Data

Data is an extremely broad term, only slightly less vague than the nearly all-encompassing *information*. What is data? (What *isn't* data?) What kinds of data are there, and what can we use with D3?

Broadly speaking, data is structured information with potential for meaning.

In the context of programming for visualization, data is stored in a digital file, typically in either text or binary form. Of course, potentially every piece of digital emphemera may be considered "data" — not just text, but bits and bytes representing images, audio, video, databases, streams, models, archives, and anything else.

Within the scope of D3 and browser-based visualization, however, we will limit ourselves to *text-based data*. That is, anything that can be represented as numbers and strings of alpha characters. If you can get your data into a `.txt` plain text file, a `.csv` comma-separated value file, or a `.json` JSON document, then you can use it with D3.

Whatever your data, it can't be made useful and visual until it is *attached* to something. In D3 lingo, the data must be *bound* to elements within the page. Let's address how to create new page elements first. Then attaching data to those elements will be a cinch.

Generating Page Elements

Typically, when using D3 to generate new DOM elements, the new elements will be circles, rectangles, or other visual forms that represent your data. But to avoid confusing matters, we'll start with a simple example and create a lowly `p` paragraph element.

Begin by creating a new document with our simple HTML template from the last chapter:

```
<!DOCTYPE html>
<html lang="en">
```

```
<head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <script type="text/javascript" src="d3/d3.v2.js"></script>
</head>
<body>
    <script type="text/javascript">
        // Your beautiful D3 code will go here
    </script>
</body>
</html>
```

Open that page up in your web browser, and pop open the web inspector. You should see something like:

Figure 5-1. Web inspector

Back in your text editor, replace the comment between the `script` tags with:

```
d3.select("body").append("p").text("New paragraph!");
```

Save and refresh, and voilà! There is text in the formerly empty browser window, and the following in the web inspector:

Figure 5-2. Web inspector, again

See the difference? Now in the DOM, there is a new paragraph element that was generated on-the-fly! This may not be exciting yet, but you will soon use a similar technique to dynamically generate tens or hundreds of elements, each one corresponding to a piece of your data set.

Let's walk through what just just happened. To understand that first line of D3 code, you must first meet your new best friend, *chain syntax*.

Chaining Methods

D3 smartly employs a technique called *chain syntax*, which you may recognize from jQuery. By "chaining" methods together with periods, you can perform several actions in a single line of code. It can be fast and easy, but it's important to understand how it works, to save yourself hours of debugging headaches later.

By the way, *functions* and *methods* are just two different words for the same concept: a chunk of code that accepts an argument as input, performs some action, and returns some other information as output.

This code

```
d3.select("body").append("p").text("New paragraph!");
```

may look like a big mess, especially if you're new to programming. So the first thing to know is that JavaScript, like HTML, doesn't care about whitespace and line breaks, so you can put each method on its own line for legibility:

```
d3.select("body")
    .append("p")
    .text("New paragraph!");
```

Both I and your optometrist highly recommend putting each method on its own indented line. But everyone has their own coding style; use whatever indents, line breaks, and whitespace (tabs or spaces) are most legible for you.

One Link at a Time

Let's deconstruct each line in this chain of code.

d3 — References the D3 object, so we can access its methods. Our D3 adventure begins here.

.select("body") — Give the select() method a CSS selector as input, and it will return a reference to the first element in the DOM that matches. (Use selectAll() when you need more than one element.) In this case, we just want the body of the document, so a reference to body is handed off to the next method in our chain.

.append("p") — append() creates whatever new DOM element you specify and appends it to the end (but *just inside*) of whatever selection it's acting on. In our case, we

want to create a new `p` within the `body`. We specified `"p"` as the input argument, but this method also sees the reference to `body` that was passed down the chain from the `se lect()` method. So an empty `p` paragraph is *appended* to the `body`. Finally, `append()` hands off a reference to the new element it just created.

