

This tutorial will get you to the point where you can make a chart- including scaling your data from an input domain to an output range, how to use scales, how to create axes, and how to make your components reusable through object oriented design.

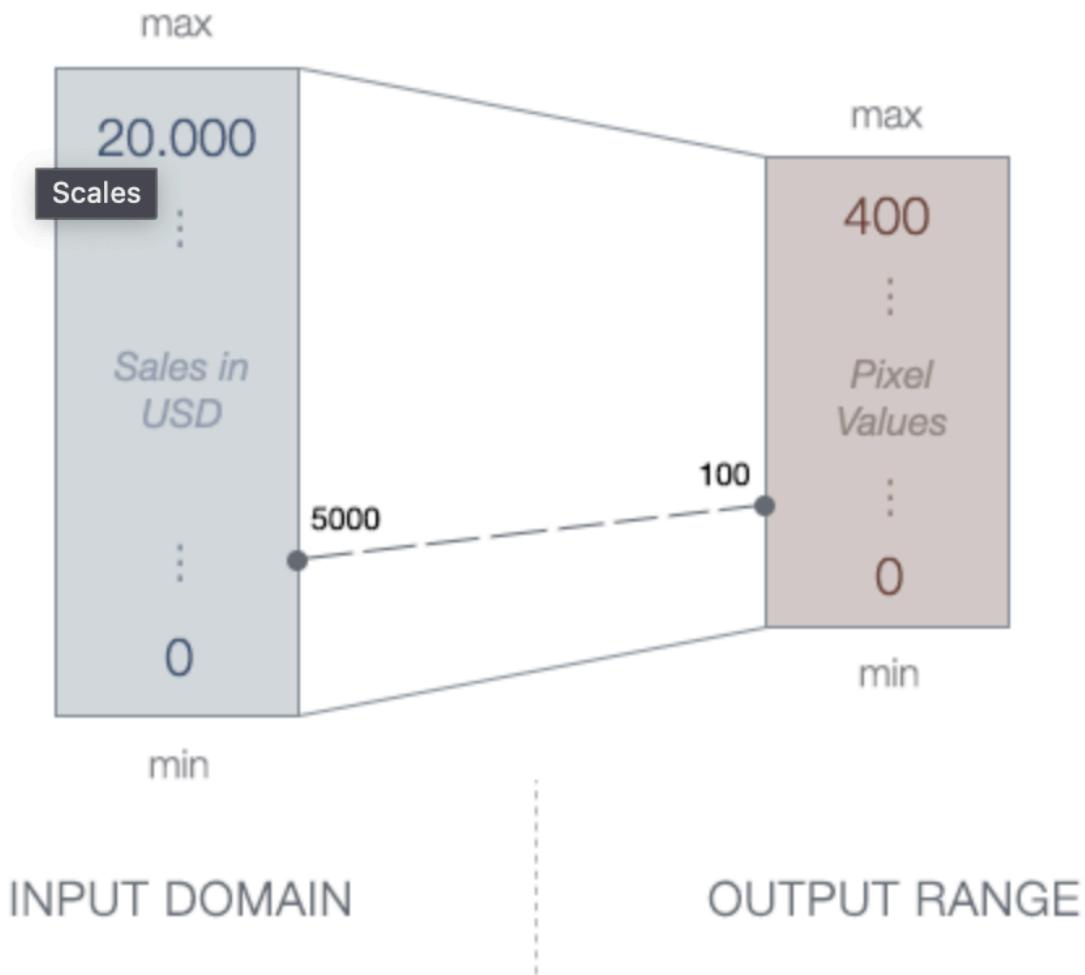
SVG Scales

Previously, we have used functions written in javascript that allow you to map data values (such as the cost of the natural disaster) to pixel values (like the radius of the circles in pixels). D3 provides ways to do this without writing your own mapping functions. These are called **scales**. **Scales allow you to take data (the input domain) and map it to an output range- pixels, color values, etc**). D3 scales can be linear, ordinal, logarithmic...

