SVG container. This is because the `left` orientation of the axis aligns its rightmost side with the left edge of the element to which it is appended. In order to see it, we need to add a `"transform"` attribute to the `<g>` element that translates (moves) it to the right of its `0,0` coordinate (Example 3.9):

Example 3.9: translating the axis 50 pixels right in *main.js*

JavaScript

```
//Example 3.8 line 1...create axis g element and add axis
 var axis = container.append("g")
     .attr("class", "axis")
     .attr("transform", "translate(50, 0)")
     .call(yAxis);
```

Now we can see our axis, lined up properly with our inner rectangle, 50 pixels in from the left edge of the container (Figure 3.6):

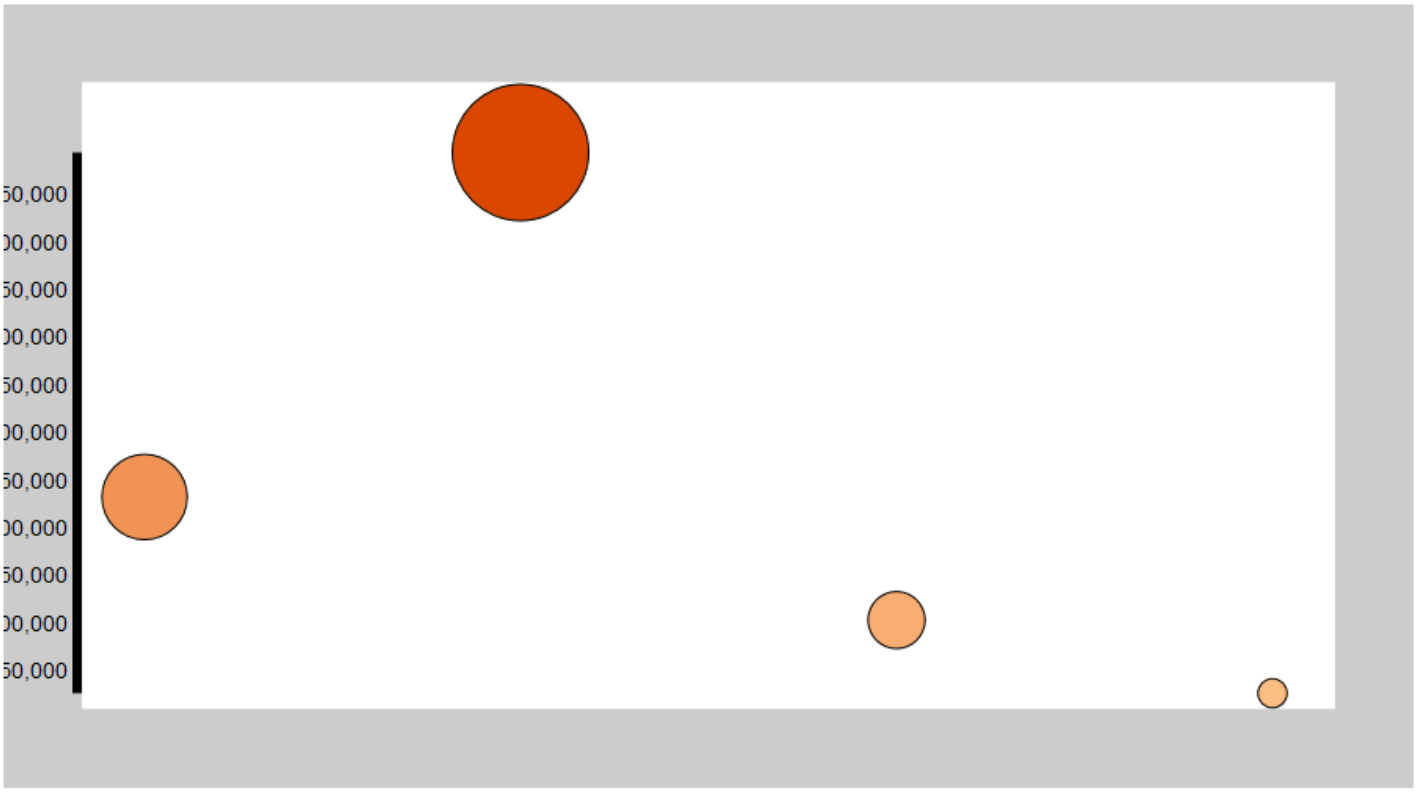

Figure 3.6: the axis, revealed

We can see our axis, but it's kind of difficult to read. We need to style it properly to make it neater. We could do this using `.style()` in our `axis` block, but since the styles we will add are not dynamically generated by the data, we may as well add them in *style.css* (Example 3.10):

Example 3.10: styling the axis in *style.css*

CSS

```
.axis path,
.axis line {
    fill: none;
    stroke: black;
    stroke-width: 1px;
    shape-rendering: crispEdges;
}

.axis text {
    font-family: sans-serif;
    font-size: 0.9em;
}
```