`.text("New paragraph!")` — `text()` takes a string and inserts it between the opening and closing tags of the current selection. Since the previous method passed down a reference to our new `p`, this code just inserts the new text between `<p>` and `</p>`. (In cases where there is existing content, it will be overwritten.)

`;` — The all-important semicolon indicates the end of this line of code. Chain over.

The Hand-off

Many, but not all, D3 methods return a selection (or, really, reference to a selection), which enables this handy technique of method chaining. Typically, a method returns a reference to the element that it just acted upon, but not always.

So remember: When chaining methods, order matters. The output type of one method has to match the input type expected by the next method in the chain. If adjacent inputs and outputs are mismatched, the hand-off will function more like a dropped baton in a middle-school relay race.

When sussing out what each function expects and returns, the API reference is your friend. It contains detailed information on each method, including whether or not it returns a selection.

Going Chainless

Our sample code could be rewritten without chain syntax as:

```
var body = d3.select("body");
var p = body.append("p");
p.text("New paragraph!");
```

Ugh! What a mess. Yet there will be times you need to break the chain, such as when you are calling so many functions that it doesn't make sense to string them all together. Or just because you want to organize your code in a way that makes more sense to you.

Now that you know how to generate new page elements with D3, it's time to attach data to them.

Binding data

What is binding, and why would I want to do it to my data?

Data visualization is a process of *mapping* data to visuals. Data in, visual properties out. Maybe bigger numbers make taller bars, or special categories trigger brighter colors. The mapping rules are up to you.

With D3, we *bind* our data input values to elements in the DOM. Binding is like "attaching" or associating data to specific elements, so that later you can reference those values to apply mapping rules. Without the binding step, we have a bunch of data-less, un-mappable DOM elements. No one wants that.

In a Bind

We use D3's `selection.data()` method to bind data to DOM elements. But there are two things we need in place first, before we can bind data:

1. The data

2. A selection of DOM elements

Let's tackle these one at a time.

Data

D3 is smart about handling different kinds of data, so it will accept practically any array of numbers, strings, or objects (themselves containing other arrays or key/value pairs). It can handle JSON (and GeoJSON) gracefully, and even has a built-in method to help you load in CSV files.

But to keep things simple, for now we will start with a boring array of five numbers. Here is our sample data set:

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

If you're feeling adventurous, or already have some data in CSV or JSON format that you want to play with, here's how to do that. Otherwise, just skip ahead to "Please Make Your Selection."

Loading CSV Data

CSV stands for comma-separated values. A CSV data file might look something like this:

```
Food,Deliciousness
Apples,9
Green Beans,5
Egg Salad Sandwich,4
Cookies,10
Vegemite,0.2
Burrito,7
```

Each line in the file has the same number of values (two, in this case), and values are separated by a comma. The first line in the file often serves as a header, providing names for each of the "columns" of data.

If we saved the CSV data above into a file called `food.csv`, then we could load the file into D3 by using the `d3.csv()` method:

```
d3.csv("food.csv", function(data) {
    console.log(data);
});
```

`csv()` takes two arguments: a string representing the path of the CSV file to load in, and an anonymous function, to be used as a *callback function*. The callback function is "called" only *after* the CSV file has been loaded and parsed successfully. So if `food.csv` can't be found, or there is some other error, it will never be called.

When called, the anonymous function is handed the result of the CSV loading and parsing process; that is, the data. Here I'm naming it `data`, but this could be called whatever you like. You should use this callback function to do all the things you can only do *after* the data has been loaded. In the example above, we are just logging the value of the `data` array to the console, to verify it. (See `01_csv_loading_exam ple.html` in the example code.)

```
[▶ Object , ▶ Object , ▶ Object , ▶ Object , ▶ Object , ▶ Object ]
```

