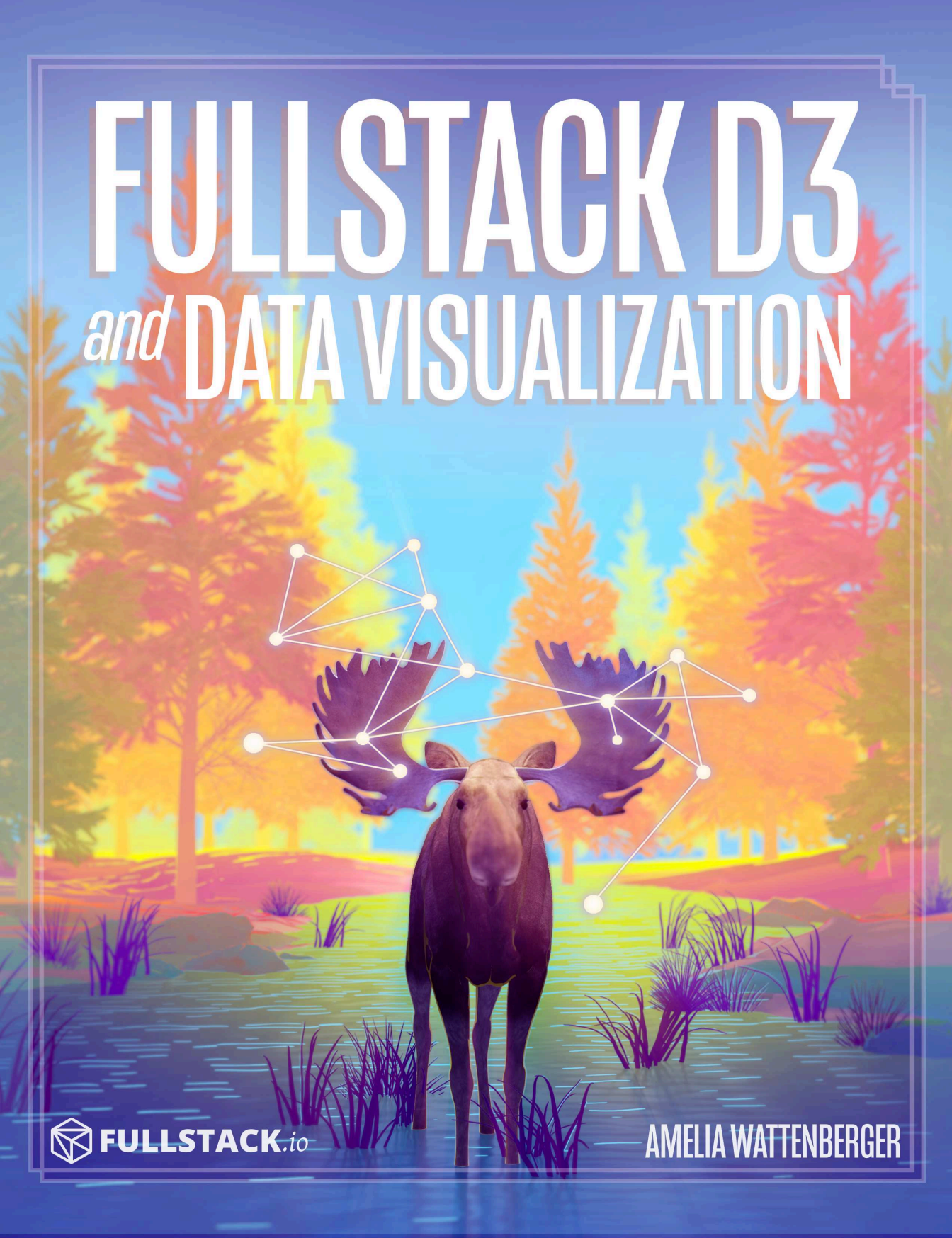
FULLSTACK D3 and DATA VISUALIZATION



FULLSTACK.io

AMELIA WATTENBERGER

Fullstack Data Visualization with D3

Build Beautiful Data Visualizations and Dashboards with D3

Written by Amelia Wattenberger
Edited by Nate Murray

© 2019 Fullstack.io

All rights reserved. No portion of the book manuscript may be reproduced, stored in a retrieval system, or transmitted in any form or by any means beyond the number of purchased copies, except for a single backup or archival copy. The code may be used freely in your projects, commercial or otherwise.

The authors and publisher have taken care in preparation of this book, but make no expressed or implied warranty of any kind and assume no responsibility for errors or omissions. No liability is assumed for incidental or consequential damagers in connection with or arising out of the use of the information or programs container herein.

Published by Fullstack.io.

FULLSTACK.io

Contents

Book Revision

- Revision 1 - 2019-05-12
- Revision 2 - 2019-05-14
- Revision 3 - 2019-05-17
- Revision 4 - 2019-05-30
- Revision 5 - 2019-06-03
- Revision 6 - 2019-06-06
- Revision 7 - 2019-06-14
- Revision 8 - 2019-07-04
- Revision 9 - 2019-08-28
- Revision 10 - 2019-09-29
- Revision 11 - 2019-10-11
- Revision 12 - 2019-12-09
- Revision 13 - 2020-01-08
- Revision 14 - 2020-03-11
- Revision 15 - 2020-04-01

Bug Reports

If you'd like to report any bugs, typos, or suggestions just email us at: **us@fullstack.io**.

Be notified of updates via Twitter

If you'd like to be notified of updates to the book on Twitter, follow us at @fullstackio[1].

We'd love to hear from you!

Did you like the book? Did you find it helpful? We'd love to add your face to our list of testimonials on the website! Email us at: us@fullstack.io[2].

