# Data visualization

COSC 480B

Reyan Ahmed

rahmed1@colgate.edu

# Lecture 8

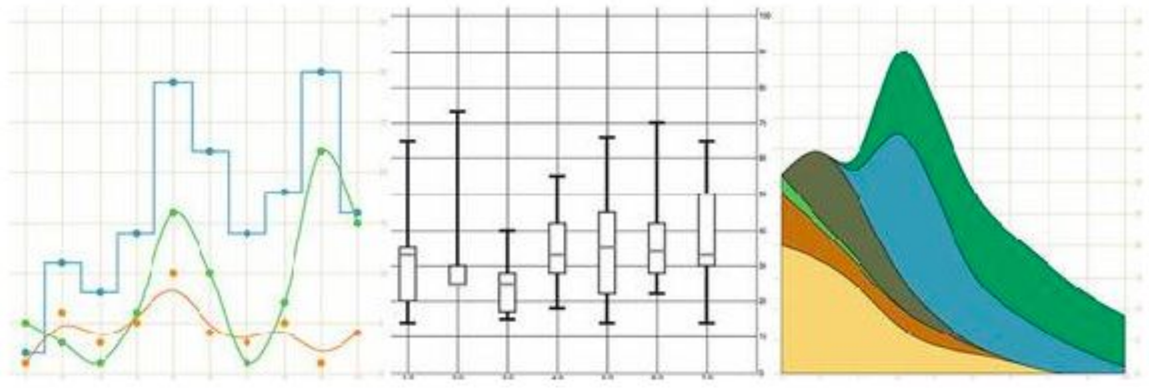
Chart components

## Overview

- Creating and formatting axis components
- Creating legends
- Using line and area generators for charts
- Creating complex shapes consisting of multiple types of SVG elements

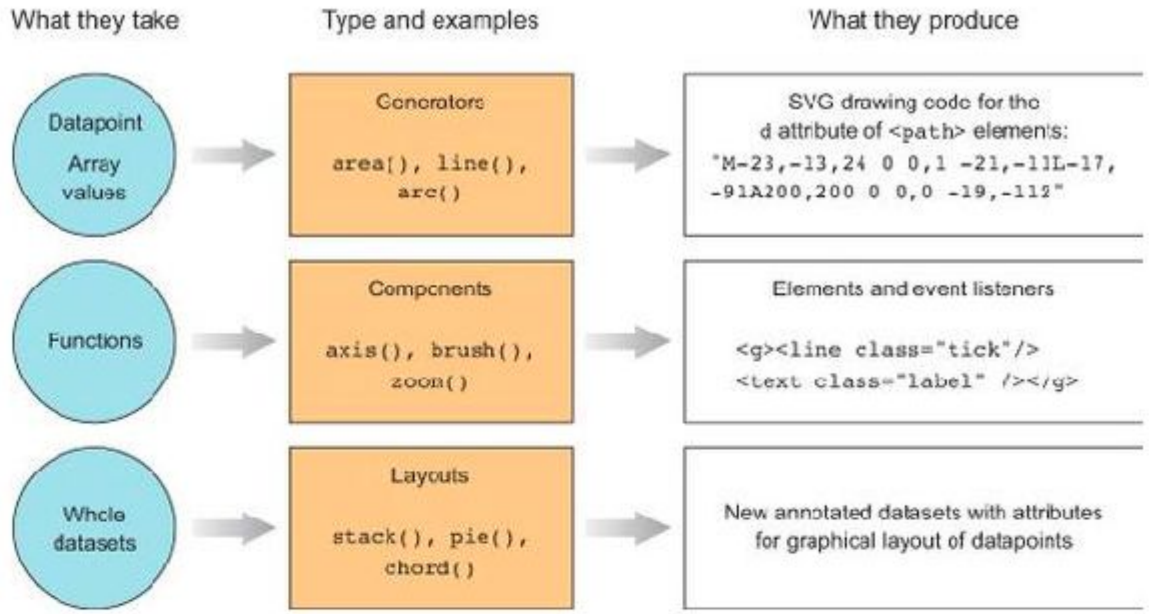
## Overview

The charts we’ll create in this chapter using D3 generators and components. From left to right: a line chart, a boxplot, and a streamgraph.



## General charting principles

The three main types of functions found in D3 can be classified as generators, components, and layouts. You'll see components and generators in this chapter and layouts in the next chapter.


What they take
Type and examples
What they produce
Datapoint
Array values
Generators
area(), line(), arc()
SVG drawing code for the d attribute of <path> elements:
"M-23,-13,24 0 0,1 -21,-11L-17, -91A200,200 0 0,0 -19,-112"
Functions
Components
axis(), brush(), zoom()
Elements and event listeners
<g><line class="tick"/> <text class="label" /></g>
Whole datasets
Layouts
stack(), pie(), chord()
New annotated datasets with attributes for graphical layout of datapoints

## General charting principles

- Generators: draw shapes from data points
    - Area
    - Line
    - Arc
- Components: generators produces the attributes (d attribute), components create graphical objects like line, group etc
    - Axis
    - Brush
    - Zoom
- Layouts:
    - Pie chart
    - Force directed layouts

## Creating an axis

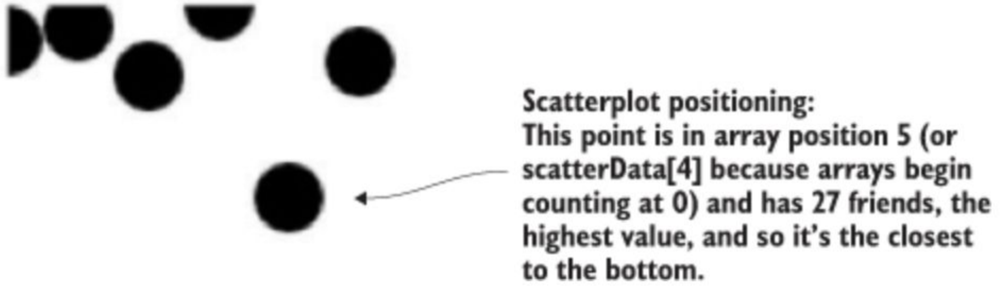
```
var scatterData = [{friends: 5, salary: 22000},
{friends: 3, salary: 18000}, {friends: 10, salary:
88000},
{friends: 0, salary: 180000}, {friends: 27, salary:
56000},
{friends: 8, salary: 74000}];
```

## Creating an axis

```
d3.select("svg").selectAll("circle")
   .data(scatterData).enter()
   .append("circle")
   .attr("r", 5)
   .attr("cx", (d,i) => i * 10)
   .attr("cy", d => d.friends)
```

## Creating an axis

Circle positions indicate the number of friends and the array position of each datapoint.



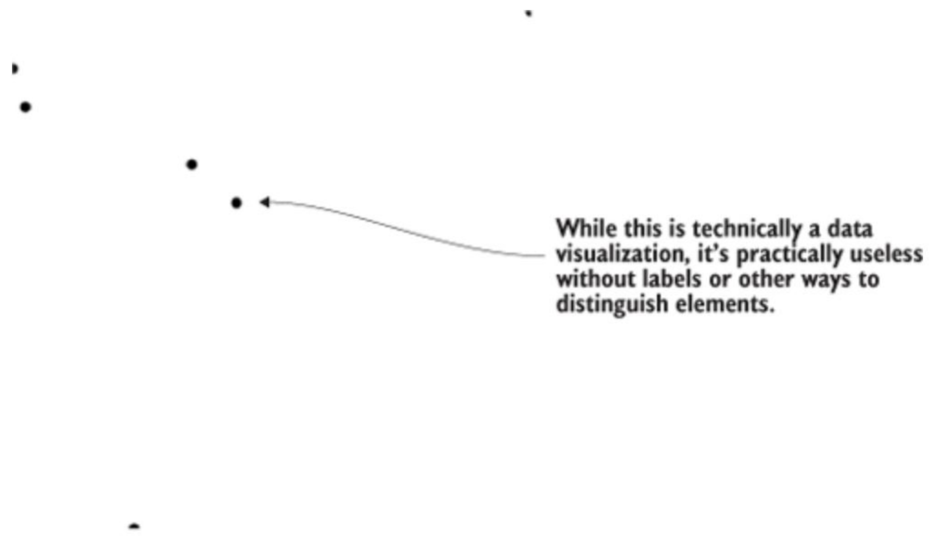
**Scatterplot positioning:**
**This point is in array position 5 (or scatterData[4] because arrays begin counting at 0) and has 27 friends, the highest value, and so it's the closest to the bottom.**