Figure 5-3. Array logged to console

You can see that `data` is an array (because of the hard brackets `[]` on either end) with six elements, each of which is an object. By toggling the disclosure triangles next to each object, we can see their values.

```
[▼ Object           , ▼ Object           , ▼ Object                    ,
    Deliciousness: "9"      Deliciousness: "5"       Deliciousness: "4"
    Food: "Apples"          Food: "Green Beans"      Food: "Egg Salad Sandwich"
  ▶ __proto__: Object     ▶ __proto__: Object      ▶ __proto__: Object
 ▼ Object           , ▼ Object           , ▼ Object                    ]
    Deliciousness: "10"     Deliciousness: "0.2"     Deliciousness: "7"
    Food: "Cookies"         Food: "Vegemite"         Food: "Burrito"
  ▶ __proto__: Object     ▶ __proto__: Object      ▶ __proto__: Object
```

Figure 5-4. Array elements expanded

Aha! Each object has both a `Food` property and a `Deliciousness` property, the values of which correspond to the values in our CSV! (There is also a third property, `__pro to__`, but that has to do with how JavaScript handles objects, and you can ignore it for now.) D3 has employed the first row of the CSV for property names, and subsequent rows for values. You may not realize it, but this just saved you a *lot* of time.

One more thing to note is that each value from the CSV is stored as a string, even the numbers. (You can tell because 9 is surrounded by quotation marks, as in `"9"` and not simply `9`.) This may cause unexpected behavior later, if you try to reference your data as a numeric value, but it is still typed as a string.

Verifying your data is a great use of the `csv()` callback function, but typically this is where you'd call other functions that construct the visualization, now that the data is available, as in:

```
var dataset;  //Declare global var

d3.csv("food.csv", function(data) {

    //Hand CSV data off to global var,
    //so it's accessible later.
    dataset = data;

    //Call some other functions that
    //generate your visualization, e.g.:
    generateVisualization();
    makeAwesomeCharts();
    makeEvenAwesomerCharts();
    thankAwardsCommittee();

});
```

One more tip: If you have *tab*-separated data in a TSV file, try the `d3.tsv()` method, which otherwise behaves exactly as above.

Loading JSON Data

We'll spend more time talking about JSON later, but for now all you need to know is that the `d3.json()` method works the same way as `csv()`. Load your JSON data in this way:

```
d3.json("waterfallVelocities.json", function(json) {
    console.log(json);  //Log output to console
});
```

Here I've named the parsed output `json`, but it could be called `data` or whatever you like.

Please Make Your Selection

The data is locked and loaded. As a reminder, we are working with this array:

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

Now you need to decide what to select. That is, what elements will your data be associated with? Again, let's keep it super simple and say that we want to make a new paragraph for each value in the data set. So you might imagine something like this would be helpful

```
d3.select("body").selectAll("p")
```

and you'd be right, but there's a catch: The paragraphs we want to select *don't exist yet*. And this gets at one of the most common points of confusion with D3: How can we select elements that don't yet exist? Bear with me, as the answer may require bending your mind a bit.

