

# D3: Diving into the library

David Leonard

City College of New York

October 11, 2014

# Scales

*'Scales are functions that map from an input domain to an output range'*

- Mike Bostock

## Items and Pixels

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

- If 500 items are sold, corresponding bar would be 500px

## Items and Pixels

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

- ▶ If 500 items are sold, corresponding bar would be 500px
- ▶ What if this value changed to 600? 800?

## Items and Pixels

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

- ▶ If 500 items are sold, corresponding bar would be 500px
- ▶ What if this value changed to 600? 800?
- ▶ Requires bigger display to view bars

## Items and Pixels

```
var dataset = [ 100, 200, 300, 400, 500 ];
```

- ▶ If 500 items are sold, corresponding bar would be 500px
- ▶ What if this value changed to 600? 800?
- ▶ Requires bigger display to view bars
- ▶ How do we scale these values?

# Linear Scales

Linear scales is nothing more than normalization, in which we map a numeric value to a new value between 0 and 1, based on the possible minimum and maximum values. For example, 365 days in a year, day 310 maps to 0.85.

With linear scales, the input value is normalized according to the domain, and then the normalized value is scaled to the output range.

# Constructing a Scale

```
var scale = d3.scale.linear()  
              .domain([100, 500])  
              .range([10, 350]);  
  
scale(100); // Returns 10  
scale(300); // Returns 180  
scale(500); // Returns 350
```



## Other Scales

Apart from Linear Scales, D3 provides the following scales:

- ▶ sqrt

## Other Scales

Apart from Linear Scales, D3 provides the following scales:

- ▶ sqrt
- ▶ pow

## Other Scales

Apart from Linear Scales, D3 provides the following scales:

- ▶ sqrt
- ▶ pow
- ▶ log

## Other Scales

Apart from Linear Scales, D3 provides the following scales:

- ▶ sqrt
- ▶ pow
- ▶ log
- ▶ quantize

## Other Scales

Apart from Linear Scales, D3 provides the following scales:

- ▶ sqrt
- ▶ pow
- ▶ log
- ▶ quantize
- ▶ ordinal

# The SVG Element

D3 is most useful when generating and manipulating visuals such as SVG. SVG is more reliable, visually consistent and faster than drawing with divs.

- Can be included directly within any HTML document

# The SVG Element

D3 is most useful when generating and manipulating visuals such as SVG. SVG is more reliable, visually consistent and faster than drawing with divs.

- ▶ Can be included directly within any HTML document
- ▶ Supported by all web browsers except IE8 or higher

# SVG Shapes

► rect



# SVG Shapes

- ▶ rect
- ▶ circle

# SVG Shapes

- ▶ rect
- ▶ circle
- ▶ ellipse

# SVG Shapes

- ▶ rect
- ▶ circle
- ▶ ellipse
- ▶ line

# SVG Shapes

- ▶ rect
- ▶ circle
- ▶ ellipse
- ▶ line
- ▶ text

# SVG Shapes

- ▶ rect
- ▶ circle
- ▶ ellipse
- ▶ line
- ▶ text
- ▶ path

## rect

```
<rect x="0" y="0" width="500" height="50"/>
```



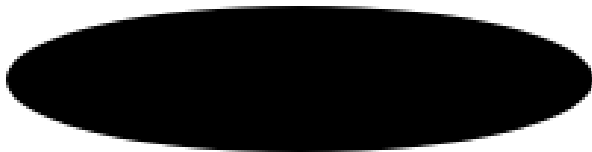
# circle

```
<circle cx="250" cy="25" r="25"/>
```



# ellipse

```
<ellipse cx="250" cy="25" rx="100" ry="25"/>
```





# line

```
<line x1="0" y1="0" x2="500" y2="50" stroke="black"/>
```



# Axes

D3 Axes are functions whose parameters we define. When called, it generates the visual elements of the axis, including lines, labels and ticks.

Axes are SVG-specific, as they generate SVG elements. They must be applied to either SVG or SVG **group** elements.

## SVG Groups

The **g** element within SVG stands for the *group* element. Group elements are invisible, but they allow us to:

- ▶ Contain / **group** elements together

## SVG Groups

The **g** element within SVG stands for the *group* element. Group elements are invisible, but they allow us to:

- ▶ Contain / **group** elements together
- ▶ We can apply transformations to these groups

## Constructing an axis function

```
var xAxis = d3.svg.axis()  
    .scale(xScale)  
    .orient("bottom")  
    .ticks(5);  
  
var yAxis = d3.svg.axis()  
    .scale(yScale)  
    .orient("left")  
    .ticks(5);
```

# Usage

```
svg.append("g")  
  .attr("class", "axis")  
  .attr("transform", "translate(0," + (h - padding) +  
    ")")  
  .call(xAxis);  
  
svg.append("g")  
  .attr("class", "axis")  
  .attr("transform", "translate(" + padding + ",0)")  
  .call(yAxis);
```

An SVG path can draw all sorts of shapes - rectangles, circles, ellipses, straight lines, curves and polygons.