> **Rule!** Assign static or default styles in *style.css.*

In Example 3.10, we give all `<path>` and `<line>` elements within the axis `<g>` element styles that render crisp, thin, black lines without any fill (lines 1-7). The `.axis text` style applies to the numbers, which we make smaller so they fit in the gray border of the chart. Here is the result (Figure 3.7):

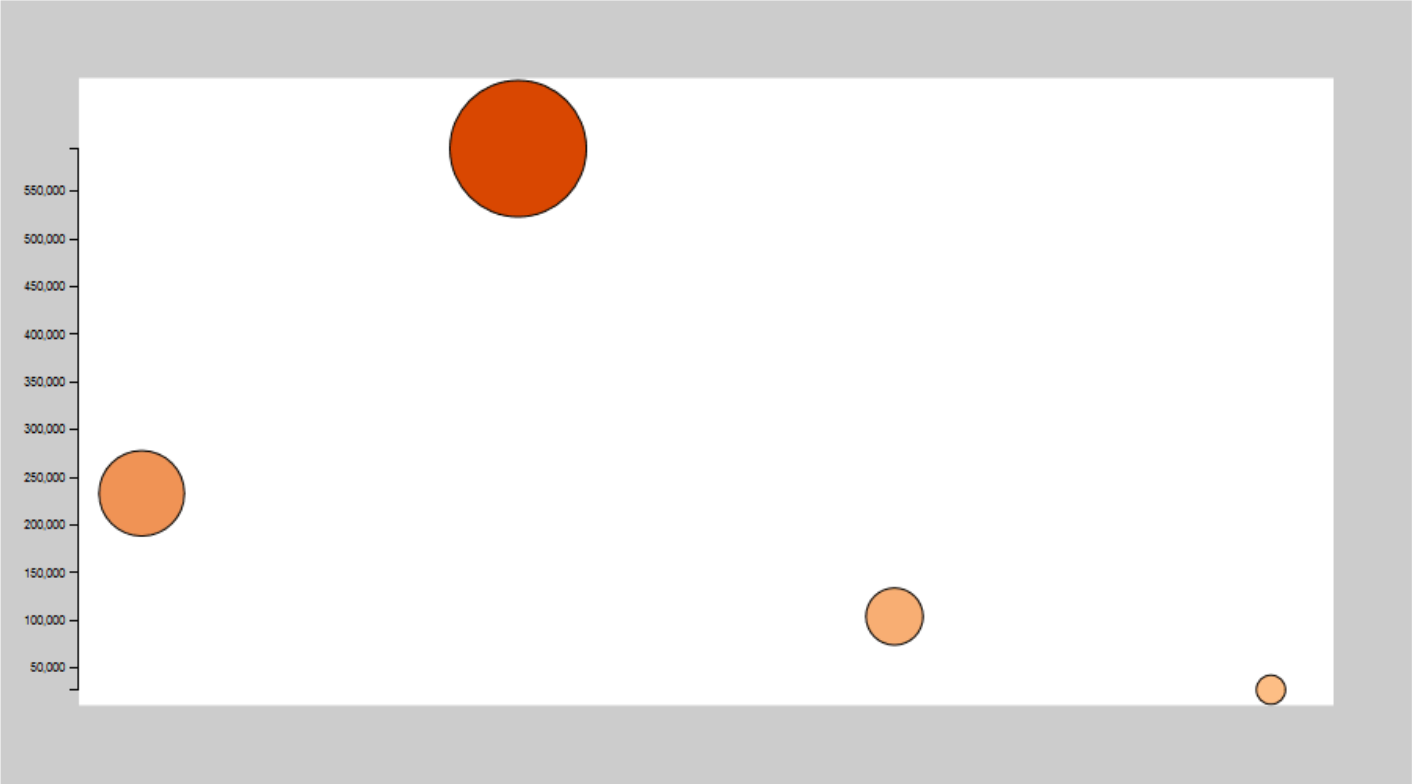

Figure 3.7: a properly-styled y axis

The only disconcerting thing about our axis is that it does not reach from the bottom of the chart to the top. Instead, it reaches from the minimum range value to the maximum range value of the `y` scale. The simple solution is to adjust our `y` scale parameters so that the domain extends to nice round numbers a bit beyond the min and max data values while the range uses the maximum and minimum y coordinates of the inner rectangle (Example 3.11):

Example 3.11: adjusting the y scale to make the axis fill the inner rectangle

JavaScript

```
//Example 3.3 line 12...scale for circles center y coordinate
var y = d3.scaleLinear()
    .range([450, 50]) //was 440, 95
    .domain([0, 700000]); //was minPop, maxPop
```

Here is the more desirable axis (Figure 3.8):