The answer lies with `enter()`, a truly magical method. See this code, which I'll explain:

```
d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

View the example code `02_creating_paragraphs.html` and you should see five new paragraphs, each with the same content, like so:
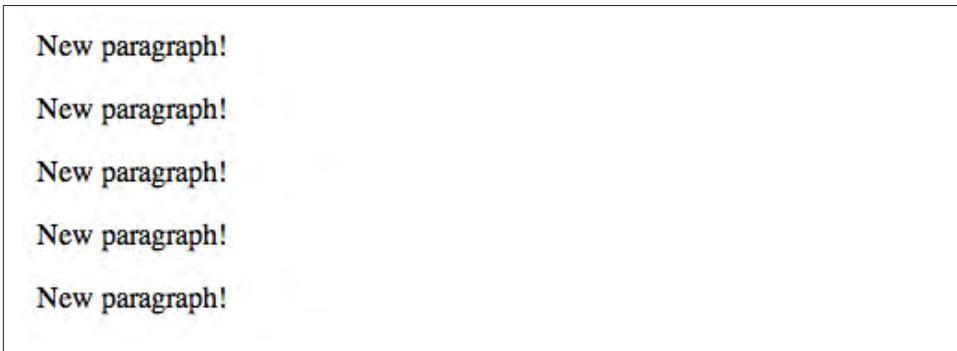

Figure 5-5. Dynamic paragraphs

Here's what's happening.

`d3.select("body")` — Finds the `body` in the DOM and hands off a reference to the next step in the chain.

`.selectAll("p")` — Selects all paragraphs in the DOM. Since none exist yet, this returns an empty selection. Think of this empty selection as representing the paragraphs that *will soon exist*.

`.data(dataset)` — Counts and parses our data values. There are five values in our array called `dataset`, so everything past this point is executed five times, once for each value.

`.enter()` — To create new, data-bound elements, you must use `enter()`. This method looks at the current DOM selection, and then at the data being handed to it. If there are more data values than corresponding DOM elements, then `enter()` *creates a new place-holder element* on which you may work your magic. It then hands off a reference to this new placeholder to the next step in the chain.

`.append("p")` — Takes the empty placeholder selection created by `enter()` and appends a `p` element into the DOM. Hooray! Then it hands off a reference to the element it just created to the next step in the chain.

`.text("New paragraph!")` — Takes the reference to the newly created `p` and inserts a text value.

Bound and Determined

All right! Our data has been read, parsed, and bound to new `p` elements that we created in the DOM. Don't believe me? Take another look at `02_creating_paragraphs.html` and whip out your web inspector.
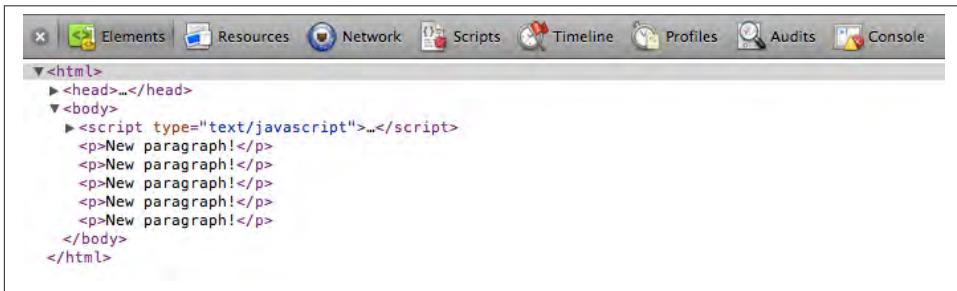

Figure 5-6. New `p` elements in the web inspector

Okay, I see five paragraphs, but where's the data? Switch to the JavaScript console, type in the following code, and hit enter:
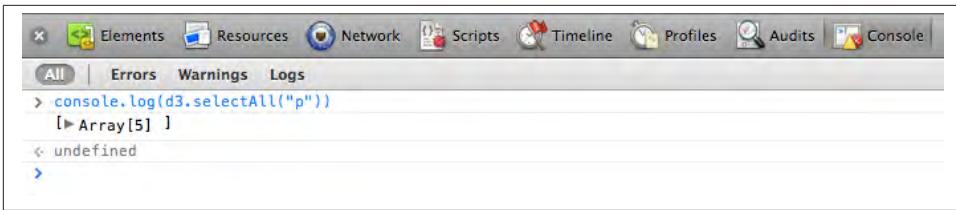
```
console.log(d3.selectAll("p"))
```

Figure 5-7. Logging to the console, from the console

An array! Or, really, an array containing another array. Click the gray disclosure triangle to reveal its contents:
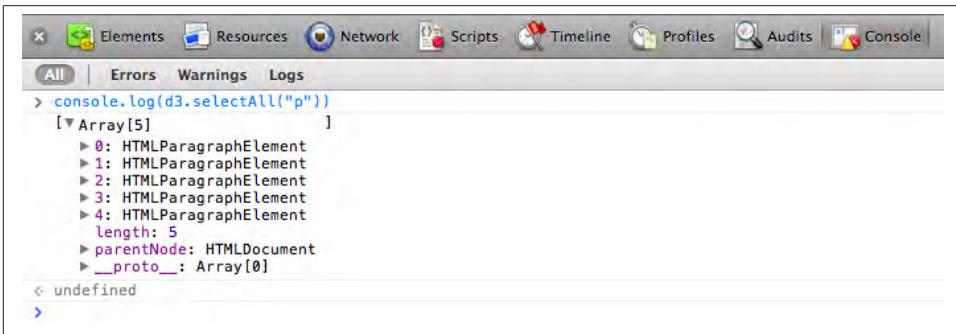

Figure 5-8. Array of arrays, expanded

You'll notice the five `HTMLParagraphElement`s, numbered 0 through 4. Click the disclosure triangle next to the first one (number zero).
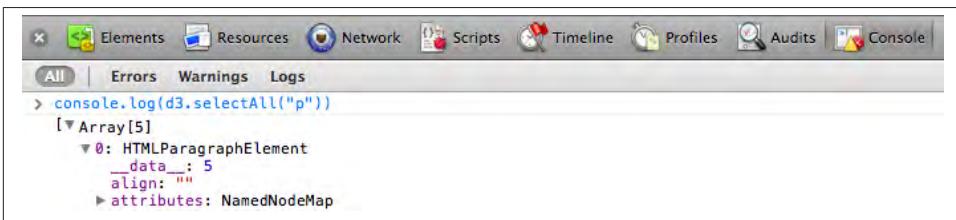

Figure 5-9. The p element, expanded

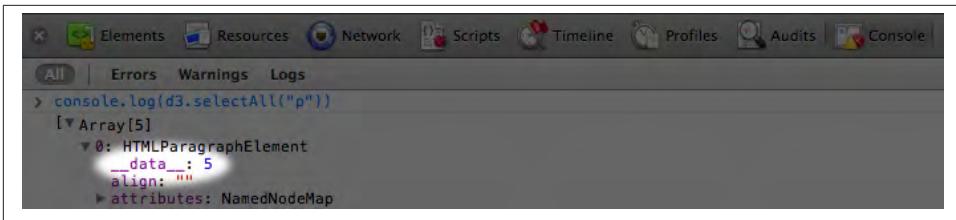See it? Do you see it? I can barely contain myself. There it is:

Figure 5-10. Finally, bound data

Our first data value, the number 5, is showing up under the first paragraph's `__data__` attribute. Click into the other paragraph elements, and you'll see they also contain `__data__` values: 10, 15, 20, and 25, just as we specified.

You see, when D3 binds data to an element, that data doesn't exist in the DOM, but it does exist in memory as a `__data__` attribute of that element. And the console is where you can go to confirm whether or not your data was bound as expected.

The data is ready. Let's do something with it.

Using your data

We can see that the data has been loaded into the page and is bound to our newly created elements in the DOM, but can we *use* it? Here's our code so far:

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

d3.select("body").selectAll("p")
    .data(dataset)
    .enter()
    .append("p")
    .text("New paragraph!");
```