## Creating an axis

```
var xExtent = d3.extent(scatterData, d => d.salary)
var yExtent = d3.extent(scatterData, d => d.friends)
var xScale =
d3.scaleLinear().domain(xExtent).range([0,500]);
var yScale =
d3.scaleLinear().domain(yExtent).range([0,500]);
d3.select("svg").selectAll("circle")
  .data(scatterData).enter().append("circle")
  .attr("r", 5).attr("cx", d => xScale(d.salary))
  .attr("cy", d => yScale(d.friends));
```

## Creating an axis

Any point closer to the bottom has more friends, and any point closer to the right has a higher salary. But that’s not clear at all without labels, which we’re going to make.



## Creating an axis

D3 provides different selections:

- d3.axisLeft()
- d3.axisRight()
- d3.axisBottom()
- d3.axisTop()

## Creating an axis

```
var yAxis = d3.axisRight().scale(yScale);
d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis)
var xAxis = d3.axisBottom().scale(xScale)
d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis)
```

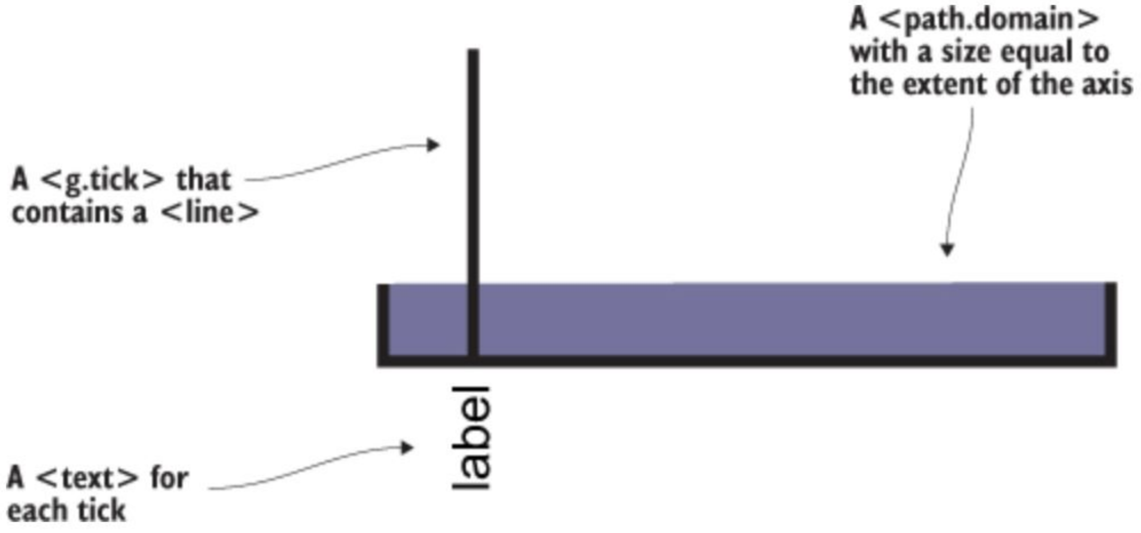
## Creating an axis

Notice that the .call() method of a selection invokes a function with the selection that’s active in the method chain, and is the equivalent of writing

```
xAxis(d3.select("svg").append("g").attr("id", "xAxisG"))
```

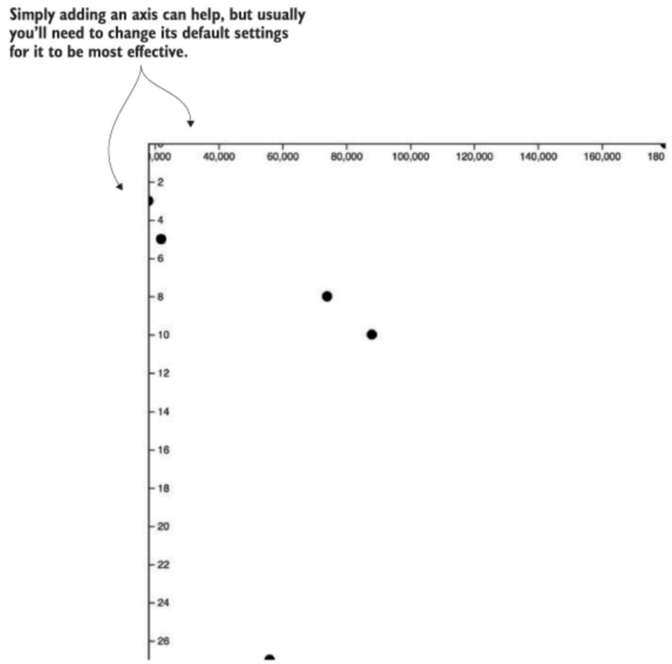
## Creating an axis

Elements of an axis created from d3.axis are a <path.domain> with a size equal to the extent of the axis, a <g.tick > that contains a <line> and a <text> for each tick.


A <path.domain> with a size equal to the extent of the axis
A <g.tick> that contains a <line>
label
A <text> for each tick

## Creating an axis

Default styles for an axis display the ticks and don't fill the domain area.



## Creating an axis

- You may need to adjust the default style a little bit
- To move our axes around, we need to adjust the .attr("translate") of their parent <g> elements, either when we draw them or later.

```
d3.selectAll("#xAxisG").attr("transform","translate(0,500)")
```

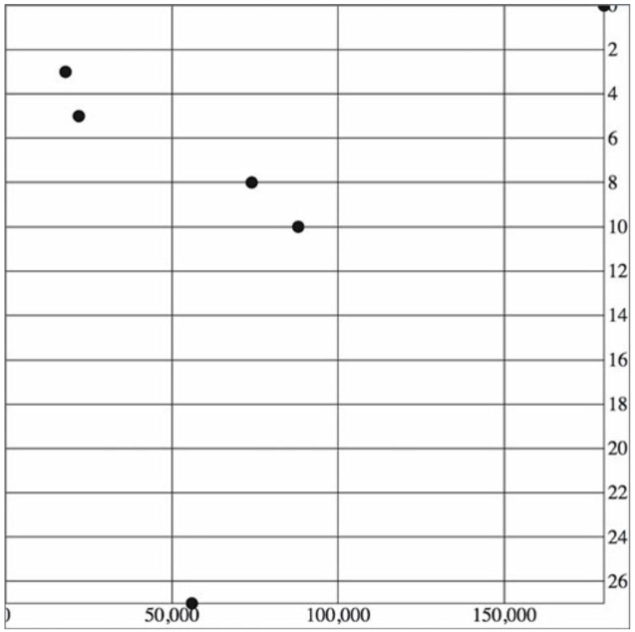
## Creating an axis

```
var scatterData = [{friends: 5, salary: 22000},
   {friends: 3, salary: 18000}, {friends: 10, salary: 88000},
   {friends: 0, salary: 180000}, {friends: 27, salary: 56000},
   {friends: 8, salary: 74000}];
var xScale = d3.scaleLinear().domain([0,180000]).range([0,500])       1
var yScale = d3.scaleLinear().domain([0,27]).range([0,500])
xAxis = d3.axisBottom().scale(xScale)
   .tickSize(500).ticks(4)                                     2
d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis)
yAxis = d3.axisRight().scale(yScale)
   .ticks(16).tickSize(500)
d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis)         3
d3.select("svg").selectAll("circle")
   .data(scatterData).enter()
   .append("circle").attr("r", 5)
   .attr("cx", d => xScale(d.salary))
   .attr("cy", d => yScale(d.friends))
```

