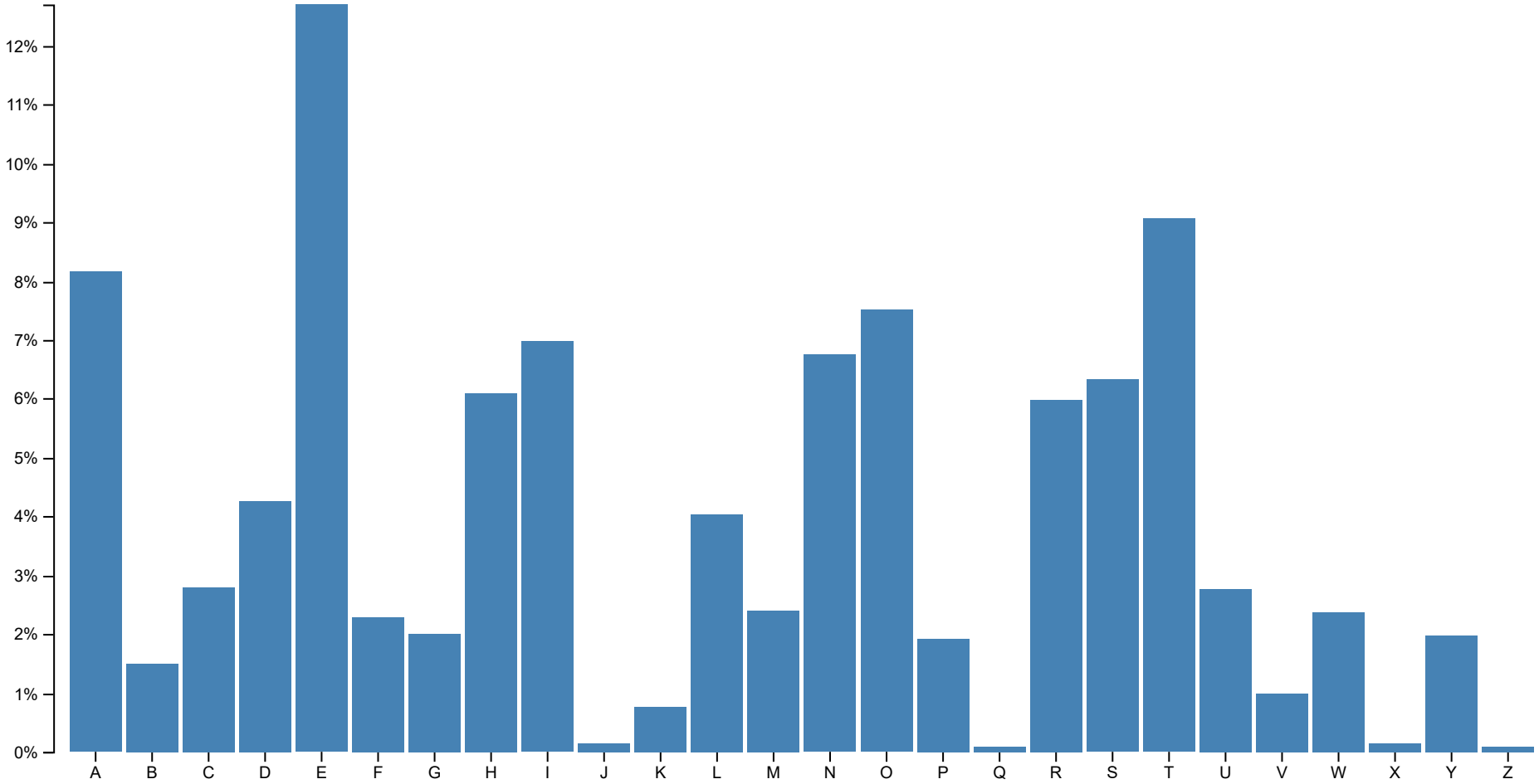
November 13, 2013 / Mike Bostock

Let's Make a Bar Chart, III

In the previous parts of this tutorial we made a basic bar chart in HTML and then in SVG; now, we'll improve the display by rotating the chart into columns and adding axes. We'll also switch to a real dataset, showing the relative frequency of letters in the English language.