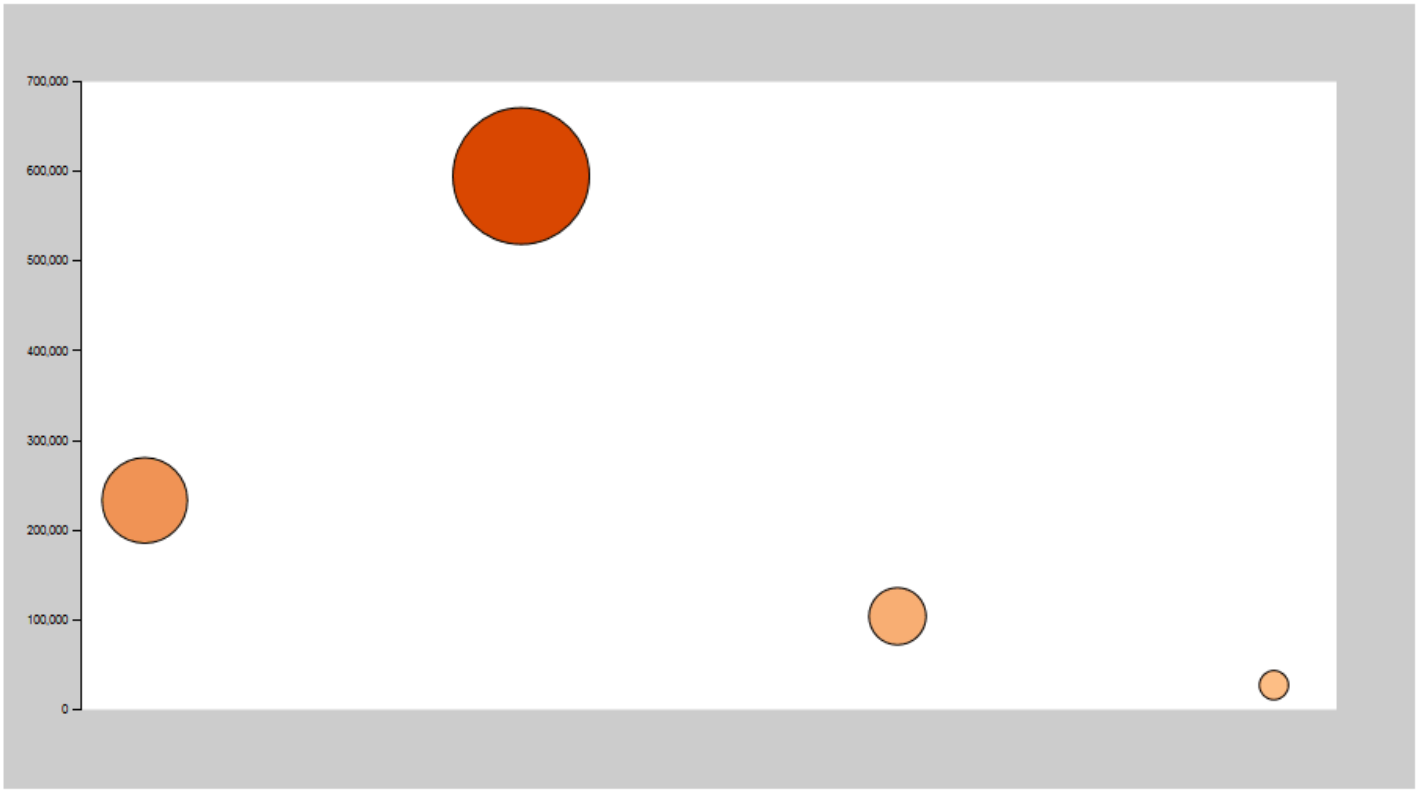

Figure 3.8: a very nice axis indeed

> **Practice!** Create a vertical axis for your chart.

3.4 Text

Our bubble chart is almost complete. It could use a title, though. Since SVG is mainly for drawing, most SVG elements do not support text content. An exception is the `<text>` element. We can add a block to append one (Example 3.12):

Example 3.12: adding a title to the chart

JavaScript

```
//below Example 3.9...create a text element and add the title
var title = container.append("text")
    .attr("class", "title")
    .attr("text-anchor", "middle")
    .attr("x", 450)
    .attr("y", 30)
    .text("City Populations");
```

In Example 3.12, the `"text-anchor"` attribute (line 4) center-justifies the text in the element, while the `x` and `y` attributes (lines 5 and 6) position the text anchor within the SVG container. The `.text()` operator (line 7) adds the text content. Now all we need to do is style our title a bit in *style.css* (Example 3.13):

Example 3.13: styling the title in *style.css*

CSS

```
.title {
    font-family: sans-serif;
    font-size: 1.5em;
    font-weight: bold;
}
```

We now have a nice title for our chart (Figure 3.9):