[1] https://twitter.com/fullstackio
[2] mailto:us@fullstack.io

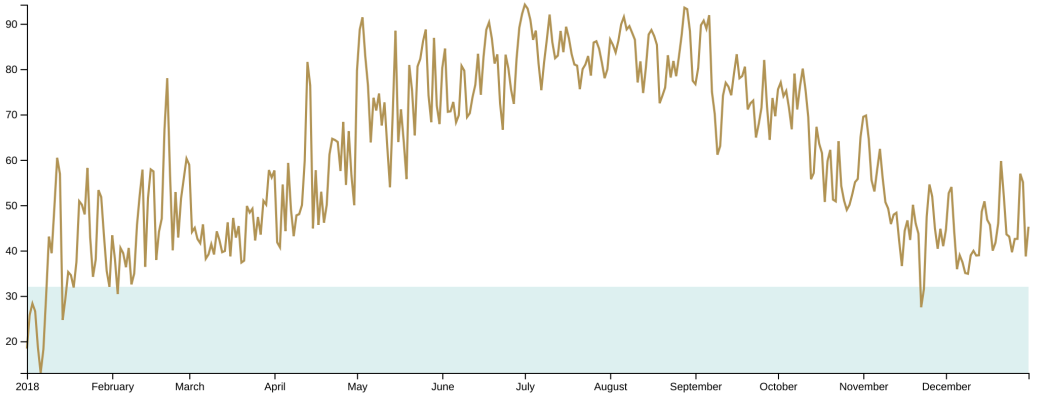
Making Your First Chart

Many books begin by talking about theory and abstract concepts. This is not one of those books. We'll dig in and create real charts right away! Once you're comfortable making your own charts, we'll discuss how to integrate into websites, data visualization fundamentals, and practical tips for chart design, along with other goodies.

Getting started

To start, let's make a line chart. Line charts are a great starting place because of their popularity, but also because of their simplicity.

In this chapter, **we'll create a line chart that plots the daily temperature**. Here's what our line chart will look like when we're finished:



Finished line graph

In the tutorial below, don't worry too much about the details! We're just going to get our hands dirty and write the code to build this line chart. This will give us a good foundation to dive deeper into each concept in Chapters 2 and 3, in which we'll create more complex charts.

The dataset we'll be analyzing contains 365 days of daily weather metrics. To make it easy, we've provided a JSON file with this data in the code download folder named `nyc_weather_data.json`. This file includes 2018 data for New York City.

We recommend that you create a dataset for your own location to keep the data tangible, plus you'll discover new insights about where you live! Refer to **Appendix A** for instructions — it should only take a few minutes and the charts we make together will be uniquely yours.

If you're using the provided dataset, rename the file to `my_weather_data.json`. This way, our code examples will know where to find the weather data.

Let's get our webpage up and running. Find the `index.html` file and open it in your browser. The url will start with `file:///`. This is a very simple webpage — we're rendering one element and loading two javascript files.

code/01-making-your-first-chart/completed/index.html

```
7         <div id="wrapper"></div>
8
9         <script src="./../../d3.v5.js"></script>
10        <script src="./chart.js"></script>
```

The page should be blank except for one `div` with an `id` of `wrapper` — this is where our chart will live.

The first script that we're loading is d3.js — this will give us access to the entire library.

> At this point, we want access to the whole library, but d3.js is made up of at least thirty modules. Later, we'll discuss how to import only the necessary modules to keep your build lean.

Next, our `index.html` file loads the javascript file in which we'll write our chart code: `chart.js`.

Let's open up the `01-making-your-first-chart/draft/chart.js` file in a code editor and dig in.

> If you don't already have a code editor, any program that lets you open a file and edit the text will do! I personally use **Visual Studio Code**[a], which I recommend — it's straightforward enough for beginners, but has many configuration options and extensions for power users.
>
> [a] https://code.visualstudio.com/

We don't have much text in here to start with.

```
async function drawLineChart() {
    // write your code here

}

drawLineChart()
```

The only thing happening so far is that we're defining a function named `drawLineChart()` and running it.

Loading the weather dataset

The first step to visualizing any dataset is understanding its structure. To get a good look at our data, we need to import it into our webpage. To do this, we will load the JSON file that holds our data.

D3.js has methods for fetching and parsing files of different formats in the **d3-fetch**[3] module, for example, `d3.csv()`, `d3.json()`, and `d3.xml()`. Since we're working with a JSON file, we want to pass our file path to `d3.json()`.

Let's create a new variable named `dataset` and load it up with the contents of our JSON file.

[3] https://github.com/d3/d3-fetch

code/01-making-your-first-chart/completed/chart.js

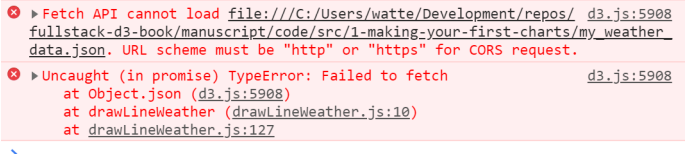
```
4  const dataset = await d3.json("./../../my_weather_data.json")
```

> `await` is a JavaScript keyword that will **pause the execution of a function until a Promise is resolved**. This will only work within an async function — note that the `drawLineChart()` function declaration is preceded by the keyword `async`.

> Don't be overwhelmed by the words **Promise**, **async**, or **await** — this just means that any code (within the function) after `await d3.json("./../../my_weather_data.json")` will wait to run until `dataset` is defined. If you're curious and want to learn more, here is a great resource on Promises in JavaScript[a].
>
> [a] https://www.youtube.com/watch?v=QO4NXhWo_NM

If you see a `SyntaxError: Unexpected end of JSON input` error message, check your `my_weather_data.json` file. It might be empty or corrupted. If so, re-generate your custom data or copy the `nyc_weather_data.json` file.

Now when we load our webpage we should get a CORS error in the console.



CORS error

CORS is short for Cross-Origin Resource Sharing, a mechanism used to restrict requests to another domain. **We'll need to start a server to get around this safety restriction**.

Getting a server running

Thankfully, there are simple ways to spin up a simple static server. We'll walk through two different ways (**node.js** or **python**) — choose whichever one you're more comfortable with. **Note that only one of these options is necessary**.

a. **node.js**

*I would recommend using this method because it has **live reload** built in, meaning that our page will update when we save our changes. No page refresh necessary!*

If you don't have **node.js** installed, take a minute to install it (instructions here[4]). You can check whether or not **node.js** is already installed by using the `node -v` command in your terminal — if it responds with a version number, you're good to go! **node.js** should also come with **npm**, which is short for *Node Package Manager*

Once **node.js** and **npm** are installed, run the following command in your terminal.

```
npm install -g live-server
```

This will install **live-server**[5], a simple static server that has live reload built-in. To start your server, run `live-server` in your terminal from the root `/code` folder — it will even open a new browser window for you!