This is my favorite guide to d3 scales: <https://www.d3indepth.com/scales/>

More here: <https://github.com/d3/d3-scale/blob/master/README.md>

Example:



To do this in D3- map from the input domain to an output range- we can use scales.

```
// Creating a linear scale function
const iceCreamScale = d3.scaleLinear()
  .domain([0, 20000])
  .range([0, 400]);

// Call the function and pass an input value
iceCreamScale(5000); // Returns: 100
```

This example has pre-calculated values for the minimum value in the domain (0) and the maximum value in the domain (20000), and same with the range(min: 0, max: 400). D3 provides built-in functions for the min, max and extent - [min, max].

```
const quarterlyReport = [
  { month: 'May', sales: 6900 },
  { month: 'June', sales: 14240 },
  { month: 'July', sales: 25000 },
  { month: 'August', sales: 17500 }
];

// Returns the maximum value in a given array (= 25000)
const max = d3.max(quarterlyReport, d => d.sales);

// Returns the minimum value in a given array (= 6900)
const min = d3.min(quarterlyReport, d => d.sales);

// Returns the min. and max. value in a given array (= [6900,25000])
const extent = d3.extent(quarterlyReport, d => d.sales);
```

D3 Scale Types

D3 has 12 different scale types. These links do an excellent job walking through the different options.

[scales with continuous input and continuous output](#)

[scales with continuous input and discrete output](#)

[scales with discrete input and discrete output](#)

I'll mention a few scale types here:

D3 Linear Scales

One of the most common scale types. Suppose you have quantitative data (data you could do math on), such as a list of numbers.

D3 linear scales will take this data, from the input domain to the output range.

```
// Creating a linear scale function
```

```

const myLinearScale = d3.scaleLinear()
  .domain([0, 5000])
  .range([0, 100]);

// Call the function and pass an input value
myLinearScale(2500); // Returns: 50

myLinearScale(1250); // Returns: 25

```

D3 Time Scales

If it is a date, you can use a time scale, using JS date objects. Here are some examples: <https://observablehq.com/@d3/d3-scaletime>

D3 Banded Scales

What if your input domain from your data is not quantitative. What if your data is ordinal—an ordered list of values? Examples: day of the week values are typically ordered (Monday, Tuesday... Sunday), as are months of the year (January, February...). So are years in high school (Freshman, Sophomore, Junior, Senior). These are ordinal values. What if your data is nominal (e.g., College majors, Ice Cream Flavors).

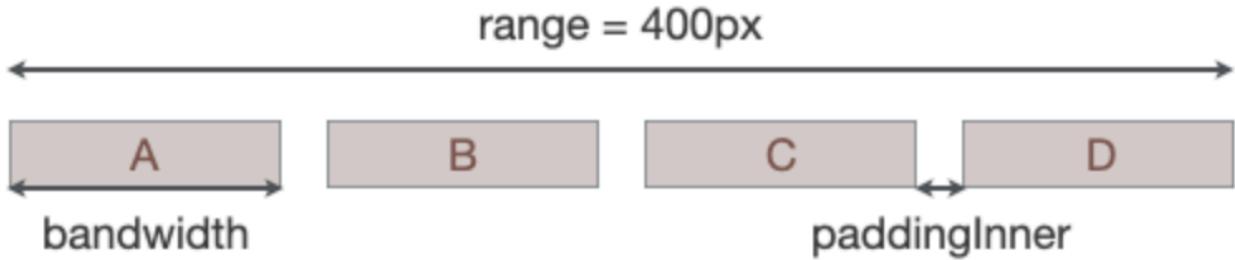
```

// Create an ordinal scale function
const xScale = d3.scaleBand()
  .domain(['Vanilla', 'Cookies', 'Chocolate', 'Pistachio'])
  .range([0, 400])           // D3 fits n (=4) bands within a 400px
space
  .paddingInner(0.05); // Adds spacing between bands

// By definition all bands have the same width
// and you can get it with `xScale.bandwidth()`

// We can use JS .map() instead of manually specifying all possible
values
const months = quarterlyReport.map(d => d.month);
months // Returns: ['May', 'June', 'July', 'August']

```



This is how you can create bar charts.

