
Bear Hacks Data Driven Visualizations Tutorial

Introduction

This tutorial will teach you the basics of creating (hopefully) beautiful webpages using data driven documents (D3.js). D3.js is a very large and powerful library with tons of features and a complicated API. However, in this tutorial, we are not going to be diving into all of it at once. Instead, we will be looking at a core subset of the features: creating HTML documents that are generated by and respond to a dataset.

Simple Text

You might already be familiar with displaying text in a simple HTML:

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
  </head>
  <body>
    <div id="container">
      <p>Hello world!</p>
    </div>
  </body>
</html>
```

With D3.js, much like jQuery or many other JavaScript libraries you may have encountered before, you are able to add elements to the document using JavaScript. To create the same “Hello world!” webpage using D3.js, we can simply include the container in HTML:

index.html

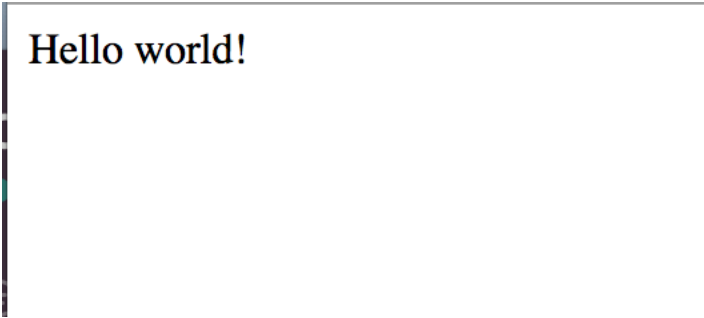
```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <script src="http://d3js.org/d3.v3.min.js"></script>
    <script src="index.js"></script>
  </head>
  <body>
    <div id="container"></div>
  </body>
</html>
```

and let JavaScript do all the heavy lifting:

index.js

```
window.onload = function() {
  d3.select('#container').append('p').text('Hello world!');
};
```

Open the .html file in your browser, and you should see something like this:



Hello world!

Code Explanation:

`window.onload` is the function that is called after the entire document has loaded. Because we are selecting elements we defined in the `body` of the HTML document, and `index.js` is included in `head`, we must wait until the entire document loads otherwise there will be nothing to select.

If you've used jQuery, the bit of code, `d3.select('#container')`, should look familiar. What `d3.select` does is query the document for the element that matches the selector. The selector is using the CSS selector format, so you can do things like `d3.select('div')` to select divs, `d3.select('.my-class')` to select elements of a certain class, or `d3.select('#element-id')`

to select an element with a specific ID.

Once we selected the element, in this case a div with ID of `container`, we can use the `.append` method to append child elements to the div. In this example we are adding a `<p>` tag inside the div.

Finally, we use `.text` to add text inside the `<p>` we just appended.

Data Driven

Up until now, it seems a bit pointless, and if anything more work, to use D3.js to populate the page with DOM elements. In the next example we see when you might want to let D3.js do the element insertion rather than manually specifying it inside the HTML document:

index.html: same as before

```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <script src="http://d3js.org/d3.v3.min.js"></script>
    <script src="index.js"></script>
  </head>
  <body>
    <div id="container"></div>
  </body>
</html>
```

index.js

```
window.onload = function() {

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

  d3.select('#container')
    .selectAll('p')
    .data(dataset)
    .enter()
    .append('p')
    .text(function(d) { return d; });