1 Creates a pair of scales to map the values in our dataset to the canvas
2 Uses method chaining to create an axis and explicitly set its orientation, tick size, and number of ticks
3 Appends a <g> element to the canvas, and calls the axis from that <g> to create the necessary graphics for the axis

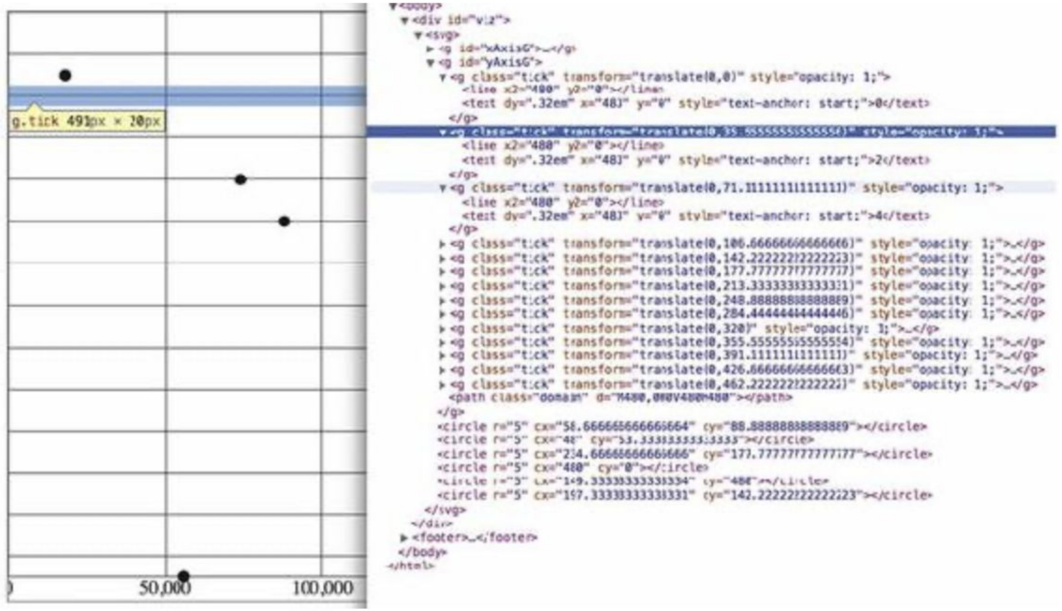
## Creating an axis

With CSS settings corresponding to the tick <line> elements, we can draw a rather attractive grid based on our two axes.



## Creating an axis

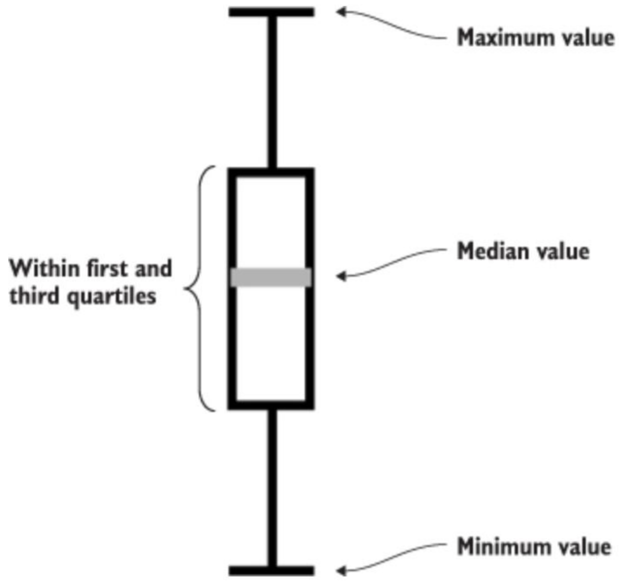
The DOM shows how tick <line> elements are appended along with a <text> element for the label to one of a set of <g.tick.major> elements corresponding to the number of ticks.



```
▼<body>
  ▼<div id="viz">
    ▼<svg>
      ▶<g id="xAxisG">…</g>
      ▼<g id="yAxisG">
        ▼<g class="tick" transform="translate(0,0)" style="opacity: 1;">
            <line x2="480" y2="0"></line>
            <text dy=".32em" x="483" y="0" style="text-anchor: start;">0</text>
          </g>
        ▼<g class="tick" transform="translate(0,35.55555555555556)" style="opacity: 1;">
            <line x2="480" y2="0"></line>
            <text dy=".32em" x="483" y="0" style="text-anchor: start;">2</text>
          </g>
        ▼<g class="tick" transform="translate(0,71.11111111111111)" style="opacity: 1;">
            <line x2="480" y2="0"></line>
            <text dy=".32em" x="483" y="0" style="text-anchor: start;">4</text>
          </g>
        ▶<g class="tick" transform="translate(0,106.66666666666666)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,142.22222222222223)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,177.77777777777777)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,213.33333333333331)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,248.88888888888889)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,284.44444444444446)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,320)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,355.55555555555554)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,391.1111111111111)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,426.66666666666663)" style="opacity: 1;">…</g>
        ▶<g class="tick" transform="translate(0,462.2222222222222)" style="opacity: 1;">…</g>
          <path class="domain" d="M480,0H0V480H480"></path>
        </g>
        <circle r="5" cx="58.666666666666664" cy="88.88888888888889"></circle>
        <circle r="5" cx="48" cy="53.333333333333336"></circle>
        <circle r="5" cx="234.66666666666666" cy="177.77777777777777"></circle>
        <circle r="5" cx="480" cy="0"></circle>
        <circle r="5" cx="149.33333333333334" cy="480"></circle>
        <circle r="5" cx="197.33333333333331" cy="142.22222222222223"></circle>
      </svg>
    </div>
  ▶<footer>…</footer>
  </body>
</html>
```

## Complex graphical objects

A box from a boxplot consists of five pieces of information encoded in a single shape: (1) the maximum value, (2) the high value of some distribution, such as the third quartile, (3) the median or mean value, (4) the corresponding low value of the distribution, such as the first quartile, and (5) the minimum value.



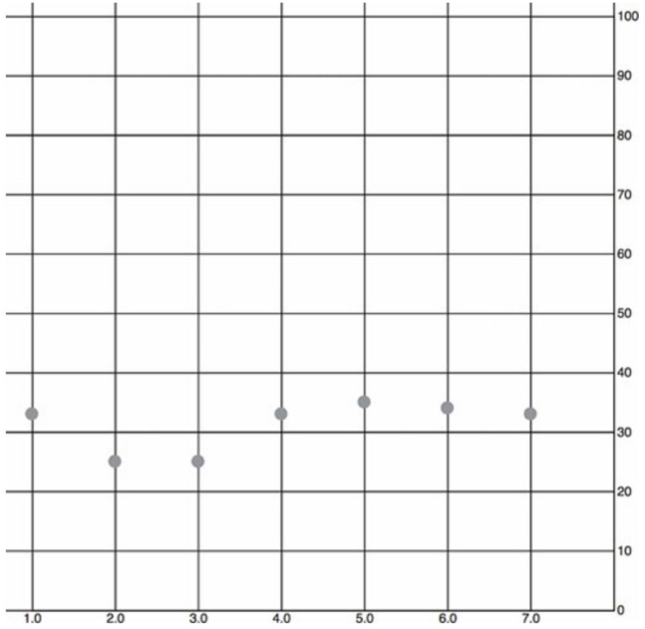
# Complex graphical objects

boxplots.csv

```
day,min,max,median,q1,q3,number
1,14,65,33,20,35,22
2,25,73,25,25,30,170
3,15,40,25,17,28,185
4,18,55,33,28,42,135
5,14,66,35,22,45,150
6,22,70,34,28,42,170
7,14,65,33,30,50,28
```

## Complex graphical objects

The median age of visitors (y-axis) by day of the week (x-axis) as represented by a scatterplot. It shows a slight dip in age on the second and third days.