D3 Color Scales

What if you want to apply color to graphical elements based on data? In the previous examples, we had to write an set of 'if/else' statements with pre-computed color values. This isn't practical. Here's a better way:

```
// Construct a new ordinal scale with a range of ten categorical
colours
var colorPalette = d3.scaleOrdinal(d3.schemeCategory10);

// We can log the color range and see 10 distinct hex colours
console.log(colorPalette.range());
// [#1f77b4", "#ff7f0e", "#2ca02c", "#d62728", "#9467bd", "#8c564b",
"#e377c2", "#7f7f7f", "#bcbd22", "#17becf"]

// Specify domain (optional)
colorPalette.domain(['Vanilla', 'Cookies', 'Chocolate', 'Pistachio']);

// Use color palette
colorPalette("Chocolate") // Returns: #2ca02c
```

You can bind the color scale to the data, and render it giving us:



Or you can use a color gradient, for instance with a linear scale.

```
const linearColor = d3.scaleLinear()  
  .domain([0,100])  
  .range(['lightgreen', 'darkgreen']);  
  
linearColor(0) // Returns: #90ee90
```



Here are some example color scales you can use :

<https://github.com/d3/d3-scale-chromatic>

We will discuss 'best practices' for applying colors in a future classes.

D3 scale invert

What if you have made a scale, for instance to go from your data values to pixel coordinates, and now you need to do the reverse- go from pixel coordinates to data values? Suppose someone interacts within your visualization and you know the

coordinate of the mouse click in pixels, but need to know what data is closest? D3 has a scale invert method.

Details here: <https://www.d3indepth.com/scales/#inversion>

SVG Groups

Last time, we created graphical elements in the DOM and applied attributes, based on data. But what if we have a set of graphical elements that we want to operate on as a group, such as shifting these elements over or apply other transformations. We can use the `<g></g>` group element, which is not visible onscreen, but it gives us the capacity to group elements together so we can put them into hierarchies or apply transformations on sets of elements- such as translate or rotate

```
// Group element with 'transform' attribute
// x = 70, y = 50 (moves the whole group 70px right and 50px down)
const group = svg.append("g").attr("transform", "translate(70, 50)");

// Append circle to the group
const circle = group.append('circle')
    .attr('r', 4)
    .attr('fill', 'blue');
```



You will use this idea of groups in a number of places, but it comes into play with the creation of axes, as discussed below.

D3 Axes

Axes represent your scale visually. Creating a scale using pure javascript would be a pain- think of how you would specify every tick mark, every text label, ensure that the text doesn't get too cluttered. D3 has built in axes for you- just attach the scale you created for your data, and apply some customizations, and it should just work.

First customization to know about: Axes position- top, bottom, left or right. If it is positioned at the top, the text will be above the axis line, the line will extend vertically. If at the bottom, the text will be below the axis line, and so on.: `d3.axisTop`, `d3.axisBottom`, `d3.axisLeft`, and `d3.axisRight` .

```
// Create a horizontal axis with labels placed below the axis
const xAxis = d3.axisBottom();

// Pass in the scale function
xAxis.scale(xScale);
```

You can add attributes to the axis through chaining, specifying:

- Number of ticks: `.ticks(5)`
- Tick format, e.g. as percentage: `.tickFormat(d3.format(".0%"))`
- Predefined tick values: `.tickValues([0, 10, 20, 30, 40])`
- Remove tick marks at the beginning and end of an axis: `.tickSizeOuter(0)`

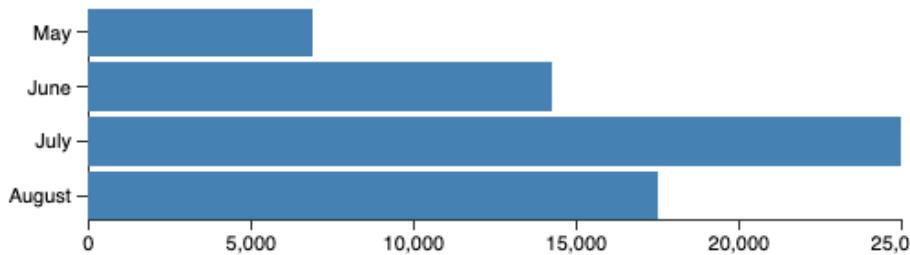
```
var xAxis = d3.axisBottom()
  .scale(xScale)
  . ... // Add options here
```