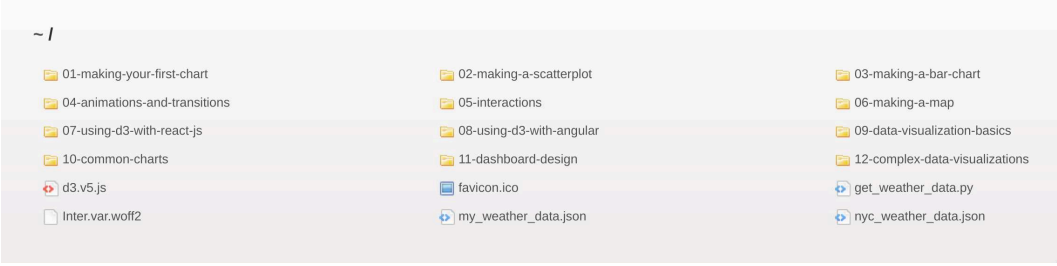
a. **python**

If you have **python** (version 3) installed already, you can use the Python 3 http server instead. Start it up by running the command `python -m http.server 8080` in your terminal from the root `/code` folder.

The particular server doesn't matter — the key idea is that if you want to load a file from JavaScript, you need to do it from a webserver, and these tools are an easy solution for a development environment. Make sure that you are in the root `/code` folder when you start either server.

[4] https://docs.npmjs.com/downloading-and-installing-node-js-and-npm
[5] https://github.com/tapio/live-server

Now we should have a server on port 8080. Load localhost:8080[6] in your web browser and you'll see a directory of code for each chapter, which looks something like this:



Directory screenshot

Click on **01-making-your-first-chart** or go to http://localhost:8080/01-making-your-first-chart/draft/[7] to load this chapter's `index.html` file.

> For all of our code examples, there will be a finished version in a sibling `/completed` folder — in this chapter, look at `/code/01-making-your-first-chart/completed/` if you want a reference. Or view to completed chart at http://localhost:8080/01-making-your-first-chart/completed/.

We'll still see a blank page since we haven't drawn anything on our page yet, but that error should be gone!

Looking at our data

Going back to our code, let's log our dataset to the console. We can do that by adding the following line of code right after we create our `dataset` file.

```
console.log(dataset)
```

We can see that our dataset is array of objects, with one object per day.

[6] http://localhost:8080
[7] http://localhost:8080/01-making-your-first-chart/draft/

Since each day seems to have the same structure, let's delete the last line and instead log a single data point to get a clearer view.

> We can use `console.table()` here, which is a great function for looking at array or object values — as long as there aren't too many!

```
console.table(dataset[0])
```

chart.js:6

(index)	Value
time	1514782800
summary	"Clear throughout the day."
icon	"clear-day"
sunriseTime	1514809280
sunsetTime	1514842810
moonPhase	0.48
precipIntensity	0
precipIntensityMax	0
precipProbability	0
temperatureHigh	18.39
temperatureHighTime	1514836800
temperatureLow	12.23
temperatureLowTime	1514894400
apparentTemperatureHigh	17.29
apparentTemperatureHighTime	1514844000
apparentTemperatureLow	4.51
apparentTemperatureLowTime	1514887200
dewPoint	-1.67
humidity	0.54
pressure	1028.26
windSpeed	4.16
windGust	13.98
windGustTime	1514829600
windBearing	309
cloudCover	0.02
uvIndex	2
uvIndexTime	1514822400
visibility	10
temperatureMin	6.17
temperatureMinTime	1514808000
temperatureMax	18.39
temperatureMaxTime	1514836800
apparentTemperatureMin	-2.19
apparentTemperatureMinTime	1514808000
apparentTemperatureMax	17.29
apparentTemperatureMaxTime	1514844000
date	"2018-01-01"

▸ Object

Our dataset

We have lots of information for each day! We can see metadata (`date`, `time`, `summary`) and details about that day's weather (`cloudCover`, `sunriseTime`, `temperatureMax`, etc). If you want to read more about each metric, check out The Dark Sky API docs[8].

[8] https://darksky.net/dev/docs#data-point

Setting up our line chart

Let's dig in by looking at `temperatureMax` over time. Our timeline will have two axes:

- a y axis (vertical) on the left comprised of max temperature values
- an x axis (horizontal) on the bottom comprised of dates

To grab the correct metrics from each data point, we'll need *accessor* functions. **Accessor functions convert a single data point into the metric value**.

Lets try it out by creating a `yAccessor` function that will take a data point and return the max temperature.

> If you think of a **dataset** as a table, a **data point** would be a row in that table. In this case, a **data point** represents an item in our `dataset` array: an object that holds the weather data for one day.

We will use `yAccessor` for plotting points on the y axis.

Looking at the data point in our console, we can see that a day's max temperature is located on the object's `temperatureMax` key. To access this value, our `yAccessor` function looks like this:

code/01-making-your-first-chart/completed/chart.js

```
6   const yAccessor = d => d.temperatureMax
```

Next, we'll need an `xAccessor` function that will return a point's date, which we will use for plotting points on the x axis.

```
const xAccessor = d => d.date
```

But look closer at the data point `date` value - notice that it is a *string* (eg. `"2018-12-25"`). Unfortunately, this string won't make sense on our x axis. How could we know how far `"2018-12-25"` is from `"2018-12-29"`?.

We need to convert the string into a JavaScript Date, which is an object that represents a single point in time. Thankfully, d3 has a **d3-time-format**[9] module with methods for parsing and formatting dates.

The `d3.timeParse()` method...

- takes a string specifying a date format, and
- outputs a function that will parse dates of that format.

For example, `d3.timeParse("%Y")` will parse a string with just a year (eg. `"2018"`).

Let's create a date parser function and use it to transform our date strings into date objects:

code/01-making-your-first-chart/completed/chart.js

```
7  const dateParser = d3.timeParse("%Y-%m-%d")
8  const xAccessor = d => dateParser(d.date)
```

Great! Now when we call `xAccessor(dataset[0])`, we'll get the first day's date.