# Complex graphical objects

## Scatterplot of average age

```
d3.csv("boxplot.csv", scatterplot)                                  1
const tickSize = 470
function scatterplot(data) {
  const xScale =
d3.scaleLinear().domain([1,8]).range([20,tickSize])
  const yScale = d3.scaleLinear().domain([0,100]).range([tickSize
+ 10,20])    2
  const yAxis = d3.axisRight()
  .scale(yScale)
  .ticks(8)
  .tickSize(tickSize)
  d3.select("svg").append("g")
    .attr("transform", `translate(${tickSize},0)`)                    3
    .attr("id", "yAxisG")
    .call(yAxis)
```

1 Any value that you use more than once should be stored in a constant, so that you only have to change it once later and so others can read your code
2 Scale is inverted, so higher values are drawn higher up and lower values toward the bottom
3 Offsets the <g> containing the axis

## Complex graphical objects

```
   const xAxis = d3.axisBottom()
      .scale(xScale)
      .tickSize(-tickSize)
      .tickValues([1,2,3,4,5,6,7])                                    4
   d3.select("svg").append("g")
      .attr("transform", `translate(0,${tickSize + 10})`)
      .attr("id", "xAxisG")
      .call(xAxis)
   d3.select("svg").selectAll("circle.median")
      .data(data)
      .enter()
      .append("circle")
      .attr("class", "tweets")
      .attr("r", 5)
      .attr("cx", d => xScale(d.day))
      .attr("cy", d => yScale(d.median))
      .style("fill", "darkgray")
}
```

4 Specifies the exact tick values to correspond with the numbered days of the week

## Complex graphical objects

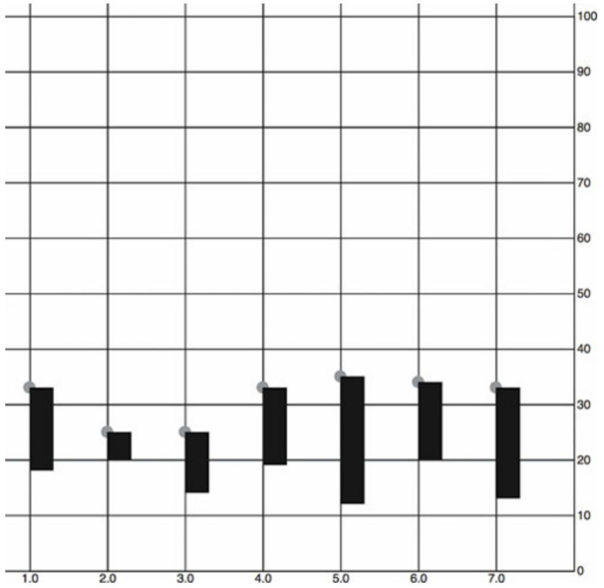
### Initial boxplot drawing code

```
d3.select("svg").selectAll("g.box")
  .data(data).enter()
  .append("g")
  .attr("class", "box")
  .attr("transform", d =>
        "translate(" + xScale(d.day) +"," + yScale(d.median) + ")"
    ).each(function(d,i) {                                          1
       d3.select(this)                                            2
         .append("rect")
         .attr("width", 20)
         .attr("height", yScale(d.q1) - yScale(d.q3));            3
    })
```

1 Your latest reminder that to get the this context you can't use arrow functions
2 Because we're inside the .each(), we can select(this) to append new child elements
3 The d and i variables are declared in the .each() anonymous function, so each time we access it, we get the data bound to the original element

# Complex graphical objects

The <rect> elements represent the scaled range of the first and third quartiles of visitor age. They're placed on top of a gray <circle> in each <g> element, which is placed on the chart at the median age. The rectangles are drawn, as per SVG convention, from the <g> down and to the right.



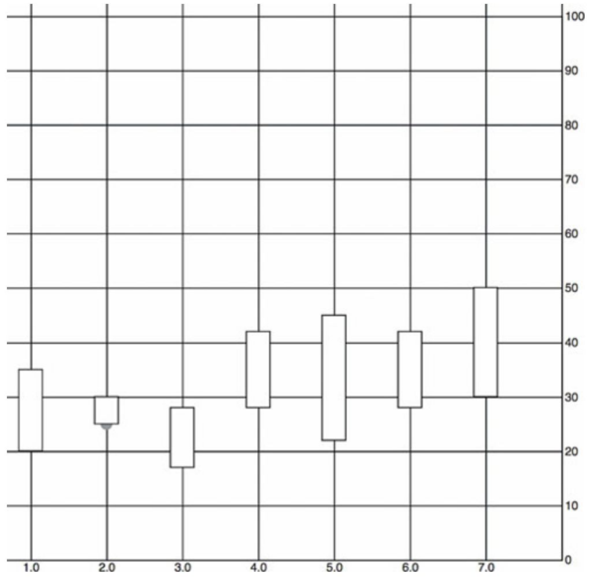
## Complex graphical objects

```
...
.each(function(d,i) {
    d3.select(this)
      .append("rect")
      .attr("width", 20)
      .attr("x", -10)                                        1
      .attr("y", yScale(d.q3) - yScale(d.median))         2
      .attr("height", yScale(d.q1) - yScale(d.q3))
      .style("fill", "white")
      .style("stroke", "black");
});
```

1 Sets a negative offset of half the width to center a rectangle horizontally
2 The height of the rectangle is equal to the difference between its q1 and q3 values, which means we need to offset the rectangle by the difference between the middle of the rectangle (the median) and the high end of the distribution—q3

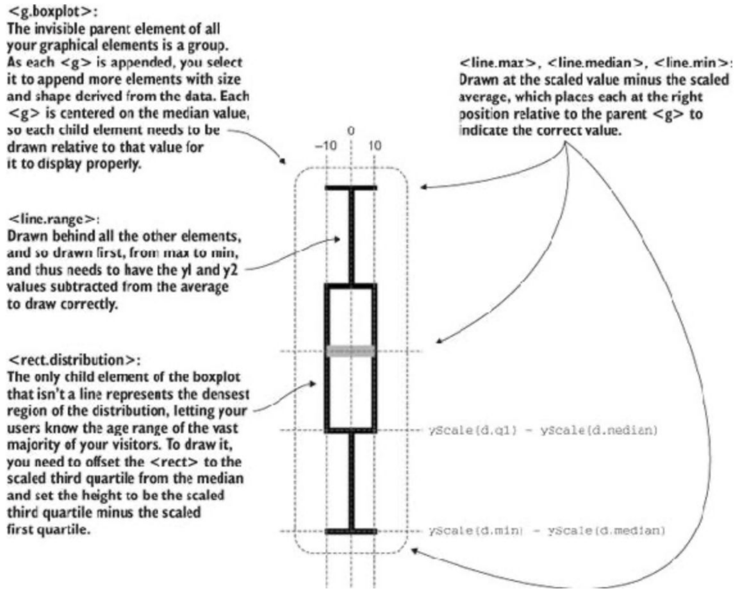
## Complex graphical objects

The <rect> elements are now properly placed so that their top and bottom correspond with the visitor age between the first and third quartiles of visitors for each day. The circles are completely covered, except for the second rectangle where the first quartile value is the same as the median age, so we can see half the gray circle peeking out from underneath it.



# Complex graphical objects

How a boxplot can be drawn in D3. Pay particular attention to the relative positioning necessary to draw child elements of a <g>. The 0 positions for all elements are where the parent <g> has been placed, so that <line.max>, <rect.distribution>, and <line.range> all need to be drawn with an offset placing their top-left corner above this center, whereas <line.min> is drawn below the center and <line.median> has a 0 y-value, because our center is the median value.


0
−10
10
yScale(d.q1) - yScale(d.median)
yScale(d.min) - yScale(d.median)

<g.boxplot>:
The invisible parent element of all your graphical elements is a group. As each <g> is appended, you select it to append more elements with size and shape derived from the data. Each <g> is centered on the median value, so each child element needs to be drawn relative to that value for it to display properly.

<line.range>:
Drawn behind all the other elements, and so drawn first, from max to min, and thus needs to have the yl and y2 values subtracted from the average to draw correctly.