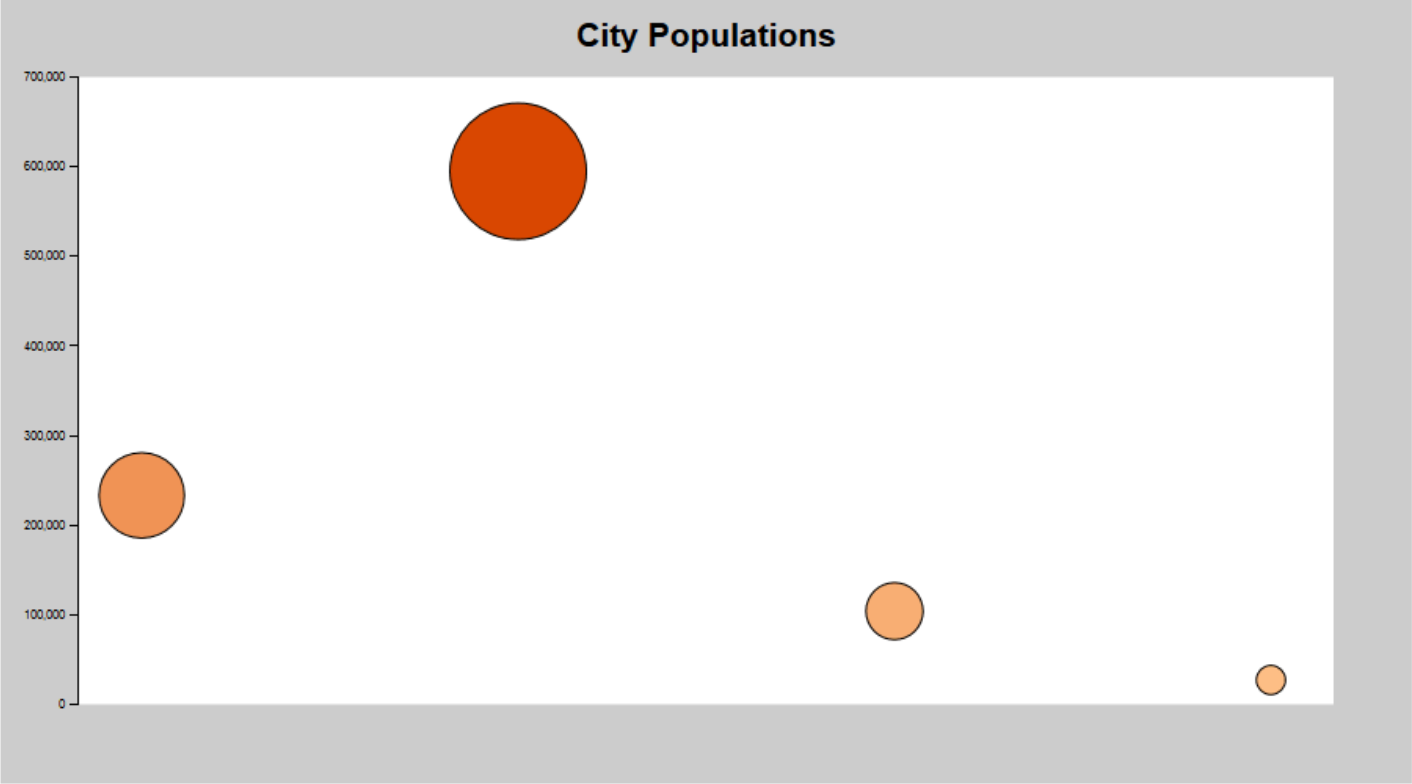

Figure 3.9: chart title

A final useful touch is to label each circle with its corresponding city name and population. We can also use `<text>` elements for this purpose. We will need to join our `cityPop` data to the selection to correctly position each label (Example 3.14):

Example 3.14: creating circle labels in *main.js*

JavaScript

```
    //below Example 3.12...create circle labels
    var labels = container.selectAll(".labels")
        .data(cityPop)
        .enter()
        .append("text")
        .attr("class", "labels")
        .attr("text-anchor", "left")
        .attr("x", function(d,i){
```

```
        //horizontal position to the right of each circle
        return x(i) + Math.sqrt(d.population * 0.01 / Math.PI) + 5;
    })
    .attr("y", function(d){
        //vertical position centered on each circle
        return y(d.population) + 5;
    })
    .text(function(d){
        return d.city + ", Pop. " + d.population;
    });
```

Result (Figure 3.10):

