

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 damages in connection with or arising out of the use of the information or programs container herein.

Published by Fullstack.io.

Contents

Book Revision	1
Bug Reports	1
Be notified of updates via Twitter	1
We'd love to hear from you!	1
Making Your First Chart	1
Getting started	1
Loading the weather dataset	3
Getting a server running	5
Looking at our data	6
Setting up our line chart	8
Drawing our chart	10
Creating our workspace	12
Adding an SVG element	13
Creating our bounding box	16
Creating our scales	17
Drawing the line	23
Drawing the axes	27
Making a Scatterplot	33
Intro	33
Deciding the chart type	33
Steps in drawing any chart	34
Access data	37
Create chart dimensions	37
Draw canvas	41
Create scales	42
The concept behind scales	43

CONTENTS

Finding the extents	43
Draw data	46
Data joins	49
Data join exercise	55
.join()	58
Draw peripherals	59
Initialize interactions	63
Looking at our chart	64
Extra credit: adding a color scale	64
Making a Bar Chart	68
Deciding the chart type	68
Histogram	69
Chart checklist	72
Access data	73
Create dimensions	73
Draw canvas	75
Create scales	76
Creating Bins	77
Creating the y scale	80
Draw data	81
Adding Labels	85
Extra credit	89
Draw peripherals	93
Set up interactions	94
Looking at our chart	94
Extra credit	95
Accessibility	100
Animations and Transitions	105
SVG <animate>	105
CSS transitions	107
Using CSS transition with a chart	112
d3.transition	117
Lines	131
Interactions	139

CONTENTS

d3 events	139
Don't use fat arrow functions	143
Destroying d3 event listeners	144
Bar chart	146
Scatter plot	160
Voronoi	165
Changing the hovered dot's color	172
Line chart	175
d3.scan()	178
Making a map	186
Digging in	187
What is GeoJSON?	187
Access data	189
Our dataset	193
Create chart dimensions	196
What is a projection?	196
Which projection should I use?	197
Finishing creating our chart dimensions	200
Draw canvas	203
Create scales	204
Draw data	207
Draw peripherals	213
Drawing a legend	213
Marking my location	223
Set up interactions	227
Wrapping up	232
Data Visualization Basics	233
Types of data	235
Qualitative Data	237
Quantitative Data	239
Ways to visualize a metric	240
Size	241
Position	242
Color	243

CONTENTS

Putting it together	244
Chart design	244
Simplify, simplify, simplify	244
Annotate in-place	245
Add enhancements, but not too many	246
Example redesign	246
Colors	252
Color scales	253
1. Representing a category	253
Custom color scales	256
Creating our own colors	260
keywords	261
rgb	261
hsl	263
hcl	264
d3-color	267
Color tips	267
Wrapping up	271
Common charts	273
Chart types	274
Timeline	275
Heatmap	279
Radar	281
Scatter	283
Pie charts & Donut charts	290
Histogram	292
Box plot	296
Conclusion	299
Dashboard Design	301
What is a dashboard?	301
Showing a simple metric	302
Dealing with dynamic data	309
Data states	309
Dancing numbers	314

CONTENTS

Designing tables	318
Designing a dashboard layout	326
Have a main focus	326
Keep the number of visible metrics small	327
Let the user dig in	329
Deciding questions	330
How familiar are users with the data?	330
How much time do users have to spend?	331
D3.js	333
What did we cover?	334
What did we not cover?	336
Going forward	339
How was your experience?	341
Appendix	342
A. Generating our own weather data	342
Chrome's Color Contrast Tool	343
B. Chart-drawing checklist	346
C. SVG elements cheat sheet	348
Changelog	349
Revision 8	349
06-26-2019	349
Revision 7	350
06-14-2019	350
Revision 6	351
06-06-2019	351
Revision 5	351
06-03-2019	351
Revision 4	352
05-24-2019	352
Revision 3	352
05-17-2019	352

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

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](https://twitter.com/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.io².