<rect.distribution>:
The only child element of the boxplot that isn't a line represents the densest region of the distribution, letting your users know the age range of the vast majority of your visitors. To draw it, you need to offset the <rect> to the scaled third quartile from the median and set the height to be the scaled third quartile minus the scaled first quartile.

<line.max>, <line.median>, <line.min>:
Drawn at the scaled value minus the scaled average, which places each at the right position relative to the parent <g> to indicate the correct value.

## Complex graphical objects

```
...
.each(function(d,i) {
    d3.select(this)
      .append("line")
      .attr("class", "range")
      .attr("x1", 0)
      .attr("x2", 0)
      .attr("y1", yScale(d.max) - yScale(d.median))            1
      .attr("y2", yScale(d.min) - yScale(d.median))            1
      .style("stroke", "black")
      .style("stroke-width", "4px");
    d3.select(this)
      .append("line")
      .attr("class", "max")
      .attr("x1", -10)
      .attr("x2", 10)
      .attr("y1", yScale(d.max) - yScale(d.median))            2
      .attr("y2", yScale(d.max) - yScale(d.median))            2
      .style("stroke", "black")
      .style("stroke-width", "4px")
```

1 Draws the line from the max to the min value
2 The top bar of the min-max line

# Complex graphical objects

```
d3.select(this)
  .append("line")
  .attr("class", "min")
  .attr("x1", -10)
  .attr("x2", 10)
  .attr("y1", yScale(d.min) - yScale(d.median))            3
  .attr("y2", yScale(d.min) - yScale(d.median))            3
  .style("stroke", "black")
  .style("stroke-width", "4px")
d3.select(this)
  .append("rect")
  .attr("class", "range")
  .attr("width", 20)
  .attr("x", -10)
  .attr("y", yScale(d.q3) - yScale(d.median))             4
  .attr("height", yScale(d.q1) - yScale(d.q3))            4
  .style("fill", "white")
  .style("stroke", "black")
  .style("stroke-width", "2px")
```

3 The bottom bar of the min-max line
4 The offset so that the rectangle is centered on the median value

## Complex graphical objects

```
    d3.select(this)
      .append("line")
      .attr("x1", -10)                                      5
      .attr("x2", 10)                                       5
      .attr("y1", 0)
      .attr("y2", 0)
      .style("stroke", "darkgray")
      .style("stroke-width", "4px")
});
```

5 Median line doesn’t need to be moved, because the parent <g> is centered on the median value

# Complex graphical objects

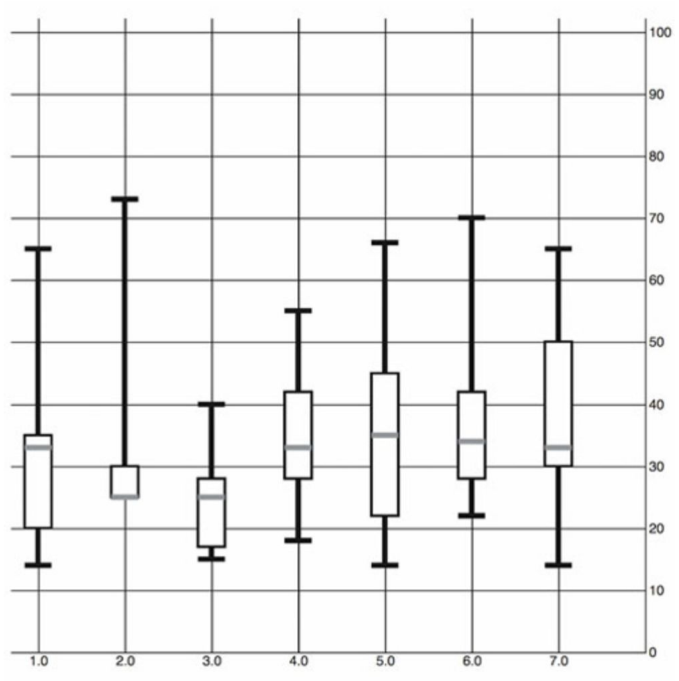
## Adding an axis using tickValues

```
const tickSize = 470
var xAxis = d3.axisBottom().scale(xScale)
.tickSize(-tickSize)                                   1
.tickValues([1,2,3,4,5,6,7]);                            2
    d3.select("svg").append("g")
      .attr("transform", `translate(0,${tickSize})`)
      .attr("id", "xAxisG").call(xAxis);                  3
    d3.select("#xAxisG > path.domain").style("display",
"none");     4
```

1 A negative tickSize draws the lines above the axis, but we need to make sure to offset the axis by the same value
2 Setting specific tickValues forces the axis to only show the corresponding values, which is useful when we want to override the automatic ticks created by the axis
3 Offsets the axis to correspond with our negative tickSize
4 We can hide this, because it has extra ticks on the ends that distract our readers

## Complex graphical objects

Our final boxplot chart. Each day now shows not only the median age of visitors but also the range of visiting ages, allowing for a more extensive examination of the demographics of site visitorship.



# Line charts and interpolations

## Callback function to draw a scatterplot from tweetdata

```
d3.csv("../data/tweetdata.csv", lineChart);
function lineChart(data) {

   const blue = "#5eaec5", green = "#92c463", orange = "#fe9a22"
   xScale = d3.scaleLinear().domain([1,10.5]).range([20,480])        1
   yScale = d3.scaleLinear().domain([0,35]).range([480,20])         1
   xAxis = d3.axisBottom()
      .scale(xScale)
      .tickSize(480)
      .tickValues([1,2,3,4,5,6,7,8,9,10])                           2

   d3.select("svg").append("g").attr("id", "xAxisG").call(xAxis)
   yAxis = d3.axisRight()
      .scale(yScale)
      .ticks(10)
      .tickSize(480)
   d3.select("svg").append("g").attr("id", "yAxisG").call(yAxis)
```

1 Our scales, as usual, have margins built in
2 Fixes the ticks of the x-axis to correspond to the days

## Line charts and interpolations

```
d3.select("svg").selectAll("circle.tweets")
    .data(data)                                        3
    .enter()
    .append("circle")
    .attr("class", "tweets")
    .attr("r", 5)
    .attr("cx", d => xScale(d.day))
    .attr("cy", d => yScale(d.tweets))                       3
    .style("fill", blue)

  d3.select("svg").selectAll("circle.retweets")
    .data(data)                                        3
    .enter()
    .append("circle")
    .attr("class", "retweets")
    .attr("r", 5)
    .attr("cx", d => xScale(d.day))
    .attr("cy", d => yScale(d.retweets))                      3
    .style("fill", green)
```

3 Each of these uses the same dataset but bases the y position on tweets, retweets, and favorites values, respectively

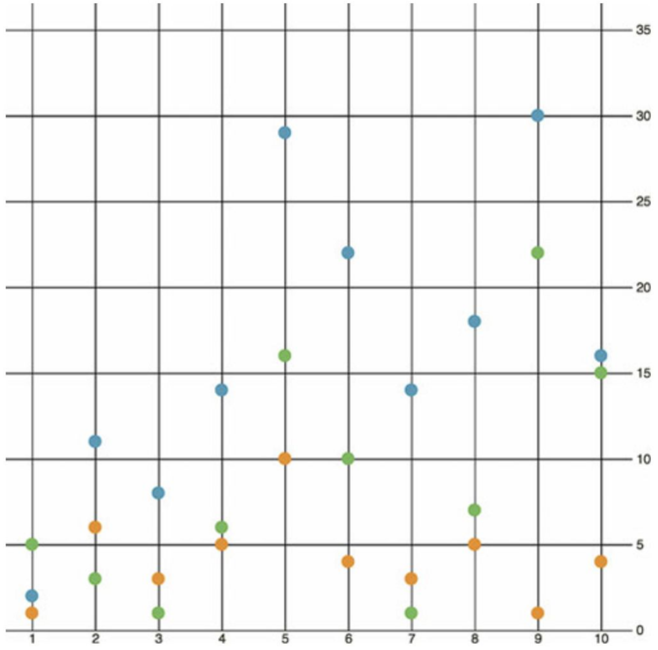
## Line charts and interpolations

```
   d3.select("svg").selectAll("circle.favorites")
      .data(data)                                         3
      .enter()
      .append("circle")
      .attr("class", "favorites")
      .attr("r", 5)
      .attr("cx", d => xScale(d.day))
      .attr("cy", d => yScale(d.favorites))                     3
      .style("fill", orange)
}
```

3 Each of these uses the same dataset but bases the y position on tweets, retweets, and favorites values, respectively

## Line charts and interpolations

A scatterplot showing the datapoints for 10 days of activity on Twitter, with the number of tweets in blue, the number of retweets in green, and the number of favorites in orange.