};
```

Run this in your browser, and you should see something like this:



Code Explanation:

This example introduces a few new API methods.

`.selectAll` works similarly to `.select`, except that it returns a list of all the elements that match the selector. In this example, it will return all the `<p>` elements within the `#container` div.

`.data` then binds `dataset` to the returned list of `<p>` elements. If there are more data than existing `<p>` (initially, there will be zero `<p>`'s), then the code after `.enter` will run each time a new item in `dataset` is read. In this example, each new data item will cause a new `<p>` to be appended to `#container`, and then populated with the value of `d`. This here is the key step to being *data driven*.

One more difference from before is that here, `.text` takes a function that returns a string as a parameter rather than just a string. What we're doing here is allowing the text we want to set to be dynamically calculated based on the data item the `<p>` is added for. D3.js will realize that it is a function, and that it is bound to `dataset`, so it will call the function on `d`, the next item in `dataset`, to get the text that should be set for data item `d`. In this case, we are simply returning the value of `d` (a number) as the string, so our `<p>`'s display 5, 10, ...

Simple Bar Graph

Let's start with our first somewhat interesting data visualization example: drawing a simple bar graph! To do this, all we need is to replace the `<p>`'s with rectangles (after all, bars are just rectangles). We can also style the rectangles a bit, so let's add some CSS, too.

index.css

```
div.bar {
  display: inline-block;
  width: 20px;
  height: 75px;
  background-color: teal;
  margin-left: 2px;
  margin-right: 2px;
}
```

Make sure we reference the CSS from `index.html`:

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <link type="text/css" rel="stylesheet" href="index.css"/>
    <script src="http://d3js.org/d3.v3.min.js"></script>
    <script src="index.js"></script>
  </head>
  <body>
    <div id="container"></div>
  </body>
</html>
```

Finally, make changes to `index.js`:

index.js

```
window.onload = function() {  
  
    var dataset = [5, 10, 15, 20, 25];  
  
    d3.select('#container')  
      .selectAll('div')  
      .data(dataset)  
      .enter()  
      .append('div')  
      .attr('class', 'bar')  
      .style('height', function(d) {  
        return d + 'px';  
      });  
  
};
```

Run in browser, you should see something like this:



Code Explanation:

The JavaScript in this example looks very similar to the previous example. Only difference is that instead of selecting and adding `<p>`'s, we are now adding divs. After all, divs are just rectangles, and we wanted rectangles. Then, instead of using `.text` to populate the `<p>`'s, we are using `.attr` to set the class of the div to `bar`, and using `.style` to set the height of the div to the height specified in `dataset`.

SVGs

While divs are acceptable for drawing rectangles, let's explore another technique used to draw more general shapes (and more consistently). It's called SVG, which is a type of vector graphics.

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
  </head>
  <body>
    <svg width="500" height="500">
      <circle cx="250" cy="50" r="25"
        fill="yellow" stroke="blue" stroke-width="10"/>
      <rect x="250" y="50" width="50" height="70"
        fill="rgba(128, 0, 128, 0.6)"/>
      <text x="100" y="100" font-family="sans-serif"
        font-size="25" fill="red">
        Hello world!
      </text>
    </svg>
  </body>
</html>
```

Expected result:



Code Explanation:

The `<svg>` tag defines a “canvas” to draw other svg objects onto. We must specify its width and height, in pixels. Then, we can add things like `circle`, `rect`, `text`, among others, onto the canvas. Each item can be defined by a set of attributes, such as defining a circle by the coordinates of its center and its radius. You can also specify the color, opacity, stroke, and fill of SVG elements. Feel free to look online for more fun things you can do with SVG!

Data Driven SVGs

Now that we know how to draw SVGs, we can also have D3.js draw SVGs for us depending on the bounded data set. The gist of it is to create and select a SVG canvas, then append SVG elements (such as `circle`) and change their attributes (`cx`, `cy`, `stroke`, etc.) to fit each data item you are trying to display.

index.html

```
<html>
  <head>
    <meta charset="utf-8">
    <title>D3 Test</title>
    <script src="http://d3js.org/d3.v3.min.js"></script>
    <script src="index.js"></script>
  </head>
  <body>
  </body>
</html>
```

index.js