Source: *Cryptological Mathematics*, Robert Lewand.

bl.ocks.org/3885304

Rotating into Columns

Rotating a bar chart into a column chart largely involves swapping *x* with *y*. However, a number of smaller incidental changes are also required. This is the cost of working directly with SVG rather than using a high-level visualization grammar like ggplot2. On the other hand, SVG offers greater customizability; and SVG is a web standard, so we can use the browser's developer tools like the element inspector, and use SVG for things beyond visualization.

When renaming the `x` scale to the `y` scale, the range becomes `[height, 0]` rather than `[0, width]`. This is because the origin of SVG's coordinate system is in the top-left corner. We want the zero-value to be positioned at the bottom of the chart, rather than the top. Likewise this means that we need to position the bar `rects` by setting the `"y"` and `"height"` attributes, whereas before we only needed to set `"width"`. (The default value of the `"x"` attribute is zero, and the old bars were left-aligned.)

We previously multiplied the `var barHeight` by the index of each data point (0, 1, 2, …) to produce fixed-height bars. The resulting chart's height thus depended on the size of the dataset. But here the opposite behavior is desired: the chart width is fixed and the bar width variable. So rather than fix the `barHeight`, now we compute the `barWidth` by dividing the available chart width by the size of the dataset, `data.length`.

Lastly, the bar labels must be positioned differently for columns rather than bars, centered just below the top of the column. The new `"dy"` attribute value of `".75em"` anchors the label at approximately the text's cap height rather than the baseline.

bl.ocks.org/7452541

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.chart rect {
  fill: steelblue;
}
```

```
.chart text {
  fill: white;
  font: 10px sans-serif;
  text-anchor: middle;
}

</style>
<svg class="chart"></svg>
<script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
<script>

var width = 960,
    height = 500;

var y = d3.scale.linear()
    .range([height, 0]);

var chart = d3.select(".chart")
    .attr("width", width)
    .attr("height", height);

d3.tsv("data.tsv", type, function(error, data) {
  y.domain([0, d3.max(data, function(d) { return d.value; })]);

  var barWidth = width / data.length;

  var bar = chart.selectAll("g")
      .data(data)
    .enter().append("g")
      .attr("transform", function(d, i) { return "translate(" + i * barWidth + ",0)"; });

  bar.append("rect")
      .attr("y", function(d) { return y(d.value); })
      .attr("height", function(d) { return height - y(d.value); })
      .attr("width", barWidth - 1);

  bar.append("text")
      .attr("x", barWidth / 2)
      .attr("y", function(d) { return y(d.value) + 3; })
      .attr("dy", ".75em")
```

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

function type(d) {
  d.value = +d.value; // coerce to number
  return d;
}

</script>
```

Encoding Ordinal Data

Unlike *quantitative* values that can be compared numerically, subtracted or divided, *ordinal* values are compared by rank. Letters are ordinal; in the alphabet, A occurs before B, and B before C. Whereas D3's linear, pow and log scales serve to encode quantitative data, ordinal scales encode ordinal data. We can thus use an ordinal scale to simplify the positioning of bars by letter.

In its most explicit form, an ordinal scale is a mapping from a discrete set of data values (such as names) to a corresponding discrete set of display values (such as pixel positions). Like quantitative scales these sets are called the *domain* and *range*, respectively.

```
var x = d3.scale.ordinal()
    .domain(["A", "B", "C", "D", "E", "F"])
    .range([0, 1, 2, 3, 4, 5]);
```

To keep this example code short, only the first six letters of the alphabet are shown. In the full code below, we'll compute the scale's domain from the data and it will include every letter.

The result of `x("A")` is `0`, `x("B")` is `1`, and so on. In specifying the domain and range, all that matters is the order of values: element *i* in the domain is mapped to element *i* in the range.