> If you look up d3 examples, you won't necessarily see accessor functions used. When I first started learning d3, I never thought about using them. Since then, I've learned my lesson and paid the price of painstakingly picking through old code and updating individual lines. I want to save you that time so you can spend it making even more wonderful charts.

Defining accessor functions might seem like unnecessary overhead right now, especially with this simple example. However, creating a separate function to read the values from our data points helps us in a few ways.

[9] https://github.com/d3/d3-time-format

- **Easy changes**: every chart is likely to change at least once — whether that change is due to business requirements, design, or data structure. These changing requirements are especially prevalent when creating dashboards with dynamic data, where you might need to handle a new edge case two months later. Having the accessor functions in one place at the top of a chart file makes them easy to update throughout the chart.
- **Documentation**: having these functions at the top of a file can give you a quick reminder of what metrics the chart is plotting and the structure of the data.
- **Framing**: sitting down with the data and planning what metrics we'll need to access is a great way to start making a chart. It's tempting to rush in, then two hours later realize that another type of chart would be better suited to the data.

Now that we know how to access our dataset, we need to **prepare to draw our chart**.

Drawing our chart

When drawing a chart, there are two containers whose dimensions we need to define: the wrapper and the bounds.

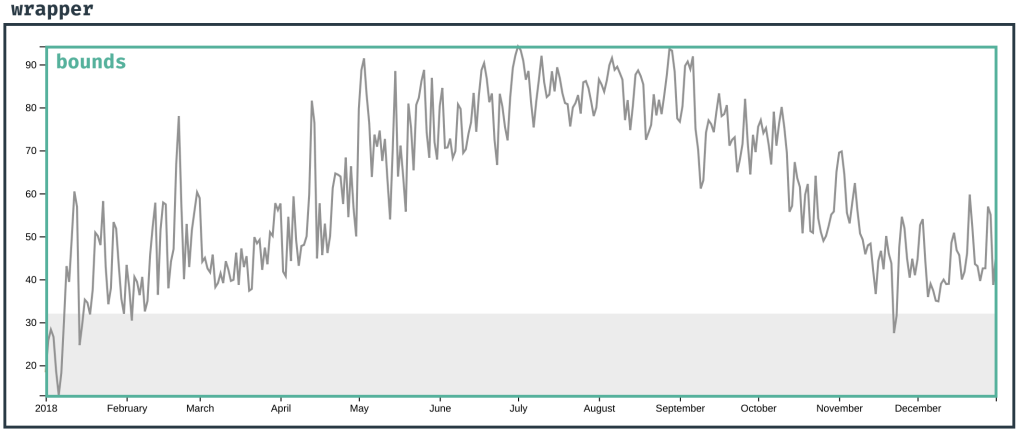


Chart dimensions

The **wrapper** contains the entire chart: the data elements, the axes, the labels, etc. Every SVG element will be contained inside here.

The **bounds** contain all of our data elements: in this case, our line.

This distinction will help us separate the amount of space we need for extraneous elements (axes, labels), and let us focus on our main task: plotting our data. One reason this is so important to define up front is the inconsistent and unfamiliar way SVG elements are sized.

When adding a chart to a webpage, we start with the amount of space we have available for the chart. Then we decide how much space we need for the margins, which will accommodate the chart axes and labels. What's left is how much space we have for our data elements.

We will rarely have the option to decide how large our timeline is and then build up from there. Our charts will need to be accommodating of window sizes, surrounding text, and more.

> While **wrapper** and **bounds** isn't terminology that you'll see in widespread use, it will be helpful for reference in this book. Defining these concepts also helps with thinking about chart dimensions and remembering to make space for your axes.

Let's define a dimensions object that will contain the size of the wrapper and the margins. We'll have one margin defined for each side of the chart: top, right, bottom, and left. For consistency, we'll mimic the order used for CSS properties.

code/01-making-your-first-chart/completed/chart.js

```
12  let dimensions = {
13    width: window.innerWidth * 0.9,
14    height: 400,
15    margin: {
16      top: 15,
17      right: 15,
18      bottom: 40,
19      left: 60,
20    },
21  }
```

We want a small `top` and `right` margin to give the chart some space. The line or the y axis might overflow the chart bounds. We'll want a larger `bottom` and `left` margin to create room for our axes.

Let's compute the size of our **bounds** and add that to our `dimensions` object.

code/01-making-your-first-chart/completed/chart.js

```
22  dimensions.boundedWidth = dimensions.width
23    - dimensions.margin.left
24    - dimensions.margin.right
25  dimensions.boundedHeight = dimensions.height
26    - dimensions.margin.top
27    - dimensions.margin.bottom
```

Creating our workspace

Now we're set up and ready to start updating our webpage!

To add elements to our page, we'll need to specify an existing element that we want to append to.

Remember the `#wrapper` element already populated in `index.html`? One of d3's modules, **d3-selection**[10], has helper functions to select from and modify the DOM.

We can use `d3.select()` , which accepts a CSS-selector-like string and returns the first matching DOM element (if any). If you're unfamiliar with CSS selector syntax, there are three main types of selectors:

- you can select all elements with a class name (`.class`)
- you can select all elements with an id (`#id`), or
- you can select all elements of a specific node type (`type`).

[10] https://github.com/d3/d3-selection

> If you've ever used jQuery or written CSS selectors, these selector strings will be familiar.

```
const wrapper = d3.select("#wrapper")
```

Let's log our new `wrapper` variable to the console to see what it looks like.

```
console.log(wrapper)
```

We can see that it's a d3 selection object, with `_groups` and `_parents` keys.

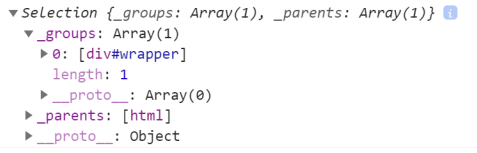


chart selection

d3 selection objects **are a subclass of Array**. They have a lot of great methods that we'll explore in depth later - what's important to us right now is the `_groups` list that contains our `#wrapper` div.