¹<https://twitter.com/fullstackio>

²<mailto:us@fullstack.io>

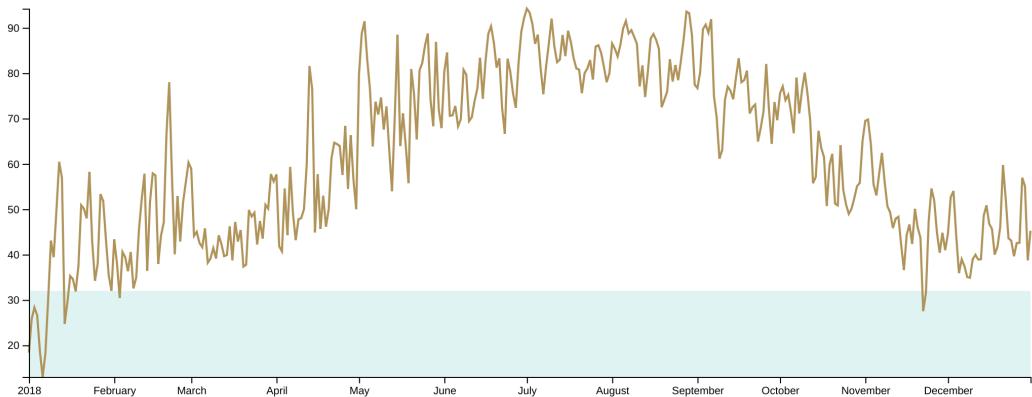
Making Your First Chart

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](#).

^ahttps://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.

```
✖ ▶ Fetch API cannot load file:///C:/Users/watte/Development/repos/fullstack-d3-book/manuscript/code/src/1-making-your-first-charts/my_weather_data.json. URL scheme must be "http" or "https" for CORS request.  
✖ ▶ Uncaught (in promise) TypeError: Failed to fetch d3.js:5908  
    at Object.json (d3.js:5908)  
    at drawLineWeather (drawLineWeather.js:10)  
    at drawLineWeather.js:127
```

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 here⁴](#)). 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.

⁴<https://docs.npmjs.com/downloading-and-installing-node-js-and-npm>

⁵<https://github.com/tapio/live-server>

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.

Now we should have a server on port 8080. Load [localhost:8080⁶](http://localhost:8080) 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/draft7`](http://localhost:8080/01-making-your-first-chart/draft) 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.

⁶<http://localhost:8080>

⁷<http://localhost:8080/01-making-your-first-chart/draft>

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

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

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])
```

(index)	Value	chart.js:6
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	1514888000	
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](https://darksky.net/dev/docs#data-point)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-format⁹](#) 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.

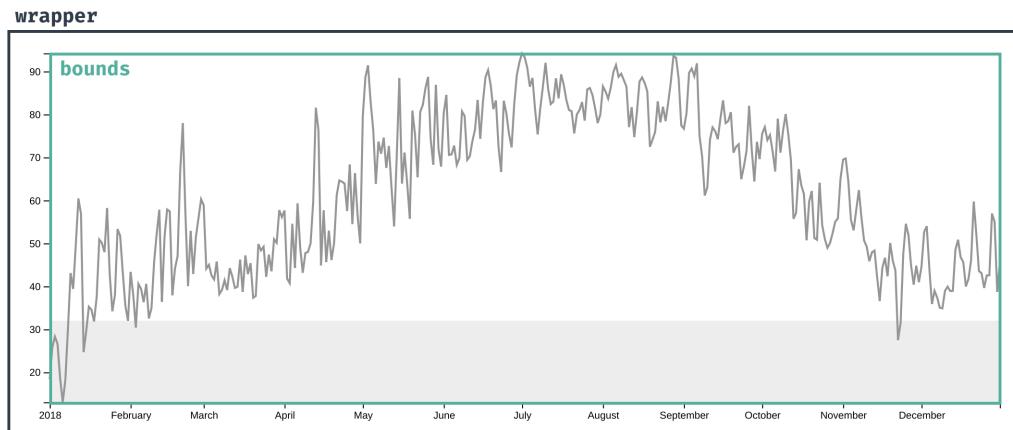


Chart dimensions

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-selection¹⁰](#), 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.