The shape of an SVG Path element is defined by the attribute **d**, which contains the series of commands and parameters from within the SVG Path Mini-Language.

These commands are analogous to a set of instructions for *how to move a pen on paper*.

## D3: Diving into the library

### └ Diving into D3

#### └ SVG Paths

```
<svg width="100" height="100">  
  <path d=" M 10 25  
          L 10 75  
          L 60 75  
          L 10 25"  
        stroke="red" stroke-width="2" fill="none" />  
</svg>
```



- ▶ M 10 25: Put the pen down at (10, 25)

Note that SVG Path commands are case sensitive. **Capitalcase** means we are using *absolute positioning* based on the SVG viewing window, **lowercase** means we are using *relative positioning*.

- ▶ M 10 25: Put the pen down at (10, 25)
- ▶ L 10 75: Draw a line to the point (10, 75) from (10, 25)

Note that SVG Path commands are case sensitive. **Capitalcase** means we are using *absolute positioning* based on the SVG viewing window, **lowercase** means we are using *relative positioning*.

- ▶ M 10 25: Put the pen down at (10, 25)
- ▶ L 10 75: Draw a line to the point (10, 75) from (10, 25)
- ▶ L 60 75: Draw a line to the point (60, 75) from (10, 75)

Note that SVG Path commands are case sensitive. **Capitalcase** means we are using *absolute positioning* based on the SVG viewing window, **lowercase** means we are using *relative positioning*.

- ▶ M 10 25: Put the pen down at (10, 25)
- ▶ L 10 75: Draw a line to the point (10, 75) from (10, 25)
- ▶ L 60 75: Draw a line to the point (60, 75) from (10, 75)
- ▶ L 10 25: Draw a line to the point (10, 25) from (60, 75)

Note that SVG Path commands are case sensitive. **Capitalcase** means we are using *absolute positioning* based on the SVG viewing window, **lowercase** means we are using *relative positioning*.

## What are voronoi diagrams?

Voronoi diagrams are a method of dividing space into a set number of regions, based on some input points. For each point, there will be a region containing all points closest to the corresponding input point.

Voronoi diagrams are useful for creating invisible interactive regions, such as: <http://www.pointerpointer.com/>

# Voronoi function

```
// Voronoi generator function
var voronoi = d3.geom.voronoi()
  .x(function(d) { return d.LONGITUDE; })
  .y(function(d) { return d.LATITUDE; })
  .clipExtent([[0, 0], [width, height]]);
```

# Update

`selection.data()`: Joins an array of data to the current selection. Results in the *update* selection, which represents the selected DOM elements that were successfully bound to the specified data elements.

The *update* method also contains a reference to the *enter* and *exit* selection, used for adding and removing nodes in correspondence with the data.

# Enter

`selection.enter()`: Returns the enter selection - placeholder nodes for each data for which no corresponding existing DOM element was found. Supports the following operators:

- ▶ `append`



# Enter

`selection.enter()`: Returns the enter selection - placeholder nodes for each data for which no corresponding existing DOM element was found. Supports the following operators:

- ▶ `append`
- ▶ `insert`

# Enter

`selection.enter()`: Returns the enter selection - placeholder nodes for each data for which no corresponding existing DOM element was found. Supports the following operators:

- ▶ `append`
- ▶ `insert`
- ▶ `select`

# Enter

`selection.enter()`: Returns the enter selection - placeholder nodes for each data for which no corresponding existing DOM element was found. Supports the following operators:

- ▶ `append`
- ▶ `insert`
- ▶ `select`
- ▶ `call`

# Exit

`selection.exit()`: Contains existing DOM elements in the current selection for which no data element was found. Exposes the **remove** operator, which allows the removal of these elements.

# Example

```
d3.select("body").selectAll("div")  
  .data([4, 8, 15, 16, 23, 42])  
  .enter().append("div")  
    .text(function(d) { return d; });
```

## Example

```
var div = d3.select("body").selectAll("div")
    .data([1, 2, 4, 8, 16, 32], function(d) { return d; });

// Append new data
div.enter().append("div")
    .text(function(d) { return d; });

// Remove existing elements [15, 23, 42]:
div.exit().remove();
```

# Why callbacks?

Callbacks in JavaScript are a pattern which solve the problem of dealing with its asynchronous behavior. Functions in JavaScript are first-class objects, meaning that they can be passed around as arguments to other functions.

```
$("#btn_1").click(function() {  
    alert("Btn 1 Clicked");  
});
```

## D3 Resources

- ▶ Tutorials: <https://github.com/mbostock/d3/wiki/Tutorials>



## D3 Resources

- ▶ Tutorials: <https://github.com/mbostock/d3/wiki/Tutorials>
- ▶ API: <https://github.com/mbostock/d3/wiki/API-Reference>

## D3 Resources

- ▶ Tutorials: <https://github.com/mbostock/d3/wiki/Tutorials>
- ▶ API: <https://github.com/mbostock/d3/wiki/API-Reference>
- ▶ Stackoverflow