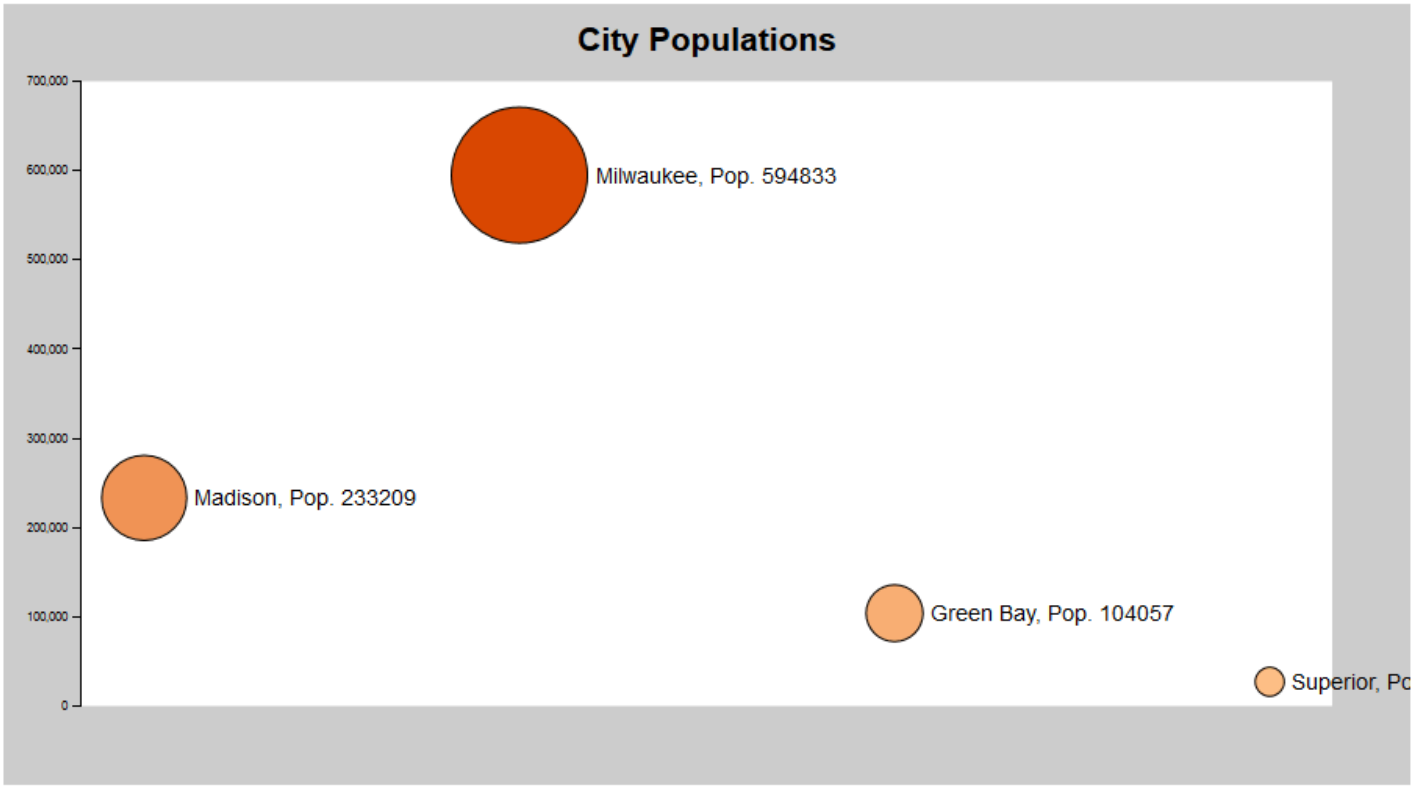

Figure 3.10: circle labels

This works quite well, except that the labels are a bit long. Superior's even overflows the container. It would be nice if we could wrap these labels onto two lines. Note that if you search for how to do this online, you are likely to be directed toward a certain **Mike Bostock tutorial (http://bl.ocks.org/mbostock/7555321)** that is quite complicated. It is much simpler to manually break lines using SVG `<tspan>` elements. These are child elements of a `<text>` element that can be independently positioned relative to their parent. We can create a separate `<tspan>` for each line of our label. Instead of setting our `"x"` attribute and text content in the `labels` block, we will instead move these to each of our separate `<tspan>` blocks to horizontally align them and give each one custom content (Example 3.15):

Example 3.15: creating `<tspan>` elements in *main.js*

JavaScript

```
//Example 3.14 line 1...create circle labels
var labels = container.selectAll(".labels")
    .data(cityPop)
    .enter()
    .append("text")
    .attr("class", "labels")
    .attr("text-anchor", "left")
    .attr("y", function(d){
        //vertical position centered on each circle
        return y(d.population) + 5;
    });

//first line of label
var nameLine = labels.append("tspan")
    .attr("class", "nameLine")
    .attr("x", function(d,i){
        //horizontal position to the right of each circle
        return x(i) + Math.sqrt(d.population * 0.01 / Math.PI) + 5;
    })
    .text(function(d){
        return d.city;
    });

//second line of label
var popLine = labels.append("tspan")
    .attr("class", "popLine")
    .attr("x", function(d,i){
        //horizontal position to the right of each circle
        return x(i) + Math.sqrt(d.population * 0.01 / Math.PI) + 5;
    })
    .text(function(d){
        return "Pop. " + d.population;
    });
```