```
window.onload = function() {

    var w = 500;
    var h = 500;

    var svg = d3.select('body')
        .append('svg')
        .attr('width', w)
        .attr('height', h);

    var dataset = [5, 10, 15, 20, 25];
    var colors = ['blue', 'red', 'green', 'yellow', 'purple'];

    svg.selectAll('circle')
        .data(dataset)
        .enter()
        .append('circle')
        .attr('cx', 100)
        .attr('cy', 100)
        .attr('r', function(d) {
            return d;
        })
        .attr('fill', function(d, i) {
            return colors[i];
        })
        .attr('opacity', 0.5);

};
```

Expected result:



Code Explanation:

This time, we moved even more stuff to JavaScript. In fact, there is nothing in the HTML body!

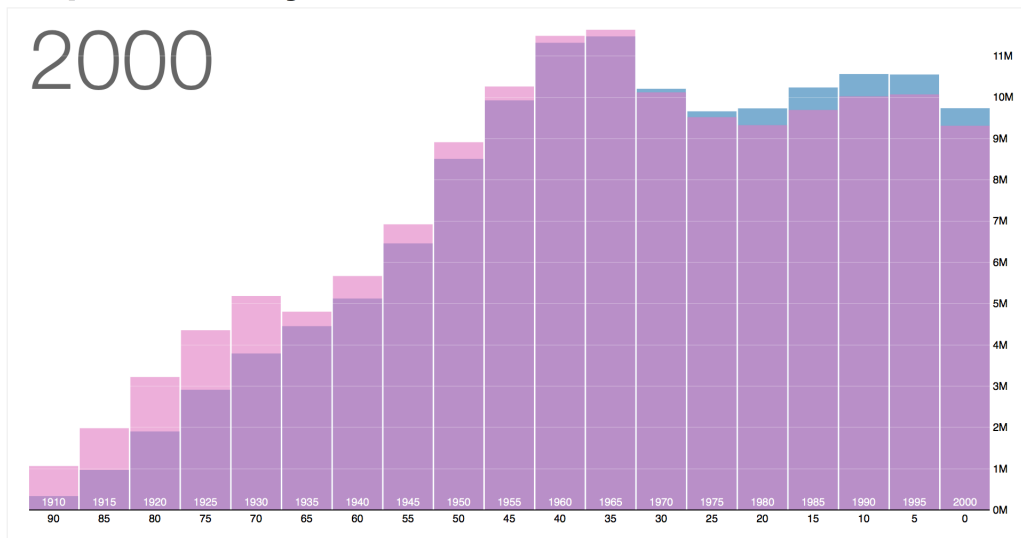
In `index.js`, we first add, select, and set the width and height of the `<svg>` canvas element. We store it in `var svg` since we'll be using it quite a bit, and it's less repetitive than calling `d3.select` each time. Try to understand the rest of the code yourself! If you get stuck, try to think of what we are trying to do here (the gist of the code is described in the beginning of this section).

Examples

Now that you understand the basics of D3.js, here are a few cool examples for inspiration! Try to look through the code and understand what they are doing. Feel free to ask any of the mentors if you have any question!

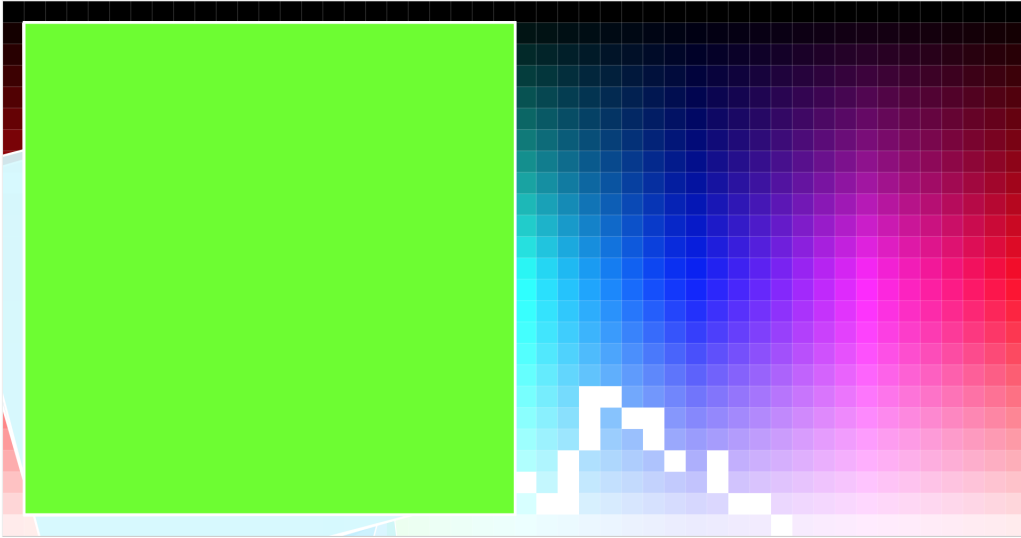
