Drawing with data – Scales

The Power of data()

This is exciting, but real-world data is never this clean:

```
var dataset = [ 5, 10, 15, 20, 25 ];
```

Let's make our data a bit messier, as in *04_power_of_data.html*:

```
var dataset = [ 25, 7, 5, 26, 11 ];
var dataset = [ 25, 7, 5, 26, 11, 8, 25, 14, 23, 19,
                14, 11, 22, 29, 11, 13, 12, 17, 18, 10,
                24, 18, 25, 9, 3 ];
```

Figure 6-6. New data values

Twenty-five data points instead of five

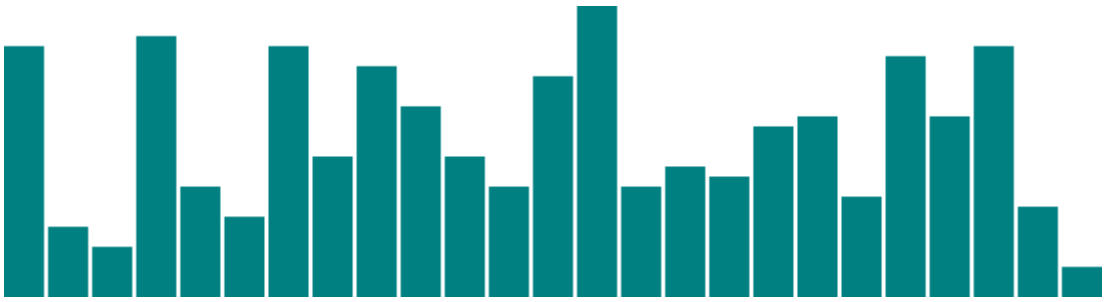


Figure 6-7. Lots more data values

How does D3 automatically expand our chart as needed?

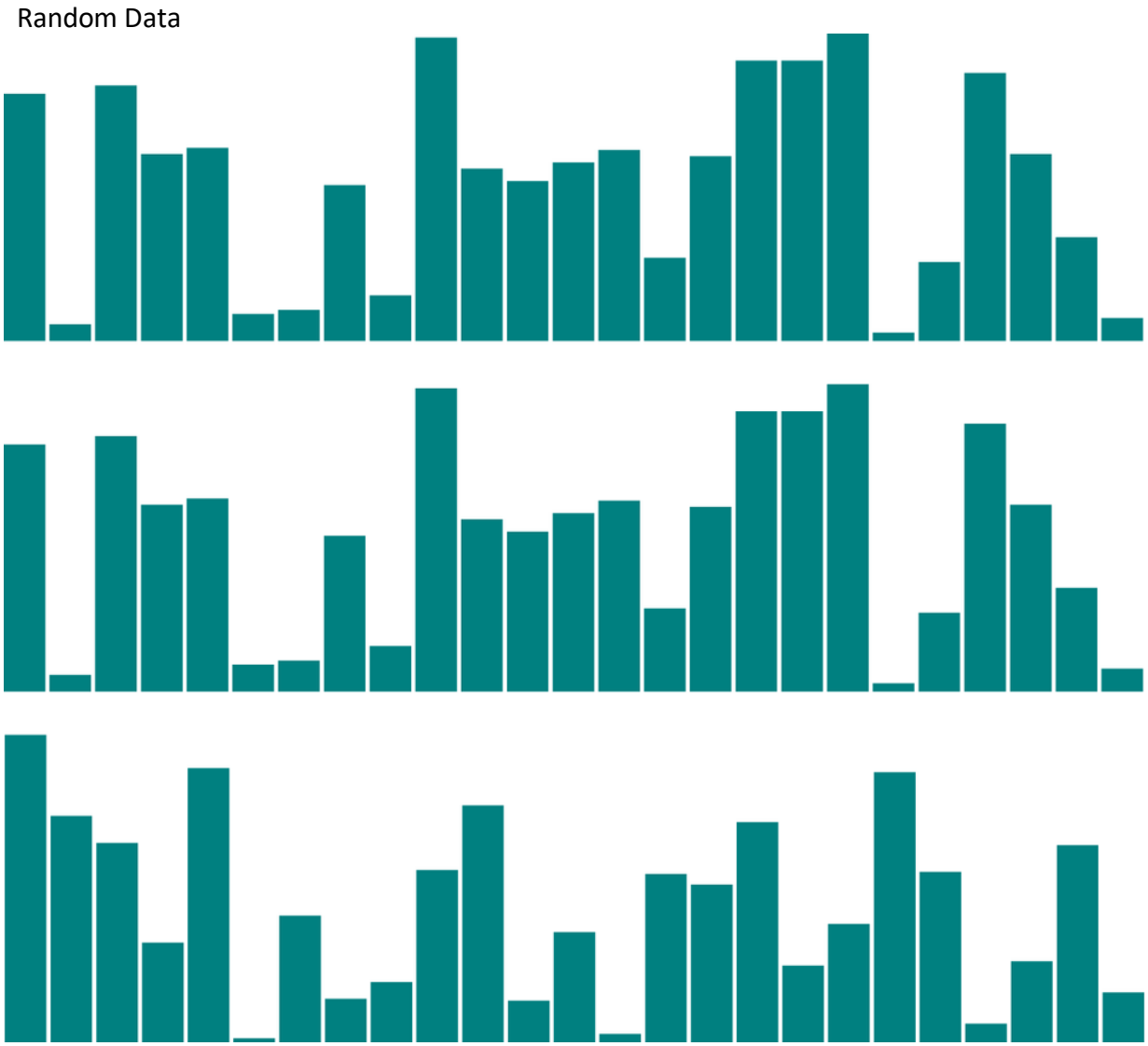
```
d3.select("body").selectAll("div")
    .data(dataset)  // <-- The answer is here!
    .enter()
    .append("div")
    .attr("class", "bar")
    .style("height", function(d) {
        var barHeight = d * 5;
        return barHeight + "px";
    });
```