Here is what we get (Figure 3.11):

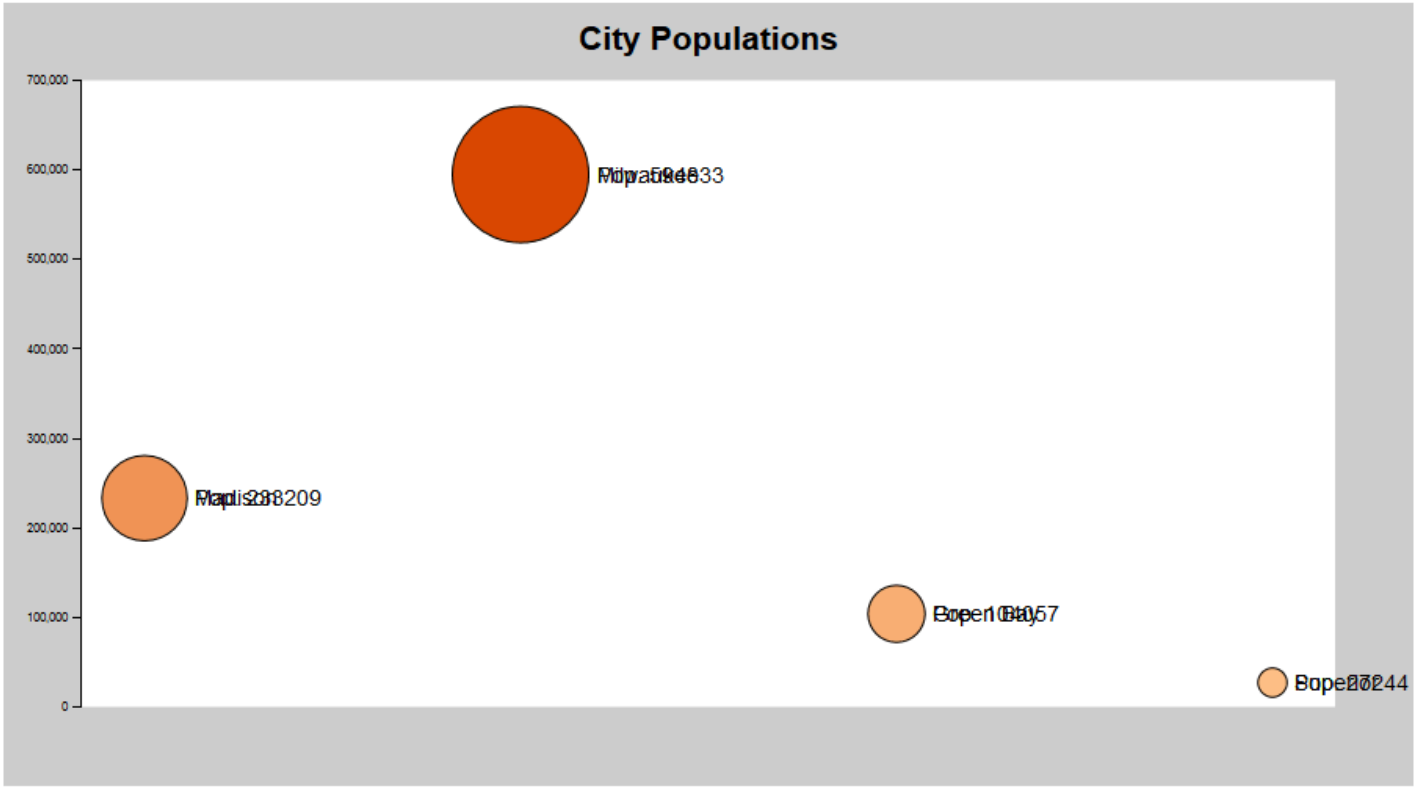

Figure 3.11: label mush

We now have separated the city name and population into two separate lines, but they are on top of each other! The solution is to offset the second line vertically from the first by adding a `"dy"` attribute to it (Example 3.16):

Example 3.16: offsetting the second line in *main.js*

JavaScript

```
//Example 3.15 line 24...second line of label
var popLine = labels.append("tspan")
    .attr("class", "popLine")
    .attr("x", function(d,i){
        return x(i) + Math.sqrt(d.population * 0.01 / Math.PI) + 5;
    })
    .attr("dy", "15") //vertical offset
    .text(function(d){
        return "Pop. " + d.population;
    });
```

With this adjustment, both lines of each label should be visible (Figure 3.12):