Population Pyramid

Population Pyramid



Transform Transitions

Transform Transitions

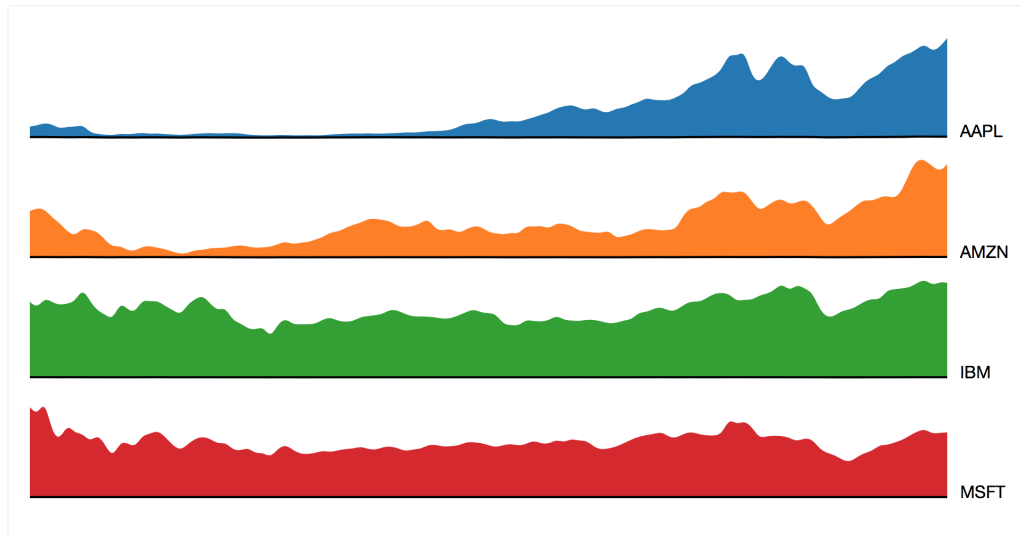


Visualizing Fisher-Yates Shuffle



Stock Prices Show Reel

D3 Show Reel



A lot of these also include animations, so make sure you click the link to experience the full demo! The demos also include example code, so if you see something you like, you can try to replicate that feature in your own project!

Of course, some of these examples (especially the show reel) use advanced features of D3.js not included in this tutorial. However, after understanding the basic motivation between D3, you can try to figure out what the more advanced code is doing!

Sample Project Ideas

Finally, here are a couple of sample project ideas. Feel free to use them as is, or as inspiration, or come up with your own completely different project!

1. Graph visualizer. Given a JSON of graph data, build a tool to visualize the graph. The JSON could specify exact node size/coordinates in addition to node-to-node connections, or even just node connections and leave spacing up to the graph visualizer. An additional challenge is to build a UI that allows users to add and delete nodes.
2. Visualizing a sorting algorithm: bar graph representing values of elements, then transitions to show where each element moves as the sorting algorithm runs. This would be similar to the Fisher-Yates shuffle visualization.