It would be tedious to enumerate the positions of each bar by hand, so instead we can convert a continuous range into a discrete set of values using rangeBands or rangePoints. The rangeBands method computes range values so as to divide the chart area into evenly-spaced, evenly-sized bands, as in a bar chart. The similar rangePoints method computes evenly-spaced range values as in a scatterplot. For example:

```
var x = d3.scale.ordinal()
    .domain(["A", "B", "C", "D", "E", "F"])
    .rangeBands([0, width]);
```

If `width` is `960`, `x("A")` is now `0` and `x("B")` is `160`, and so on. These positions serve as the left edge of each bar, while `x.rangeBand()` returns the width of each bar. But rangeBands can also add padding between bars with an optional third argument, and the rangeRoundBands variant will compute positions that snap to exact pixel boundaries for crisp edges. Compare:

The full range can be retrieved for inspection with `x.range()`.

```
var x = d3.scale.ordinal()
    .domain(["A", "B", "C", "D", "E", "F"])
    .rangeRoundBands([0, width], .1);
```

Now `x("A")` is `17` and each bar is 141 pixels wide. And, rather than hard-code the letters in our domain, we can compute them from data using array.map and array.sort. All together:

bl.ocks.org/7440840

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.chart rect {
  fill: steelblue;
}

.chart text {
  fill: white;
  font: 10px sans-serif;
  text-anchor: middle;
}

</style>
<svg class="chart"></svg>
<script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
<script>

var width = 960,
    height = 500;
```

```
var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);

var y = d3.scale.linear()
    .range([height, 0]);

var chart = d3.select(".chart")
    .attr("width", width)
    .attr("height", height);

d3.tsv("data.tsv", type, function(error, data) {
  x.domain(data.map(function(d) { return d.name; }));
  y.domain([0, d3.max(data, function(d) { return d.value; })]);

  var bar = chart.selectAll("g")
      .data(data)
    .enter().append("g")
      .attr("transform", function(d) { return "translate(" + x(d.name) + ",0)"; });

  bar.append("rect")
      .attr("y", function(d) { return y(d.value); })
      .attr("height", function(d) { return height - y(d.value); })
      .attr("width", x.rangeBand());

  bar.append("text")
      .attr("x", x.rangeBand() / 2)
      .attr("y", function(d) { return y(d.value) + 3; })
      .attr("dy", ".75em")
      .text(function(d) { return d.value; });
});

function type(d) {
  d.value = +d.value; // coerce to number
  return d;
}

</script>
```

Preparing Margins

Ordinal scales are often used in conjunction with D3's axis component to quickly display labeled tick marks, improving the chart's legibility. But before we can add an axis, we need to clear some space in the margins.

By convention, margins in D3 are specified as an object with top, right, bottom and left properties. Then, the *outer* size of the chart area, which includes the margins, is used to compute the *inner* size available for graphical marks by subtracting the margins. For example, reasonable values for a 960×500 chart are:

Unlike HTML, the SVG container is implicitly overflow: hidden. IE9 allows overflow, but other browsers do not, so you shouldn't rely on it.

```
var margin = {top: 20, right: 30, bottom: 30, left: 40},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;
```

Thus, 960 and 500 are the outer width and height, respectively, while the computed inner `width` and `height` are 890 and 450. These inner dimensions can be used to initialize scale ranges. To apply the margins to the SVG container, we set the width and height of the SVG element to the outer dimensions, and add a `g` element to offset the origin of the chart area by the top-left margin.

```
var chart = d3.select(".chart")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

Any elements subsequently added to `chart` will thus inherit the margins.

Adding Axes

We define an axis by binding it to our existing *x*-scale and declaring one of the four orientations. Since our *x*-axis will appear below the bars, here we use the `"bottom"` orientation.

```
var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");
```

The resulting `xAxis` object be used to render multiple axes by repeated application using selection.call. Think of it as a rubber stamp which can print out axes wherever they are needed. The axis elements are positioned relative to a local origin, so to transform into the desired position we set the `"transform"` attribute on a containing `g` element:

```
chart.append("g")
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);
```

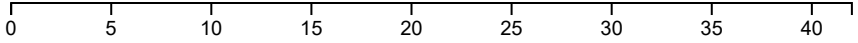
The axis container should also have a class name so that we can apply styles. The name `"axis"` here is arbitrary; call it whatever you like. Multiple class names, such as `"x axis"`, are useful for styling axes differently by dimension while retaining some shared styles across dimensions.