Let's change the last line to:

```
    .text(function(d) { return d; });
```

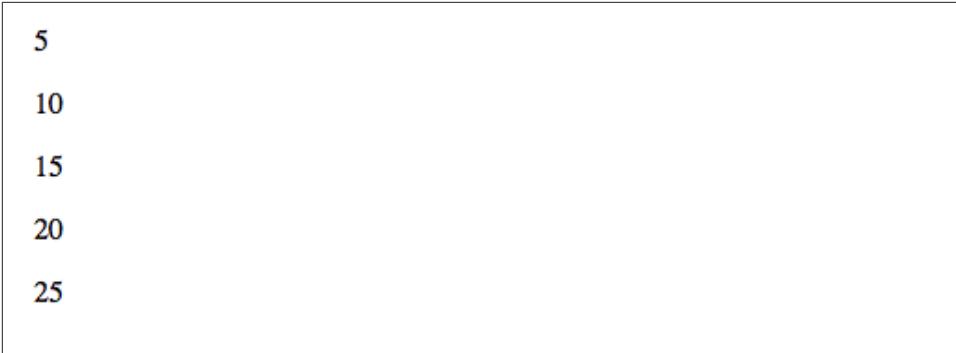Now test out that new code in `03_creating_paragraphs_text.html`. You should see:

```
5

10

15

20

25
```