Adding an SVG element

Our `wrapper` object also has methods for manipulating the linked DOM element — let's use its `append` method to add a new SVG element.

```
const wrapper = d3.select("#wrapper")
const svg = wrapper.append("svg")
```

If we log `svg` to the console, we'll see that it looks like our `wrapper` object. However, if we expand the `_groups` key, we'll see that the linked element is our new `<svg>` element.

One trick to make sure we're grabbing the correct element is to hover the logged DOM element. If we expand the `_groups` object and hover over the `<svg>` element, the browser will highlight the corresponding DOM element on the webpage.

svg selection with hover

On hover, the browser will also show the element's size: 300px by 150px. This is the default size for SVG elements in Google Chrome, but it will vary between browsers and even browser versions. SVG elements don't scale the way most DOM elements do — there are many rules that will be unfamiliar to an experienced web developer.

To maintain control, let's tell our `<svg>` element what size we want it to be.

d3 selection objects have an `.attr()` method that will add or replace an attribute on the selected DOM element. The first argument is the attribute name and the second argument is the value.

The value argument to `.attr()` can either be a constant, which is all we need right now, or a function, which we'll cover later.

```
const wrapper = d3.select("#wrapper")
const svg = wrapper.append("svg")
svg.attr("width", dimensions.width)
svg.attr("height", dimensions.height)
```

Most `d3-selection` methods will return a selection object.

- any method that selects or creates a new object will return the new selection

- any method that manipulates the current selection will return the same selection

This allows us to keep our code concise by chaining when we're using multiple methods. For example, we can rewrite the above code as:

```
const wrapper = d3.select("#wrapper")
const svg = wrapper.append("svg")
    .attr("width", dimensions.width)
    .attr("height", dimensions.height)
```

In this book, we'll follow the common d3 convention of using 4 space indents for methods that return the same selection. This will make it easy to spot when our selection changes.

Since we're not going to re-use the `svg` variable, we can rewrite the above code as:

code/01-making-your-first-chart/completed/chart.js

```
31  const wrapper = d3.select("#wrapper")
32    .append("svg")
33      .attr("width", dimensions.width)
34      .attr("height", dimensions.height)
```

When we refresh our `index.html` page, we should now see that our `<svg>` element is the correct size. Great!

Creating our bounding box

Our SVG element is the size we wanted, but we want our chart to respect the margins we specified.

Let's create a **group** that shifts its contents to respect the top and left margins so we can deal with those in one place.

Any elements inside of an `<svg>` have to be SVG elements (with the exception of `<foreignObject>` which is fiddly to work with). Since we'll be inserting new chart elements inside of our `<svg>`, we'll need to use SVG elements for the rest of the chart.

The `<g>` SVG element is not visible on its own, but is used to group other elements. Think of it as the `<div>` of SVG — a wrapper for other elements. We can draw our chart inside of a `<g>` element and shift it all at once using the CSS `transform` property.

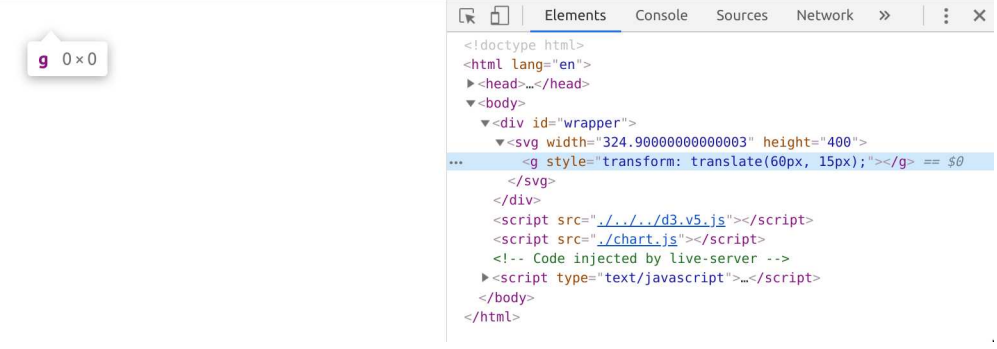
d3 selection objects have a `.style()` method for adding and modifying CSS styles. The `.style()` method is invoked similarly to `.attr()` and takes a key-value pair as its first and second arguments. Let's use `.style()` to shift our bounds.

code/01-making-your-first-chart/completed/chart.js

```
36  const bounds = wrapper.append("g")
37      .style("transform", `translate(${
38        dimensions.margin.left
39      }px, ${
40        dimensions.margin.top
41      }px)`)
```

> We're using backticks (`) instead of quotes (' or ") to create our `translate` string. This lets us use ES6 string interpolation — if you're unfamiliar, read more here[a].
>
> [a] https://babeljs.io/docs/en/learn/#template-strings

If we look at our **Elements** panel, we can see our new `<g>` element.



g element

We can see that the `<g>` element size is 0px by 0px — instead of taking a `width` or `height` attribute, a `<g>` element will expand to fit its contents. When we start drawing our chart, we'll see this in action.

Creating our scales

Let's get back to the data.

On our y axis, we want to plot the max temperature for every day.

Before we draw our chart, we need to decide what temperatures we want to visualize. Do we need to plot temperatures over 1,000°F or under 0°F? We could hard-code a standard set of temperatures, but that range could be too large (making the data hard to see), or it could be too small or offset (cutting off the data). Instead, let's use the actual range by finding the lowest and highest temperatures in our dataset.

> We've all seen over-dramatized timelines with a huge drop, only to realize that the change is relatively small. When defining an axis, we'll often want to start at 0 to show scale. We'll go over this more when we talk about types of data.

As an example, let's grab a sample day's data — say it has a maximum temperature of 55°F. We *could* draw our point 55 pixels above the bottom of the chart, but that won't scale with our `boundedHeight`.

Additionally, if our lowest temperature is below 0 we would have to plot that value below the chart! Our y axis wouldn't be able to handle all of our temperature values.