Give `data()` 10 values, and it will loop through 10 times. Give it one million values, and it will loop through one million times. (Just be patient.)

That is the power of `data()`—being smart enough to loop through the full length of whatever dataset you throw at it, executing each method beneath it in the chain, while updating the context in which each method operates, so `d` always refers to the current datum at that point in the loop.

That might be a mouthful, and if it all doesn't make sense yet, it will soon. I encourage you to make a copy of *05_power_of_data_more_points.html*, tweak the `dataset` values, and note how the bar chart changes.

Remember, the *data* is driving the visualization—not the other way around.



Bar charts with random values

View the source, and you'll see this code:

```
var dataset = [];                           //Initialize empty array
for (var i = 0; i < 25; i++) {              //Loop 25 times
    var newNumber = Math.random() * 30;     //New random number (0-30)
    dataset.push(newNumber);                //Add new number to array
}
```

That code doesn't use any D3 methods; it's just JavaScript. Without going into too much detail, this code does the following:

1. Creates an empty array called `dataset`.
2. Initiates a `for` loop, which is executed 25 times.
3. Each time, it generates a new random number with a value between 0 and 30. (Well, technically, *almost* 30. `Math.random()` returns values as low as 0.0 all the way up to, but not including, 1.0. So if `Math.random()` returned 0.99999, then the result would be 0.99999 times 30, which is 29.9997, or the teensiest bit less than 30.)
4. That new number is appended to the `dataset` array. (`push()` is an array method that appends a new value to the end of an array.)

Just for kicks, open up the JavaScript console and enter `console.log(dataset)`. You should see the full array of 25 randomized data values, as shown

```
> console.log(dataset)
  [14.793717765714973, 21.65710132336244, 22.01914135599509, 10.693866850342602,
  8.197558452375233, 8.327909619547427, 9.349913026671857, 6.715130957309157,
  20.352523955516517, 20.892786516342312, 18.432767554186285, 7.062793713994324,
  11.519823116250336, 8.91862049465999, 5.422192756086588, 8.956057007890195,
  13.239774140529335, 24.165618284605443, 14.453229457139969, 27.792113937903196,
  2.717762708198279, 12.752952876035124, 1.7288982309401035, 21.01240729680285,
  26.07524922117591]
```

. Random values in console

Notice that they are all decimal or floating point values (such as 14.793717765714973), not whole numbers or integers (such as 14) like we used initially. For this example, decimal values are fine, but if you ever need whole numbers, you could use JavaScript's `Math.round()` or `Math.floor()` methods. `Math.round()` rounds any number to the nearest integer, whereas `Math.floor()` always rounds down, for greater control over the result. For example, you could wrap the random number generator from this line:

```
var newNumber = Math.random() * 30;
```

as follows:

```
var newNumber = Math.floor(Math.random() * 30);
```

Using this code, `newNumber` would always be either 0 or 29, or any integer in between. Why not 30? Because `Math.random()` always returns values *less than* 1.0, and `Math.floor()` will always *round down*, so 29 is the highest possible return value.

Try it out in *07_power_of_data_rounded.html*, and use the console to verify that the numbers have indeed been rounded to integers, as displayed

```
> console.log(dataset)
  [23, 19, 18, 16, 24, 29, 1, 5, 13, 4, 29, 23, 11, 9, 16, 10, 15, 4, 28, 23, 13,
  19, 20, 20, 27]
```

Figure 6-10. Random integer values in console

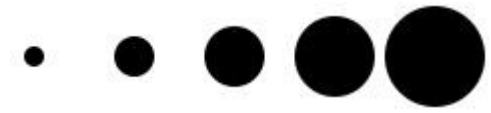
That's about all we can do visually with `div`s. Let's expand our visual possibilities with SVG.

To make it easy to reference all of the `circle`s later, we can create a new variable to store references to them all:

```
var circles = svg.selectAll("circle")
                 .data(dataset)
                 .enter()
                 .append("circle");
```

Great, but all these circles still need positions and sizes, displayed in Figure 6-11. Be warned, the following code might blow your mind:

```
circles.attr("cx", function(d, i) {
            return (i * 50) + 25;
        })
       .attr("cy", h/2)
       .attr("r", function(d) {
            return d;
       });
```



Row of data circles