Figure 5-11. More dynamic paragraphs

Whoa! We used our data to populate the contents of each paragraph, all thanks to the magic of the `data()` method. You see, when chaining methods together, anytime after you call `data()`, you can create an anonymous function that accepts `d` as input. The magical `data()` method ensures that `d` is set to the corresponding value in your original data set, given the current element at hand.

The value of "the current element" changes over time as D3 loops through each element. For example, when looping through the third time, our code creates the third `p` tag, and `d` will correspond to the third value in our data set (or `dataset[2]`). So the third paragraph gets text content of "15".

High-functioning

In case you're new to writing your own functions (a.k.a. methods), the basic structure of a function definition is:

```
function(input_value) {
    //Calculate something here
    return output_value;
}
```

The function we used above is dead simple, nothing fancy

```
function(d) {
    return d;
}
```

and it's wrapped within D3's `text()` function, so whatever our function returns is handed off to `text()`.

```
.text(function(d) {
    return d;
});
```

Our anonymous function is executed first. Then its result is handed off to `text()`. Then `text()` finally works its magic (by inserting its input argument as text within the selected DOM element).

But we can (and will) get much fancier, because you can customize these functions however you want. Yes, this is both the pleasure and pain of writing your own JavaScript. We can define our own custom functions however we want. Maybe you'd like to add some extra text, as in

```
.text(function(d) {
    return "I can count up to " + d;
});
```

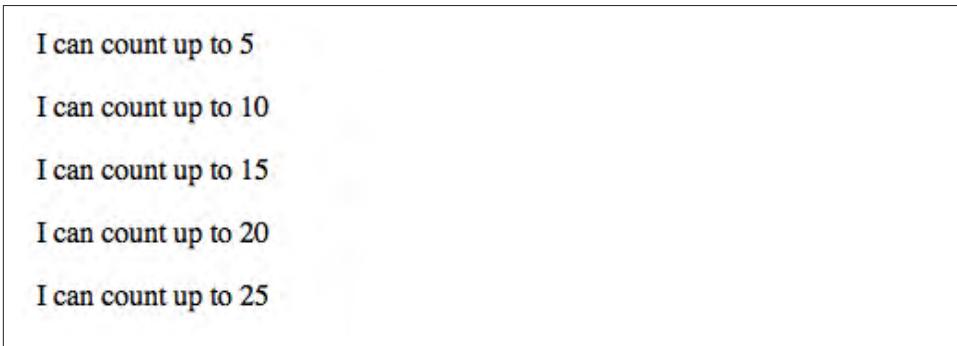which produces this result, as seen in example file `04_creating_paragraphs_count ing.html`:

I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25

Figure 5-12. Still more dynamic paragraphs

Data Wants to be Held

You may be wondering why you have to write out `function(d) { … }` instead of just `d` on its own. For example, this won't work:

```
.text("I can count up to " + d);
```

In this context, without wrapping `d` in an anonymous function, `d` has no value. Think of `d` as a lonely little placeholder value that just needs a warm, containing hug from a kind, caring function's parantheses. (Extending this metaphor further, yes, it is creepy that the hug is being given by an *anonymous* function — stranger danger! — but that only confuses matters.)