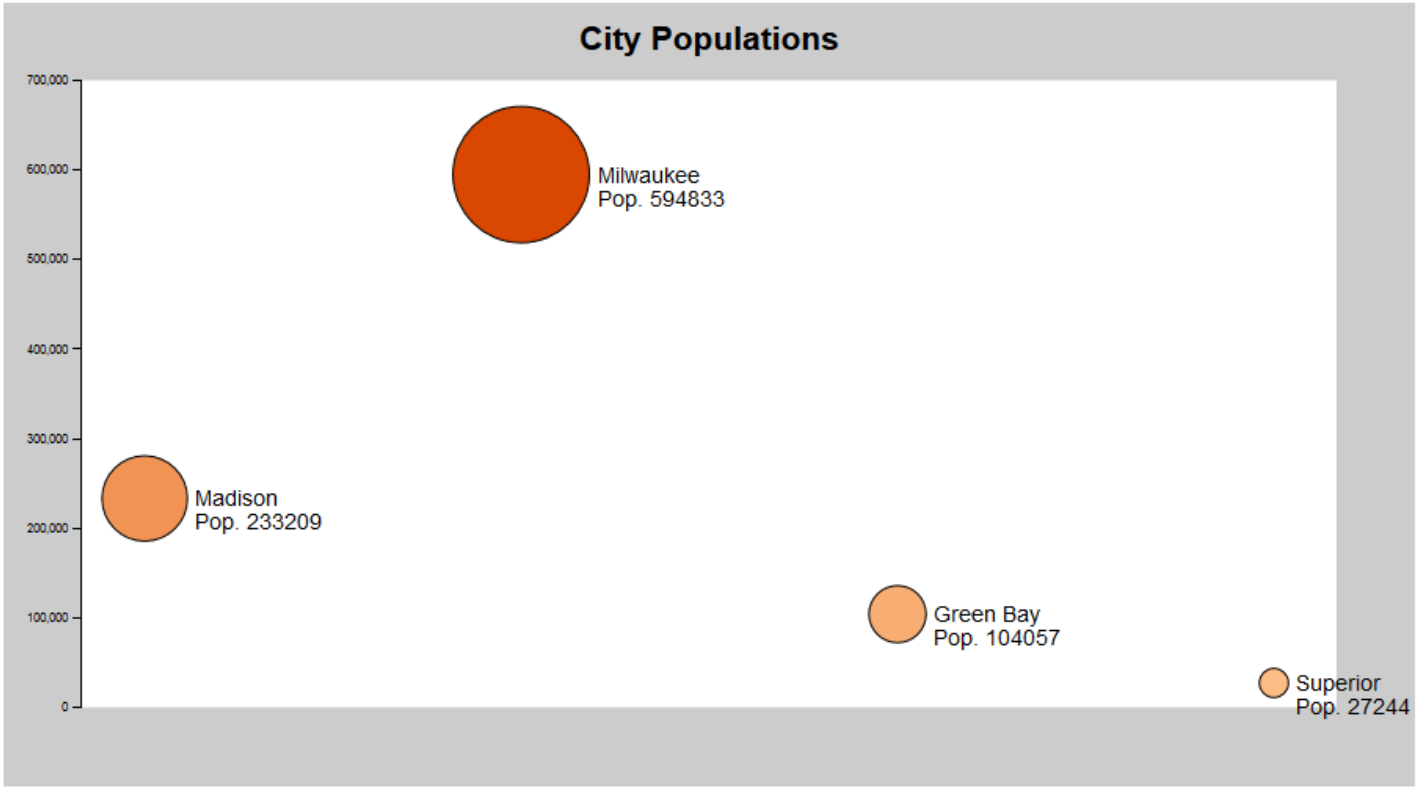

Figure 3.12: double-barrel labels

Now, if we are really finnicky, we can critique our population numbers for not being correctly formatted with commas (or periods in some countries) every three decimal places. Luckily, D3 provides a handy method, `d3.format()` (https://github.com/d3/d3-format/blob/master/README.md#format), for formatting numbers as text. This method takes a format specifier string modeled on Python's **format specifications** (https://docs.python.org/release/3.1.3/library/string.html#formatspec) as its parameter.
Calling `d3.format()` with a format specifier returns a generator function, which can then be passed a value to format (Example 3.17):

Example 3.17: formatting population numbers in *main.js*

JavaScript

```
//create format generator
var format = d3.format(",");

//Example 3.16 line 1...second line of label
var popLine = labels.append("tspan")
    .attr("class", "popLine")
    .attr("x", function(d,i){
        return x(i) + Math.sqrt(d.population * 0.01 / Math.PI) + 5;
    })
    .attr("dy", "15") //vertical offset
    .text(function(d){
        return "Pop. " + format(d.population); //use format generator to format numbers
    });
```

There are two other slight adjustments we will make to finish our chart:

- adjust the `"y"` attribute in the `labels` block (Example 3.15 line 10) to vertically center the entire label with each circle;
- adjust the maximum range value of our `x` scale (Example 3.1 line 3) to bring Superior's label entirely into the frame.