## Line charts and interpolations

If you have some gaps, you may consider drawing multipart lines. In that case you can use:

```
d3.line().defined(d => d.y !== null)
```

## Line charts and interpolations

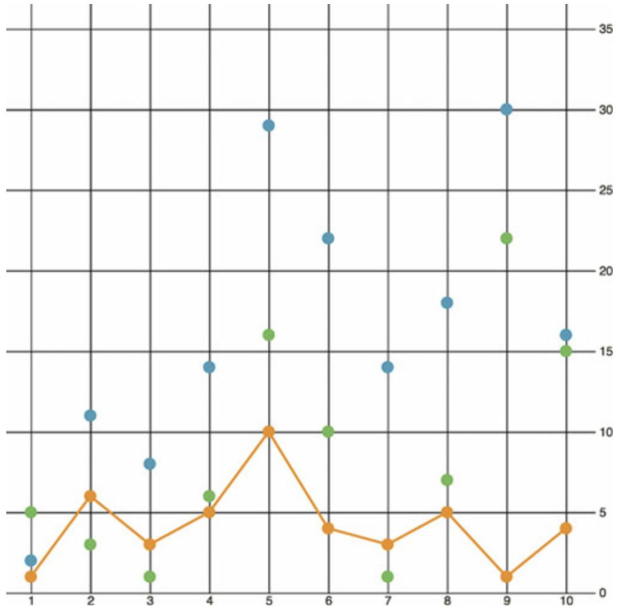
### New line generator code inside the callback function

```
var tweetLine = d3.line()
   .x(d => xScale(d.day))                1
   .y(d => yScale(d.tweets))              2
    d3.select("svg")
   .append("path")
   .attr("d", tweetLine(data))           3
   .attr("fill", "none")
   .attr("stroke", "#fe9a22")
   .attr("stroke-width", 2)
```

1 Defines an accessor for data like ours—in this case we take the day attribute and pass it to xScale first
2 This accessor does the same for the number of tweets
3 The appended path is drawn according to the generator with the loaded tweetdata variable passed to it

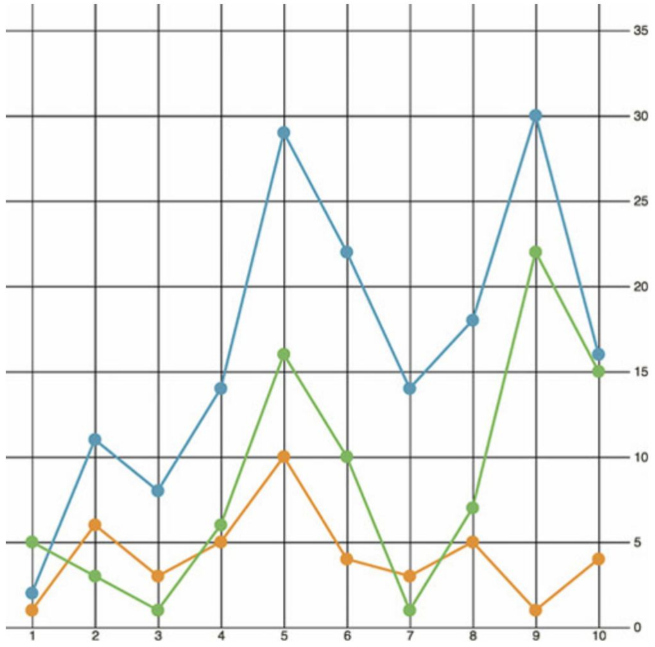
# Line charts and interpolations

The line generator takes the entire dataset and draws a line where the x,y position of every point on the canvas is based on its accessor. In this case, each point on the line corresponds to the day, and tweets are scaled to fit the x and y scales we created to display the data on the canvas.



## Line charts and interpolations

The dataset is first used to draw a set of circles, which creates the scatterplot from the beginning of this section. The dataset is then used three more times to draw each line.



## Line charts and interpolations

### Line generators for each tweetdata

```
const lambdaXScale = d => xScale(d.day)
1
   var tweetLine = d3.line()                        2
     .x(lambdaXScale)
     .y(d => yScale(d.tweets))
   var retweetLine = d3.line()
     .x(lambdaXScale)
     .y(d => yScale(d.retweets))                   3
   var favLine = d3.line()
     .x(lambdaXScale)
     .y(d => yScale(d.favorites))
```

1 Naming and reusing functions is also a good use of const.
2 A more efficient way to do this would be to define one line generator and then modify the .y() accessor on the fly as we call it for each line, but it's easier to see the functionality this way
3 Notice how only the y accessor is different between each line generator.

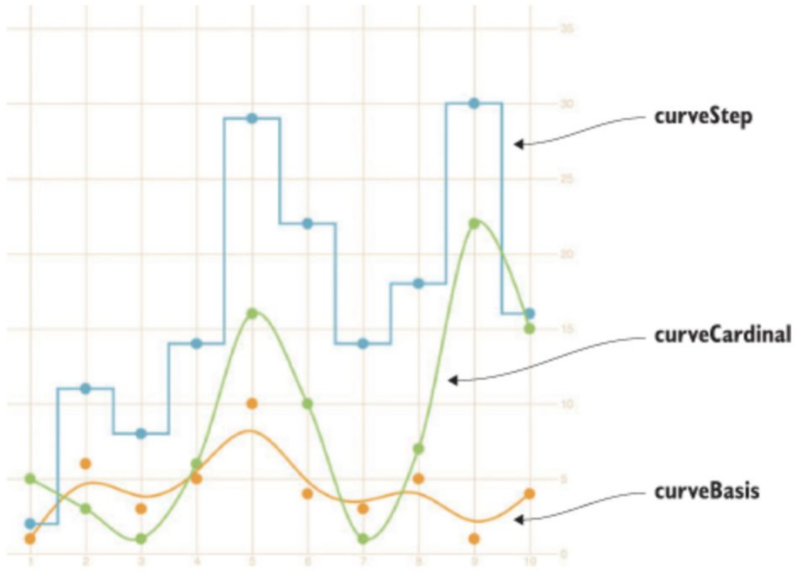
## Line charts and interpolations

```
d3.select("svg")
    .append("path")                          4
    .attr("d", tweetLine(data))
    .attr("fill", "none")
    .attr("stroke", blue)
    .attr("stroke-width", 2)
  d3.select("svg")
    .append("path")
    .attr("d", retweetLine(data))
    .attr("fill", "none")
    .attr("stroke", green)
    .attr("stroke-width", 2)
  d3.select("svg")
    .append("path")
    .attr("d", favLine(data))
    .attr("fill", "none")
    .attr("stroke", orange)
    .attr("stroke-width", 2)
```