Here is `d` being gently and appropriately held by a function:

```
.text(function(d) {  // <-- Note tender embrace at left
    return "I can count up to " + d;
});
```

The reason for this syntax is that `.text()`, `attr()`, and many other D3 methods take a *function* as an argument. For example, `text()` can take either simply a static string of text as an argument *or* a function. Both of these are valid ways to use `text()`:

```
.text("someString")
.text(someFunction())
```

When you write

```
.text(function(d) {
    return d;
})
```

you are defining an anonymous function. If D3 sees a function there, it will *call* that function, while handing off the current datum `d` as the function's argument. Without the function in place, D3 can't know to whom it should hand off the argument `d`. When no one is there to receive the value of `d`, D3 may get confused and even start crying. (D3 is more emotional than you'd expect.)

At first, this may seem silly and like a lot of extra work to just get at `d`, but the value of this approach will become clear as we work on more complex pieces.

Beyond Text

Things get a lot more interesting when we explore D3's other methods, like `attr()` and `style()`, which allow us to set HTML attributes and CSS properties on selections, respectively.

For example, adding one more line to our code

```
.style("color", "red");
```

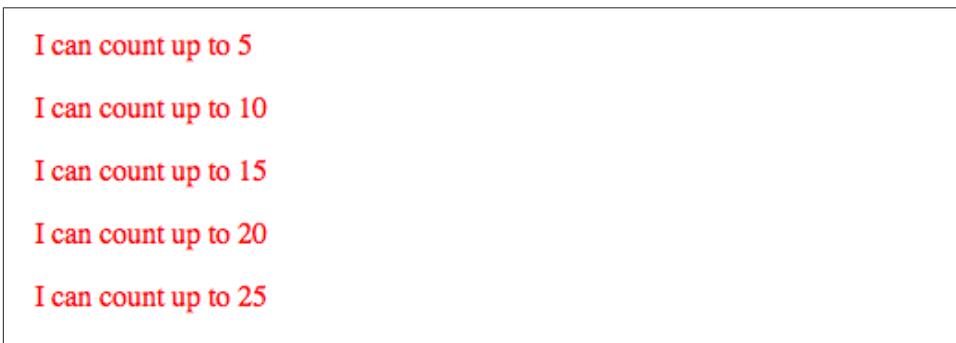produces this result, as seen in `05_creating_paragraphs_with_style.html`:

I can count up to 5

I can count up to 10

I can count up to 15

I can count up to 20

I can count up to 25

Figure 5-13. Red paragraphs

All the text is now red; big deal. But we could use a custom function to make the text red only if the current datum exceeds a certain threshold. So we revise that last line to use a function instead of a string:

```
.style("color", function(d) {
    if (d > 15) {    //Threshold of 15
        return "red";
    } else {
        return "black";
    }
});
```

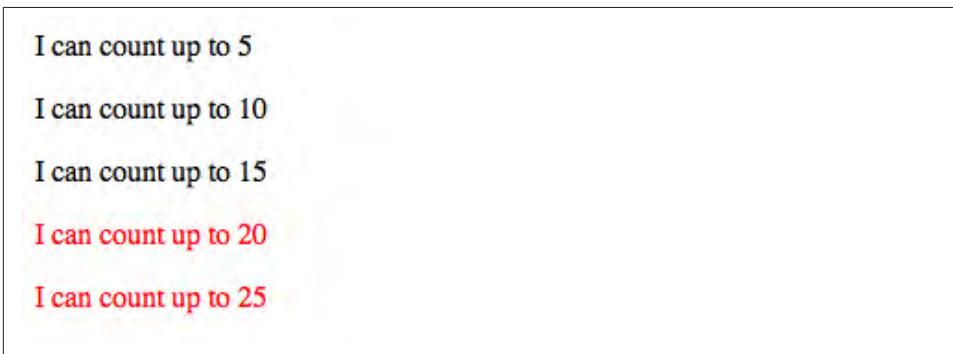See the resulting change in `06_creating_paragraphs_with_style_functions.html`:

Figure 5-14. Dynamically styled paragraphs

Notice how the first three lines are black, but once d exceeds the arbitrary threshold of 15, the text turns red.

Okay, we've got data loaded in and dynamically DOM elements bound to that data. I'd say we're ready to start drawing with data!