With these adjustments made, we have a complete, readable data graphic (Figure 3.13):

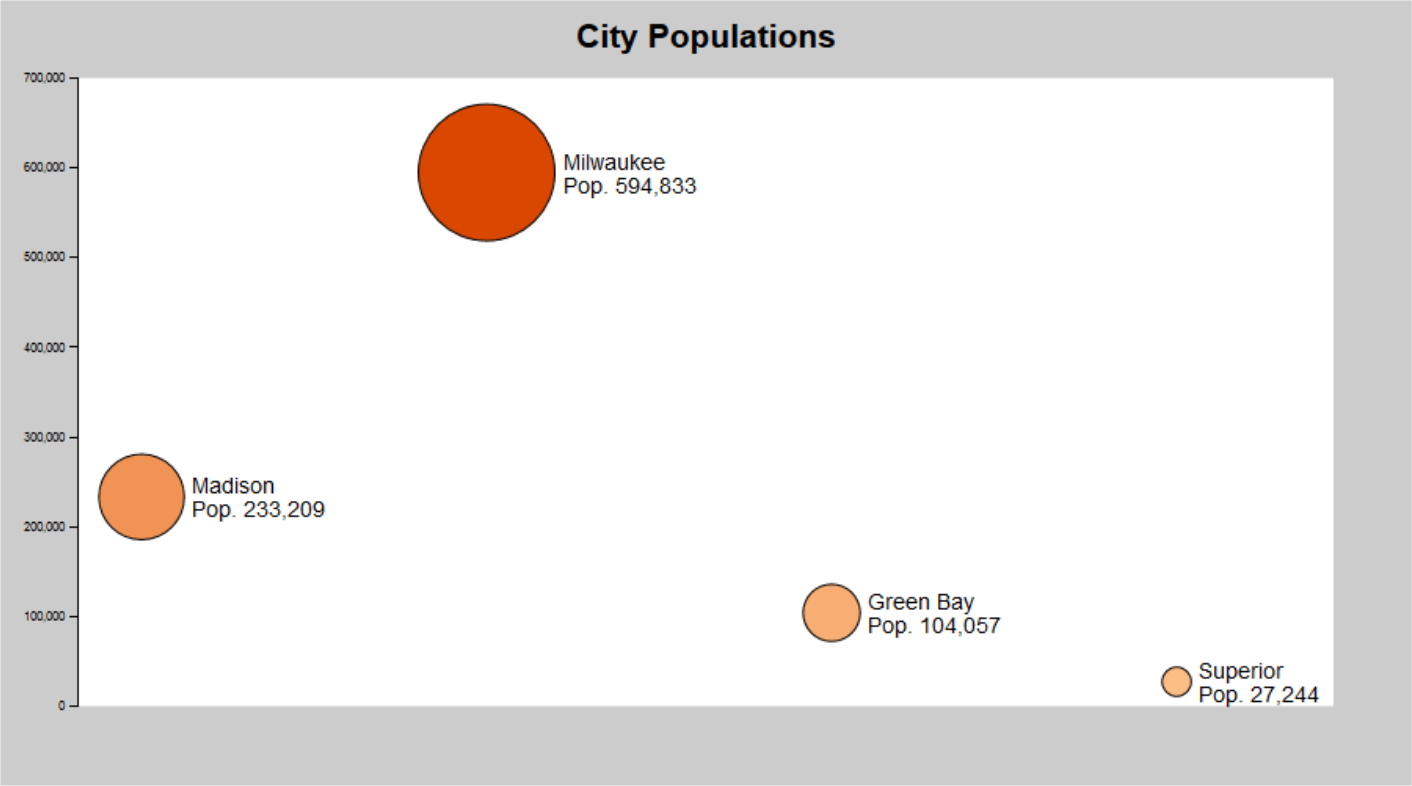

Figure 3.13: the finished bubble chart

> **Practice!** Add a title and correctly formatted labels to your bubble chart.

> **Self-Check:**
>
> **1. When should you begin a new code block instead of adding operators onto an existing one?**
>
> **a.** when creating a new selection
> **b**. when adding a *second* `.append()` operator to the block
> **c.** when the operator you are adding changes the operand so that it no longer matches the block name
> **d.** all of the above
>
> **2. True/False:** Each line in a code block should include a semicolon at the end of the line to properly separate the operators.

Module Reminder

1. Create a *d3-demo* web directory and Git repository. Sync this repository with GitHub.
2. Using D3, create an SVG bubble chart based on city population data for four cities (you must find your own data!). The circles on your chart should be sized, positioned, and colored proportionately the population data.
3. Add a vertical axis, title, and labels to your chart. Keep your code neat and include explanatory comments where appropriate. You will be graded on the neatness and legibility of your D3 code.
4. Be sure to include your dataset in the *data* folder of your repository and include in your commit. With the above tasks completed, commit changes to your *d3-demo* web directory and sync with GitHub.
5. Based on the instructions in the D3 Lab Activity, find and format a multivariate dataset of interest to you that you will use to complete the Lab.