4 Each line generator needs to be called by a corresponding new <path> element.

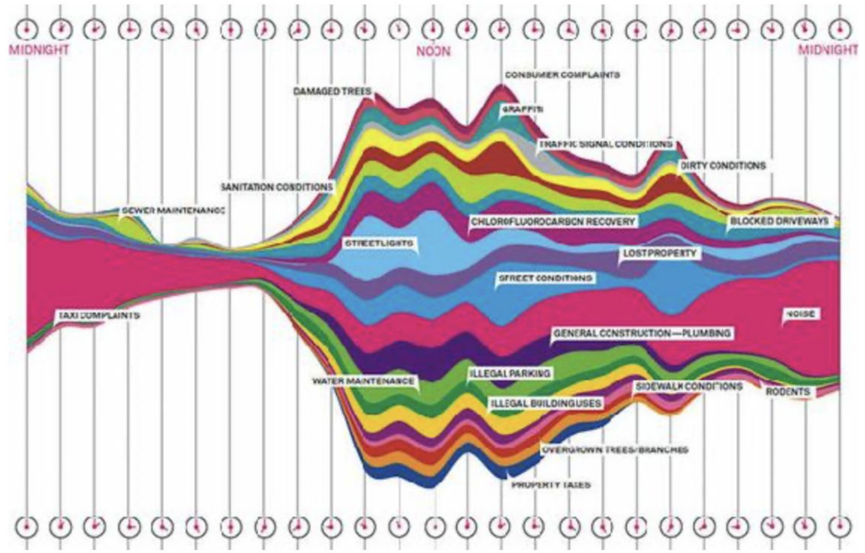
# Line charts and interpolations

Three common curve methods you'll see in charts. Orange is a "basis" interpolation that provides an organic curve averaged by the points (and therefore rarely touching them); a blue "step" interpolation changes the position of the line at right angles; and a green "cardinal" interpolation provides a curve that touches each sample point.



## Line charts and interpolations

Behold the glory of the streamgraph. Look on my works, ye mighty, and despair! (Figure by Pitch Interactive from Wired, November 1, 2010, www.wired.com/2010/11/ff_311_new_york/all/1.) (Wesley Grubbs/WIRED © Condé Nast)



## Line charts and interpolations

```
tweetLine.curve(d3.curveBasis)            1
retweetLine.curve(d3.curveStep)
favLine.curve(d3.curveCardinal)
```

1 We can add this code right after we create our line generators and before we call them to change the interpolate method, or we can set .curve() as we’re defining the generator

## Complex accessor functions

movies.csv

```
day,titanic,avatar,akira,frozen,deliverance,avengers
1,20,8,3,0,0,0
2,18,5,1,13,0,0
3,14,3,1,10,0,0
4,7,3,0,5,27,15
5,4,3,0,2,20,14
6,3,1,0,0,10,13
7,2,0,0,0,8,12
8,0,0,0,0,6,11
9,0,0,0,0,3,9
10,0,0,0,0,1,8
```

## Complex accessor functions

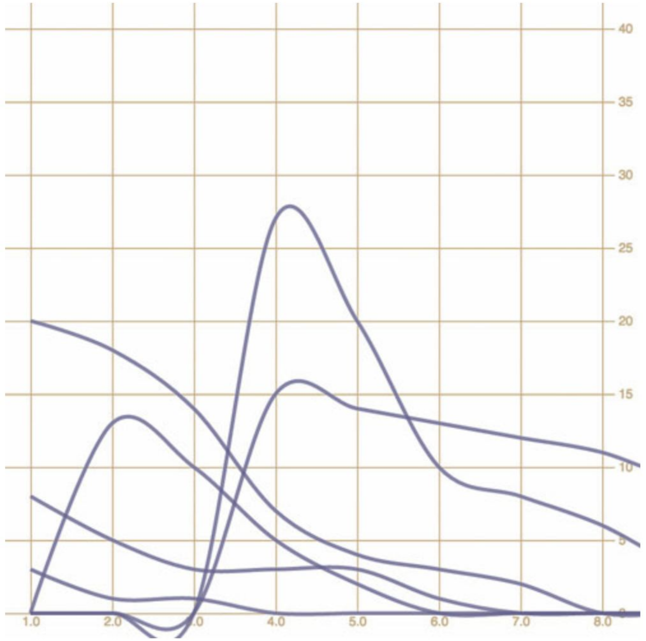
The callback function to draw movies.csv as a line chart

```
var xScale = d3.scaleLinear().domain([ 1, 8 ]).range([ 20, 470 ]);
var yScale = d3.scaleLinear().domain([ 0, 40 ]).range([ 480, 20 ]);
Object.keys(data[0]).forEach(key => {
   if (key != "day") {                              1
      var movieArea = d3.line()                       2
        .x(d => xScale(d.day))                       3
        .y(d => yScale(d[key]))                      4
        .curve(d3.curveCardinal);
      d3.select("svg")
        .append("path")
        .attr("id", key + "Area")
        .attr("d", movieArea(data))
        .style("fill", "none")
        .style("stroke", "#75739F")
        .style("opacity", .75)
   };
});
```

1 Iterates through our data attributes with forEach, where x is the name of each column from our data (“day”, “movie1”, “movie2”, and so on), which allows us to dynamically create and call generators
2 Instantiates a line generator for each movie
3 Every line uses the day column for its x value
4 Dynamically sets the y-accessor function of our line generator to grab the data from the appropriate movie for our y variable

## Complex accessor functions

Each movie column is drawn as a separate line. Notice how the “cardinal” interpolation creates a graphical artifact, where it seems several movies made negative money.



# Complex accessor functions

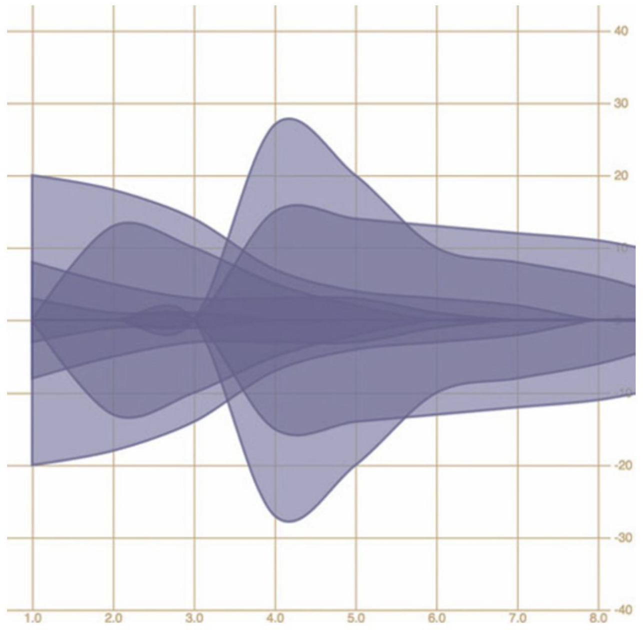
## Area accessors

```
var xScale = d3.scaleLinear().domain([ 1, 8 ]).range([ 20, 470 ]);
  var yScale = d3.scaleLinear().domain([ -40, 40 ]).range([ 480, 20 ]);
  Object.keys(data[0]).forEach(key => {
     if (key != "day") {
        var movieArea = d3.area()
        .x(d => xScale(d.day))
        .y0(d => yScale(d[key]))
        .y1(d => yScale(-d[key]))                   1
        .curve(d3.curveCardinal);
     d3.select("svg")
       .append("path")
       .style("id", key + "Area")
       .attr("d", movieArea(data))
       .attr("fill", "#75739F")
       .attr("stroke", "#75739F")
       .attr("stroke-width", 2)
       .style("stroke-opacity", .75)
       .style("fill-opacity", .5);
     };
})
```

1 This new accessor provides the ability to define where the bottom of the path is—in this case, we start by making the bottom equal to the inverse of the top, which mirrors the shape

## Complex accessor functions

By using an area generator and defining the bottom of the area as the inverse of the top, we can mirror our lines to create an area chart. Here they're drawn with semitransparent fills, so that we can see how they overlap.