To add the axis to the svg, we use a group element. Why- you will see in the example below, we will need the capacity to shift it over and move it within the svg to account for margins so a plot fits comfortably in the svg. We create an SVG group element as a selection and use the `call()` function to hand it off to the `axis` function. All the axis elements are getting generated within that group.

```
// Draw the axis
svg.append('g')
  .attr('class', 'axis x-axis')
  .call(xAxis);
```

Let's make a bar chart!

We are going to make a bar chart.



In this first version, we will use what we have learned so far, and then we will have to amend this version to account for margins. Walk through it. You can also try creating this yourself in the starter template. Don't forget to run a local server, so you load the data.

```
/*
 * Load data from CSV file
```

```

*/
d3.csv('data/sales.csv')
  .then(data => {
    // Convert sales strings to numbers
    data.forEach(d => {
      d.sales = +d.sales;
    });

    showBarChart(data);
  })
  .catch(error => {
    console.error('Error loading the data');
  });
}

/*
 * Draw chart
 */
function showBarChart(data) {
  const width = 500;
  const height = 120;

  // Append empty SVG container and set size
  const svg = d3.select('#chart').append('svg')
    .attr('width', width)
    .attr('height', height);

  // Init linear+ordinal scales (input domain and output range)
  const xScale = d3.scaleLinear()
    .domain([0, d3.max(data, d => d.sales)]) //max from array
    .range([0, width]);

  const yScale = d3.scaleBand()
    .domain(data.map(d => d.month)) //month field in objs in array
    .range([0, height])
    .paddingInner(0.1);

  // Initialize axes
  const xAxis = d3.axisBottom(xScale);
  const yAxis = d3.axisLeft(yScale);

```

```

// Draw the axis
const xAxisGroup = svg.append('g')
  .attr('class', 'axis x-axis')
  .call(xAxis);

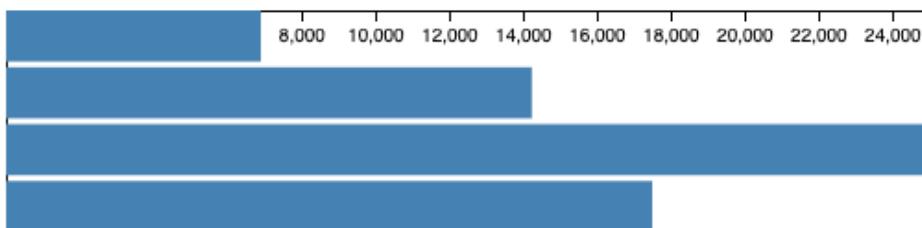
const yAxisGroup = svg.append('g')
  .attr('class', 'axis y-axis')
  .call(yAxis);

// Add rectangles
const bars = svg.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
    .attr('class', 'bar')
    .attr('fill', 'steelblue')
    .attr('width', d => xScale(d.sales))
    .attr('height', yScale.bandwidth())
    .attr('y', d => yScale(d.month))
    .attr('x', 0);
}

}

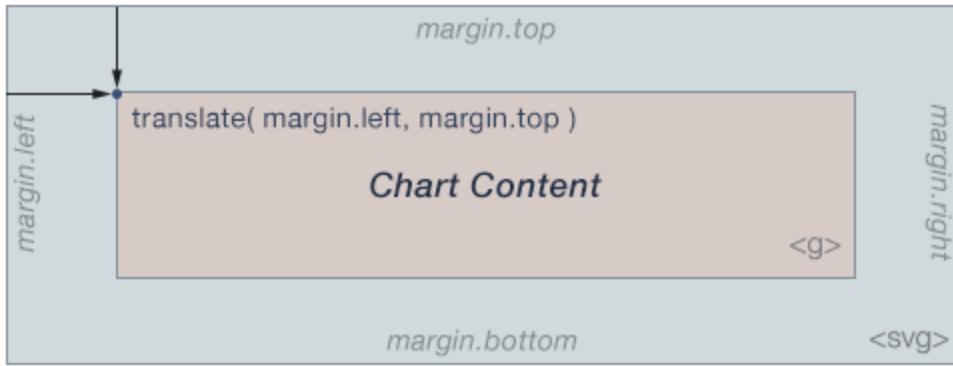
```

This intermediate result is almost there.