The axis component consists of a `path` element which displays the domain, and multiple `g` `".tick"` elements for each tick mark. A tick in turn contains a `text` label and a `line` mark. Most of D3's examples therefore use the following minimalist style:

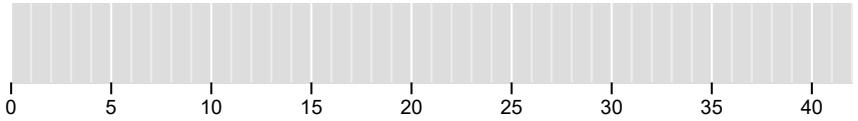
```
.axis text {
  font: 10px sans-serif;
}

.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}
```

This produces an axis reminiscent of R:



However, axes are highly customizable. This more elaborate axis is styled after ggplot2:



Beyond styles, you can further customize the appearance of an axis by selecting its elements and modifying them after the axis is created; the elements of an axis are part of its public API. The ggplot2-style axis above is rendered using two overlaid axes, one inside the chart area and one outside, in the bottom margin. The major and minor ticks are styled differently.

With axes, we can now remove the labels from the bar. The complete code:

bl.ocks.org/7441121

```
<!DOCTYPE html>
<meta charset="utf-8">
<style>

.bar {
  fill: steelblue;
}

.axis text {
  font: 10px sans-serif;
}

.axis path,
.axis line {
  fill: none;
  stroke: #000;
  shape-rendering: crispEdges;
}

.x.axis path {
  display: none;
```

```
}

</style>
<svg class="chart"></svg>
<script src="//d3js.org/d3.v3.min.js" charset="utf-8"></script>
<script>

var margin = {top: 20, right: 30, bottom: 30, left: 40},
    width = 960 - margin.left - margin.right,
    height = 500 - margin.top - margin.bottom;

var x = d3.scale.ordinal()
    .rangeRoundBands([0, width], .1);

var y = d3.scale.linear()
    .range([height, 0]);

var xAxis = d3.svg.axis()
    .scale(x)
    .orient("bottom");

var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left");

var chart = d3.select(".chart")
    .attr("width", width + margin.left + margin.right)
    .attr("height", height + margin.top + margin.bottom)
  .append("g")
    .attr("transform", "translate(" + margin.left + "," + margin.top + ")");

d3.tsv("data.tsv", type, function(error, data) {
  x.domain(data.map(function(d) { return d.name; }));
  y.domain([0, d3.max(data, function(d) { return d.value; })]);

  chart.append("g")
      .attr("class", "x axis")
      .attr("transform", "translate(0," + height + ")")
      .call(xAxis);

  chart.append("g")
```

```
      .attr("class", "y axis")
      .call(yAxis);

  chart.selectAll(".bar")
      .data(data)
    .enter().append("rect")
      .attr("class", "bar")
      .attr("x", function(d) { return x(d.name); })
      .attr("y", function(d) { return y(d.value); })
      .attr("height", function(d) { return height - y(d.value); })
      .attr("width", x.rangeBand());
});

function type(d) {
  d.value = +d.value; // coerce to number
  return d;
}

</script>
```

Communicating

At this point in the tutorial, I must apologize. I have done you a disservice.

In my effort to explain the technical details of chart construction, I have glossed over a critical component of effective visualization: *effective communication*. It doesn't matter how good a chart looks if it doesn't communicate anything! We must label the chart to give the reader sufficient context to interpret it.

This problem is more pervasive than you might expect. When visualizing data, it's easy to internalize and forget the extra knowledge you have about your dataset or your intent. *You* know it's a bar chart of relative frequency of letters in the English language. But unless you state that explicitly, your reader may not. Labels, captions, legends and other explanatory elements are essential to understanding. A title can be added to the *y*-axis by appending a `text` element and positioning it as desired.

```
chart.append("g")
    .attr("class", "y axis")
    .call(yAxis)
  .append("text")
    .attr("transform", "rotate(-90)")
    .attr("y", 6)
    .attr("dy", ".71em")
    .style("text-anchor", "end")
    .text("Frequency");
```

Unit-appropriate number formatting also improves legibility by tailoring the display to your data. Since our chart displays relative frequency, percentages are more appropriate than the default behavior which shows a number between 0 and 1. A format string as the second argument to axis.ticks will customize the tick formatting, and the scale will automatically choose a precision appropriate to the tick interval.

```
var yAxis = d3.svg.axis()
    .scale(y)
    .orient("left")
    .ticks(10, "%");
```

For total control over formatting, you can instead pass a function to axis.tickFormat.

Next: Part 4

The code for part 3 of the tutorial is available at bl.ocks.org/3885304.

The next tutorial in this series will cover interaction and transitions between views. Follow me on Twitter to be notified when the next section is published.