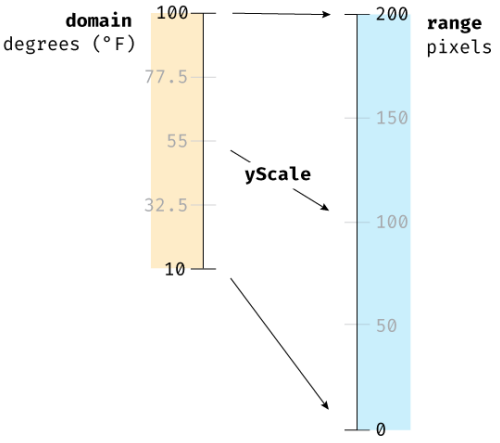
To plot the max temperature values in the correct spot, **we need to convert them into pixel space**.

d3's **d3-scale**[11] module can create different types of scales. A scale is **a function that converts values between two domains**.

For our y axis, we want to convert values from the **temperature domain** to the **pixel domain**. If our chart needs to handle temperatures from 10°F to 100°F, a day with a max of 55°F will be halfway up the y axis.

[11] https://github.com/d3/d3-scale

Let's create a scale that converts those degrees into a y value. If our y axis is 200px tall, the y scale should convert 55°F into 100, the halfway point on the y axis.



Our dataset

d3-scale[12] can handle many different types of scales - in this case, we want to use `d3.scaleLinear()` because our y axis values will be numbers that increase linearly. To create a new scale, we need to create an instance of `d3.scaleLinear()`.

code/01-making-your-first-chart/completed/chart.js

```
45  const yScale = d3.scaleLinear()
```

Our scale needs two pieces of information:

- the **domain**: the minimum and maximum input values
- the **range**: the minimum and maximum output values

Let's start with the **domain**. We'll need to create an array of the smallest and largest numbers our y axis will need to handle — in this case the lowest and highest max temperature in our dataset.

The **d3-array**[13] module has a `d3.extent()` method for grabbing those numbers. `d3.extent()` takes two parameters:

[12] https://github.com/d3/d3-scale
[13] https://github.com/d3/d3-array

1. an array of data points
2. an accessor function which defaults to an identity function (`d => d`)

Let's test this out by logging `d3.extent(dataset, yAccessor)` to the console. The output should be an array of two values: the minimum and maximum temperature in our dataset. Perfect!

Let's plug that into our scale's domain:

code/01-making-your-first-chart/completed/chart.js

```
45  const yScale = d3.scaleLinear()
46    .domain(d3.extent(dataset, yAccessor))
```

Next, we need to specify the **range**. As a reminder, the **range** is the highest and lowest number we want our scale to output — in this case, the maximum & minimum number of pixels our point will be from the x axis. We want to use our `boundedHeight` to stay within our margins. Remember, SVG y-values count from top to bottom so we want our range to start at the top.

code/01-making-your-first-chart/completed/chart.js

```
45  const yScale = d3.scaleLinear()
46    .domain(d3.extent(dataset, yAccessor))
47    .range([dimensions.boundedHeight, 0])
```

We just made our first scale function! Let's test it by logging some values to the console. At what y value is the freezing point on our chart?

```
console.log(yScale(32))
```

The outputted number should tell us how far away the freezing point will be from the bottom of the y axis.

> If this returns a negative number, congratulations! You live in a lovely, warm place. Try replacing it with a number that "feels like freezing" to you. Or highlight another temperature that's meaningful to you.

Let's visualize this threshold by adding a rectangle covering all temperatures below freezing. The SVG `<rect>` element can do exactly that. We just need to give it four attributes: `x`, `y`, `width`, and `height`.

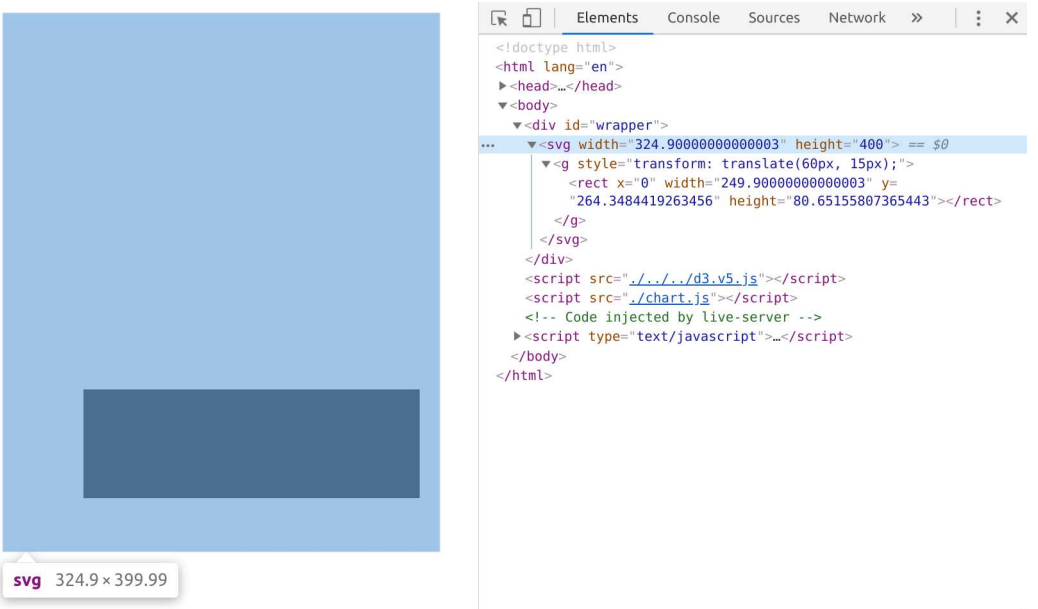
For more information about SVG elements, the MDN docs[14] are a wonderful resource: here is the page for `<rect>`[15].

code/01-making-your-first-chart/completed/chart.js

```
49  const freezingTemperaturePlacement = yScale(32)
50  const freezingTemperatures = bounds.append("rect")
51      .attr("x", 0)
52      .attr("width", dimensions.boundedWidth)
53      .attr("y", freezingTemperaturePlacement)
54      .attr("height", dimensions.boundedHeight
55        - freezingTemperaturePlacement)
```

Now we can see a black rectangle spanning the width of our bounds.