But, the chart isn't quite right- we can't see the left axis, the bottom axis is drawn on the top. Because we have put these in a group, we can apply a transform to shift the axis over, to accommodate margins.

D3 has a convention for margins in the plot. Here is what that is:



Here's how this shows up in our revised code- just the showBarChart function.

```
function showBarChart(data) {
    // Margin object with properties for the four directions
    const margin = {top: 5, right: 5, bottom: 20, left: 50};

    // Width and height as the inner dimensions of the chart area
    const width = 500 - margin.left - margin.right,
        height = 140 - margin.top - margin.bottom;

    // Define 'svg' child-element (g) from drawing area with spaces
    const svg = d3.select('#chart').append('svg')
        .attr('width', width + margin.left + margin.right)
        .attr('height', height + margin.top + margin.bottom)
        .append('g')
        .attr('transform', `translate(${margin.left},
${margin.top})`);

    // All subsequent functions/properties can ignore the margins
    // Initialize linear + ordinal scales
    const xScale = d3.scaleLinear()
        .domain([0, d3.max(data, d => d.sales)])
        .range([0, width]);

    const yScale = d3.scaleBand()
        .domain(data.map(d => d.month))
        .range([0, height])
        .paddingInner(0.15);

    // Initialize axes
```

```

const xAxis = d3.axisBottom(xScale)
  .ticks(6)
  .tickSizeOuter(0);

const yAxis = d3.axisLeft(yScale)
  .tickSizeOuter(0);

// Draw the axis (move xAxis to the bottom with 'translate')
const xAxisGroup = svg.append('g')
  .attr('class', 'axis x-axis')
  .attr('transform', `translate(0, ${height})`)
  .call(xAxis);

const yAxisGroup = svg.append('g')
  .attr('class', 'axis y-axis')
  .call(yAxis);

// Add rectangles
svg.selectAll('rect')
  .data(data)
  .enter()
  .append('rect')
    .attr('class', 'bar')
    .attr('width', d => xScale(d.sales))
    .attr('height', yScale.bandwidth())
    .attr('y', d => yScale(d.month))
    .attr('x', 0);

```

To make the chart look nice, we also can apply some styles in our style sheet. Take note of 'shape-rendering: crispEdges' which will help us avoid blurry lines and bars.

```

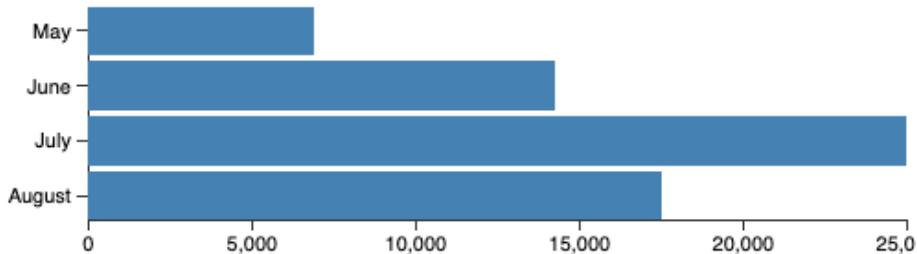
.axis path,
.axis line {
  fill: none;
  stroke: #333;
  shape-rendering: crispEdges;
}

.axis text {

```

```
    font-family: sans-serif;
    font-size: 11px;
}

.bar {
    fill: steelblue;
    shape-rendering: crispEdges;
}
```



Credits:

Coding examples are from :

https://github.com/UBC-InfoVis/2021-436V-tutorials/tree/master/2_D3_Tutorial