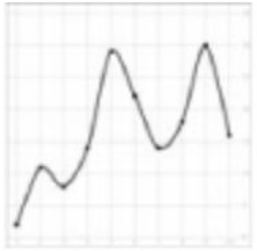


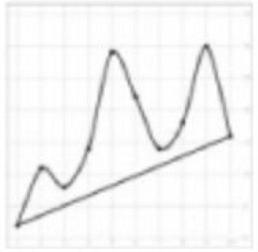
## Complex accessor functions

Closed path changes

```
d3.select("svg")
    .append("path")
    .attr("d",movieArea(data))
    .attr("fill", "none")
    .attr("stroke", "black")
    .attr("stroke-width", 3);
```

```
d3.select("svg")
   .append("path")
   .attr("d", movieArea(data) + "Z")
   .attr("fill", "none")
   .attr("stroke", "black")
   .attr("stroke-width", 3);
```





## Complex accessor functions

```
d3.select("svg")
    .append("path")
    .attr("d", movieArea(data))
    .attr("fill", "none")
    .attr("stroke", "black")
    .attr("stroke-width", 3);
```

```
d3.select("svg")
    .append("path")
    .attr("d", movieArea(data) + "Z")
    .attr("fill", "gray")
    .attr("stroke", "black")
    .attr("stroke-width", 3);
```

# Complex accessor functions

## Callback function for drawing stacked areas

```
var fillScale = d3.scaleOrdinal()
.domain(["titanic", "avatar", "akira", "frozen", "deliverance", "avengers"])
.range(["#fcd88a", "#cf7c1c", "#93c464", "#75734F", "#5eafc6", "#41a368"]) 1
  var xScale = d3.scaleLinear().domain([ 1, 10 ]).range([ 20, 470 ]);
  var yScale = d3.scaleLinear().domain([0, 55]).range([ 480, 20 ])

  Object.keys(data[0]).forEach(key => {
     if (key != "day") {                                    2
        var movieArea = d3.area()                               3
           .x(d => xScale(d.day))
           .y0(d => yScale(simpleStacking(d, key) - d[key]))
           .y1(d => yScale(simpleStacking(d, key)))
           .curve(d3.curveBasis)
        d3.select("svg")                                    4
           .append("path")
           .style("id", key + "Area")
           .attr("d", movieArea(data))
           .attr("fill", fillScale(key))
           .attr("stroke", "black")
           .attr("stroke-width", 1)
     }
  })
```

1 Creates a color ramp that corresponds to the six different movies
2 We won’t draw a line for the day value of each object, because this is what provides us with our x coordinate
3 A d3.area() generator for each iteration through the object that corresponds to one of our movies using the day value for the x coordinate, but iterating through the values for each movie for the y coordinates
4 Draws a path using the current constructor. We’ll have one for each attribute not named “day”. Give it a unique ID based on which attribute we’re drawing an area for. Fill the area with a color based on the color ramp we built.

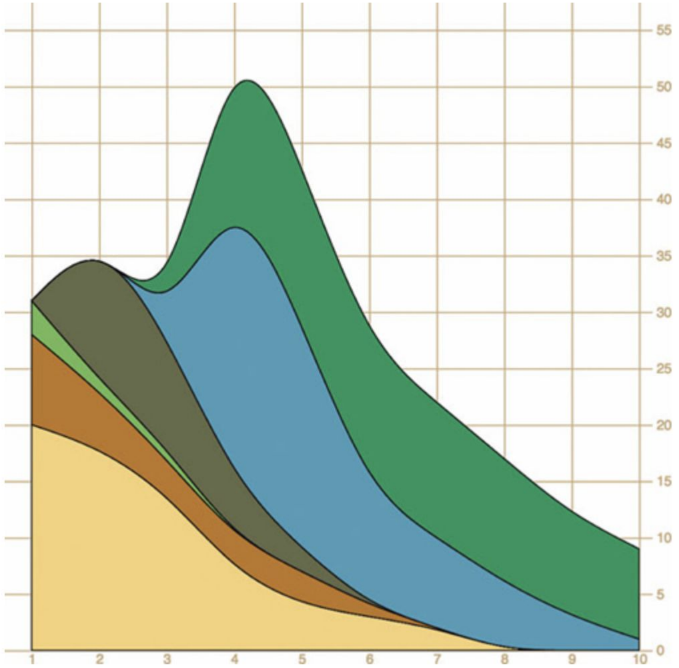
## Complex accessor functions

```
   function simpleStacking( lineData, lineKey) {                    5
         var newHeight = 0
         Object.keys(lineData).every(key => {
           if (key !== "day") {
             newHeight += parseInt(lineData[key]);
             if (key === lineKey) {
               return false
             }
           }
               return true
      })
      return newHeight
}
```

5 This function takes the incoming bound data and the name of the attribute and loops through the incoming data, adding each value until it reaches the current named attribute. As a result, it returns the total value for every movie during this day up to the movie we’ve sent.

# Complex accessor functions

Our stacked area code represents a movie by drawing an area, where the bottom of that area equals the total amount of money made by any movies drawn earlier for that day.



## Using third-party D3 modules to create legends

- We will use another piece of software
- Created by Susie Lu

```
<script
src="https://cdnjs.cloudflare.com/ajax/libs/d3-legend/2.21.0/d3-legend.min.js"></s
cript>
```

## Using third-party D3 modules to create legends

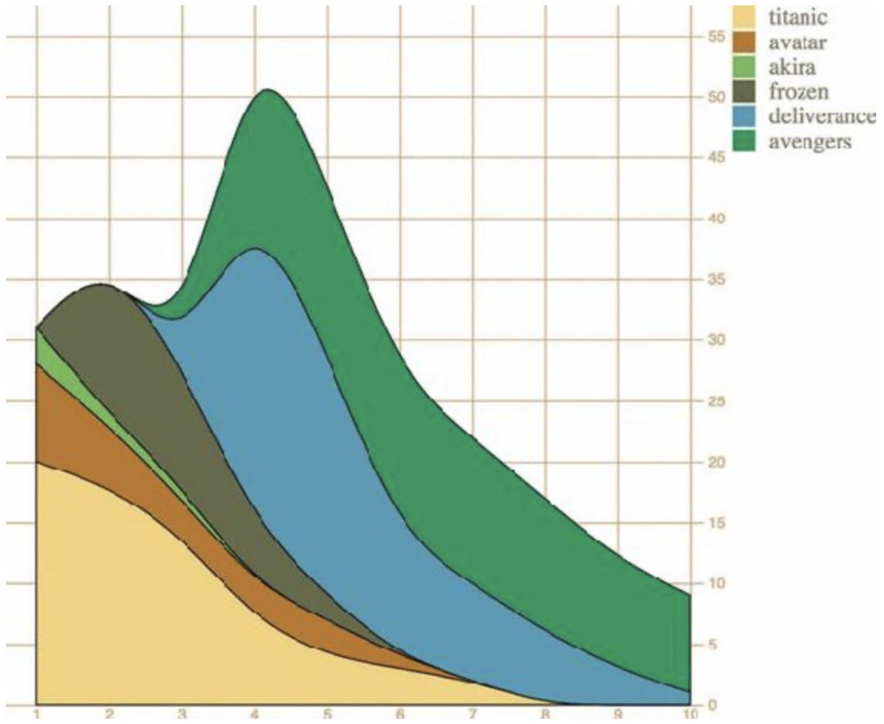
Adding a color legend

```
var legendA = d3.legendColor().scale(fillScale)
d3.select("svg")
  .style("width", "1000px")

d3.select("svg")
  .append("g")
  .attr("transform", "translate(500, 0)")
  .call(legendA)
```

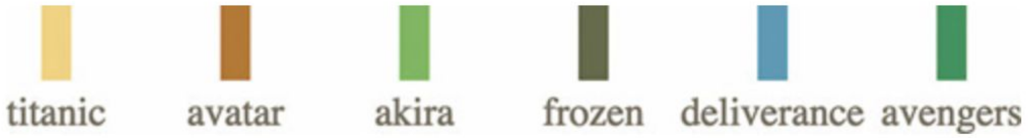
## Using third-party D3 modules to create legends

Our stacked chart with a legend telling the reader which color corresponds to which movie



## Using third-party D3 modules to create legends

A horizontal oriented colorLegend from d3-svg-legend rendered with custom settings for shapePadding, shapeWidth, and shapeHeight.



## Using third-party D3 modules to create legends

Adjusted legend settings

```
legendA.orient("horizontal")
    .shapePadding(60)
    .shapeWidth(12)
.shapeHeight(30)
```