[14] https://developer.mozilla.org/en-US/docs/Web/SVG/Element
[15] https://developer.mozilla.org/en-US/docs/Web/SVG/Element/rect



freezing point rectangle

Let's make it a frosty blue to connote "freezing" and decrease its visual importance. You can't style SVG elements with `background` or `border` — instead, we can use `fill` and `stroke` respectively. We'll discuss the differences later in more depth. As we can see, the default fill for SVG elements is black and the default stroke color is `none` with a width of 1px.

code/01-making-your-first-chart/completed/chart.js

```
50  const freezingTemperatures = bounds.append("rect")
51      .attr("x", 0)
52      .attr("width", dimensions.boundedWidth)
53      .attr("y", freezingTemperaturePlacement)
54      .attr("height", dimensions.boundedHeight
55        - freezingTemperaturePlacement)
56      .attr("fill", "#e0f3f3")
```

Let's look at the rectangle in the **Elements** panel to see how the `.attr()` methods manipulated it.

```
<rect
  x="0"
  width="1530"
  y="325.7509689922481"
  height="24.24903100775191"
  fill="rgb(224, 243, 243)"
></rect>
```

Looking good!

> Some SVG styles can be set with either a CSS style or an attribute value, such as **fill**, **stroke**, and **stroke-width**. It's up to you whether you want to set them with `.style()` or `.attr()`. Once we're familiar with styling our charts, we'll apply classes using `.attr("class", "class-name")` and add styles to a separate CSS file.
>
> In this code, we're using `.attr()` to set the fill because an attribute has a lower CSS precedence than linked stylesheets, which will let us overwrite the value. If we used `.style()`, we'd be setting an inline style which would require an `!important` CSS declaration to override.

Let's move on and create a scale for the x axis. This will look like our y axis but, since we're working with date objects, we'll use a time scale which knows how to handle date objects.

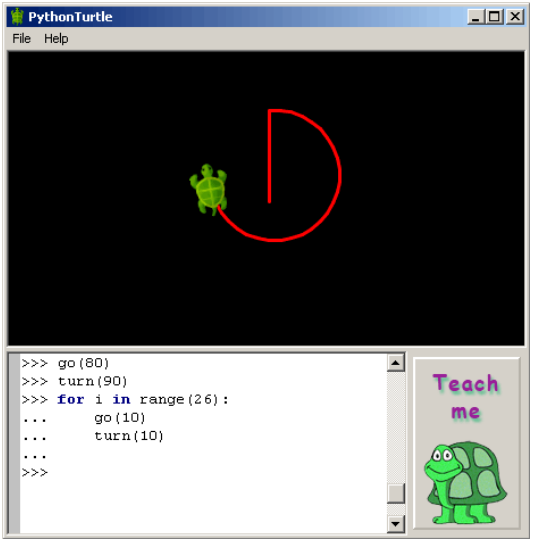
code/01-making-your-first-chart/completed/chart.js

```
58  const xScale = d3.scaleTime()
59    .domain(d3.extent(dataset, xAccessor))
60    .range([0, dimensions.boundedWidth])
```

Now that we have our scales defined, we can start drawing our chart!

Drawing the line

The timeline itself will be a single **path** SVG element. **path** elements take a **d** attribute (short for data) that tells them what shape to make. If you've ever played a learn-to-program game for kids, creating a **d** string is similar.



Coding turtle via http://pythonturtle.org/

The **d** attribute will take a few commands that can be capitalized (if giving an absolute value) or lowercased (if giving a relative value):

- **M** will move to a point (followed by x and y values)
- **L** will draw a line to a point (followed by x and y values)
- **Z** will draw a line back to the first point
- …

For example, let's draw this path:

```
bounds.append("path").attr("d", "M 0 0 L 100 0 L 100 100 L 0 50 Z")
```

We can see a new shape at the top of our chart.

d shape example

More **d** commands exist, but thankfully we don't need to learn them. d3's module **d3-shape**[16] has a `d3.line()` method that will create a generator that converts data points into a **d** string.

code/01-making-your-first-chart/completed/chart.js

```
64  const lineGenerator = d3.line()
```

[16] https://github.com/d3/d3-shape

Our generator needs two pieces of information:

1. how to find an x axis value, and
2. how to find a y axis value.

We set these values with the `x` and `y` method, respectively, which each take one parameter: a function to convert a data point into an x or y value.

We want to use our accessor functions, but remember: our accessor functions return the **unscaled value**.

We'll transform our data point with both the accessor function and the scale to get the scaled value in pixel space.

code/01-making-your-first-chart/completed/chart.js

```
64  const lineGenerator = d3.line()
65    .x(d => xScale(xAccessor(d)))
66    .y(d => yScale(yAccessor(d)))
```

Now we're ready to add the `path` element to our bounds.

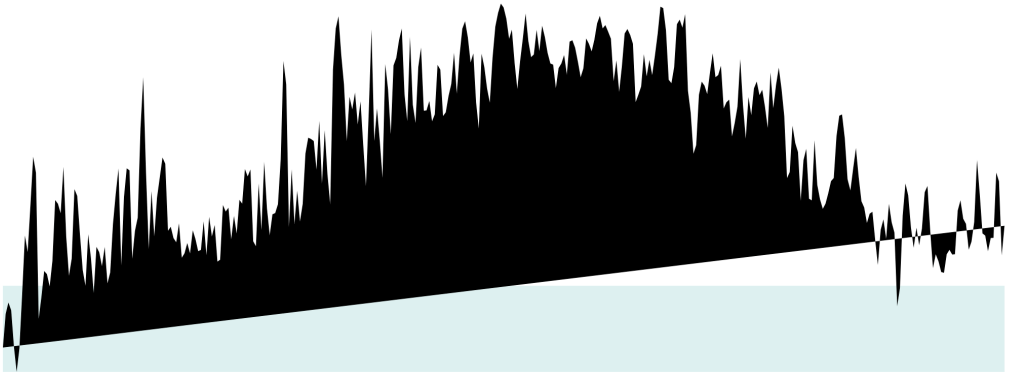
```
const line = bounds.append("path")
```

Let's feed our dataset to our line generator to create the **d** attribute and tell the line what shape to be.

```
const line = bounds.append("path")
    .attr("d", lineGenerator(dataset))
```

Success! We have a chart with a line showing our max temperature for the whole year.

Something looks wrong, though:

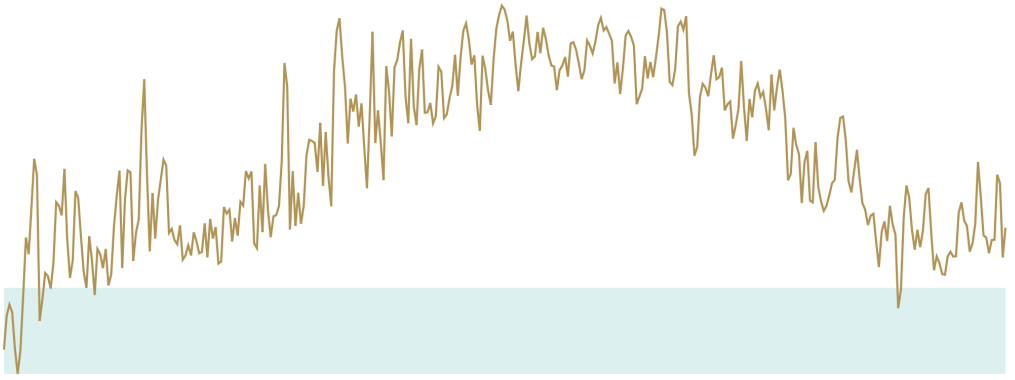


Our line!

Remember that SVG elements default to a black fill and no stroke, which is why we see this dark filled-in shape. This isn't what we want! Let's add some styles to get an orange line with no fill.

code/01-making-your-first-chart/completed/chart.js

```
68  const line = bounds.append("path")
69      .attr("d", lineGenerator(dataset))
70      .attr("fill", "none")
71      .attr("stroke", "#af9358")
72      .attr("stroke-width", 2)
```



Our line!

We're almost there, but something is missing. Let's finish up by drawing our axes.

Drawing the axes

Let's start with the y axis. d3's **d3-axis**[17] module has axis generator methods which will **draw an axis** *for the given scale.*

Unlike the methods we've used before, d3 axis generators will append multiple elements to the page.

[17] https://github.com/d3/d3-axis

There is one method for each orientation, which will specify the placement of labels and tick marks:

- `axisTop`
- `axisRight`
- `axisBottom`
- `axisLeft`

Following common convention, we want the labels of our y axis to be to the left of the axis line, so we'll use `d3.axisLeft()` and pass it our y scale.

code/01-making-your-first-chart/completed/chart.js

```
76  const yAxisGenerator = d3.axisLeft()
77    .scale(yScale)
```

When we call our axis generator, it will create a lot of elements — let's create a `g` element to hold all of those elements and keep our DOM organized. Then we'll pass that new element to our `yAxisGenerator` function to tell it where to draw our axis.

```
const yAxis = bounds.append("g")

yAxisGenerator(yAxis)
```

This method works but it will break up our chained methods. To fix this, d3 selections have a `.call()` method that will execute the provided function with the selection as the first parameter.

We can use `.call()` to:

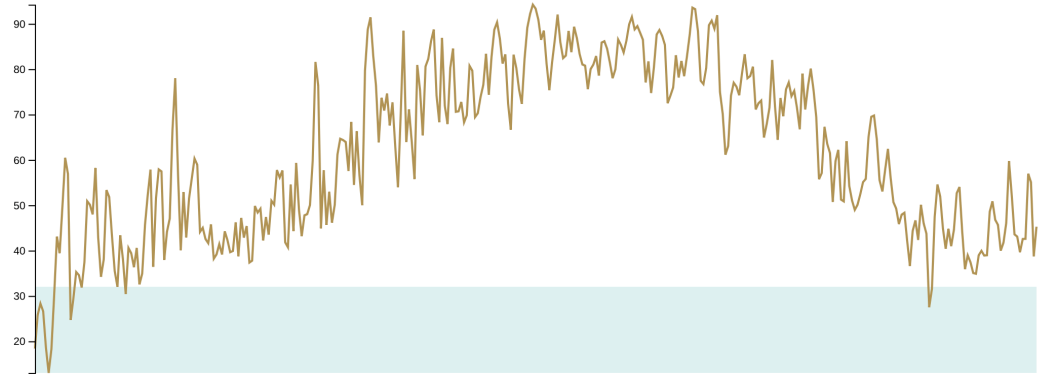
1. prevent saving our selection as a variable, and
2. preserve the selection for additional chaining.

Note that this code does exactly the same thing as the snippet above - we are passing the **function** `yAxisGenerator` to `.call()`, which then runs the function for us.

code/01-making-your-first-chart/completed/chart.js

```
79  const yAxis = bounds.append("g")
80    .call(yAxisGenerator)
```

And voila, we have our first axis!



Y axis

The small notches perpendicular to the axis are called *tick marks*. d3 has made behind-the-scenes decisions about how many tick marks to make and how far apart to draw them. We'll learn more about how to customize this later.

Axis

Let's create the x axis in the same way, this time using `d3.axisBottom()`.

code/01-making-your-first-chart/completed/chart.js

```
82   const xAxisGenerator = d3.axisBottom()
83     .scale(xScale)
```

Alright! Now let's create another <g> element and draw our axis.

```
const xAxis = bounds.append("g")
    .call(xAxisGenerator)
```

> We *could* `.call()` our x axis directly on our **bounds**:
>
> `const xAxis = bounds.call(xAxisGenerator)`
>
> This would create our axis directly under our **bounds** (in the DOM).
>
> However, it's a good idea to create a <g> element to contain our axis elements for three main reasons:
>
> 1. to keep our DOM organized, for debugging or exporting
> 2. if we want to remove or update our axis, we'll want an easy way to target all of the elements
> 3. modifying our whole axis at once, for example when we want to move it around.

The axis looks right, but it's in the wrong place:

GET THE FULL BOOK

This is the end of the preview!

Head over to:

https://fullstack.io/fullstack-d3

to get the full version!

Featuring:

- 12+ lessons
- 300+ pages
- Complete code

GET IT NOW