```
▼ Selection {_groups: Array(1), _parents: Array(1)} ⓘ  
  ▼ _groups: Array(1)  
    ► 0: [div#wrapper]  
      length: 1  
    ► __proto__: Array(0)  
  ▼ _parents: [html]  
    ► __proto__: Object
```

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.

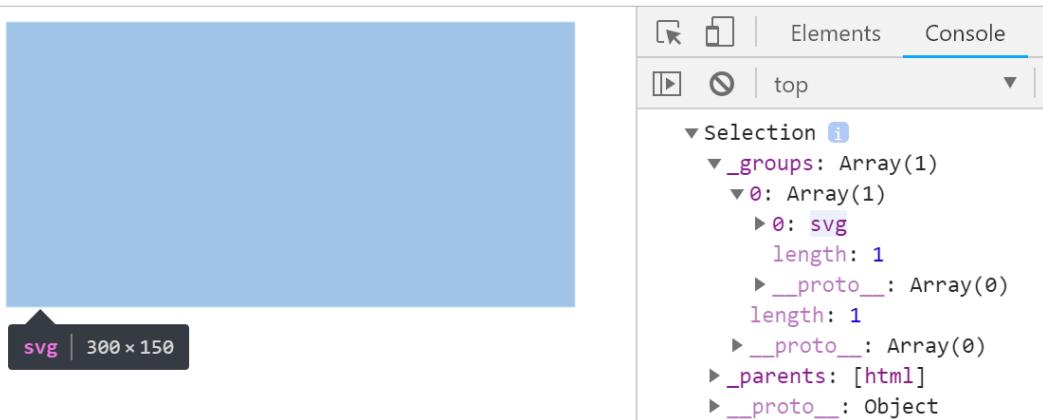
Adding an SVG element

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](#)⁴.

⁴<https://babeljs.io/docs/en/learn/#template-strings>

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



```
<!doctype html>
<html lang="en">
  <head>...</head>
  <body>
    <div id="wrapper">
      <svg width="324.9000000000003" height="400">
        ... <g style="transform: translate(60px, 15px);"></g> == $0
      </svg>
    </div>
    <script src="/../../d3.v5.js"></script>
    <script src="_/chart.js"></script>
    <!-- Code injected by live-server -->
    ><script type="text/javascript">...</script>
  </body>
</html>
```

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`.

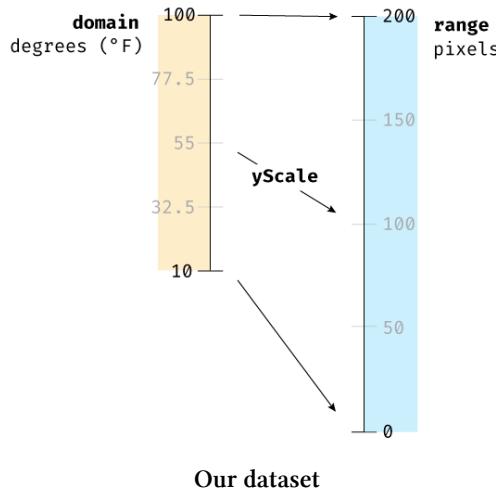
Additionally, if our lowest temperature is below 0 we would have to plot that value below the chart! Our y axis wouldn't be able to handle all of our temperature values.

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

d3's [d3-scale¹¹](#) 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.

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.



[d3-scale¹²](#) 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()`.

¹¹<https://github.com/d3/d3-scale>

¹²<https://github.com/d3/d3-scale>

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:

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.

¹³<https://github.com/d3/d3-array>

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 docs¹⁴](#) are a wonderful resource: [here is the page for `<rect>`¹⁵](#).

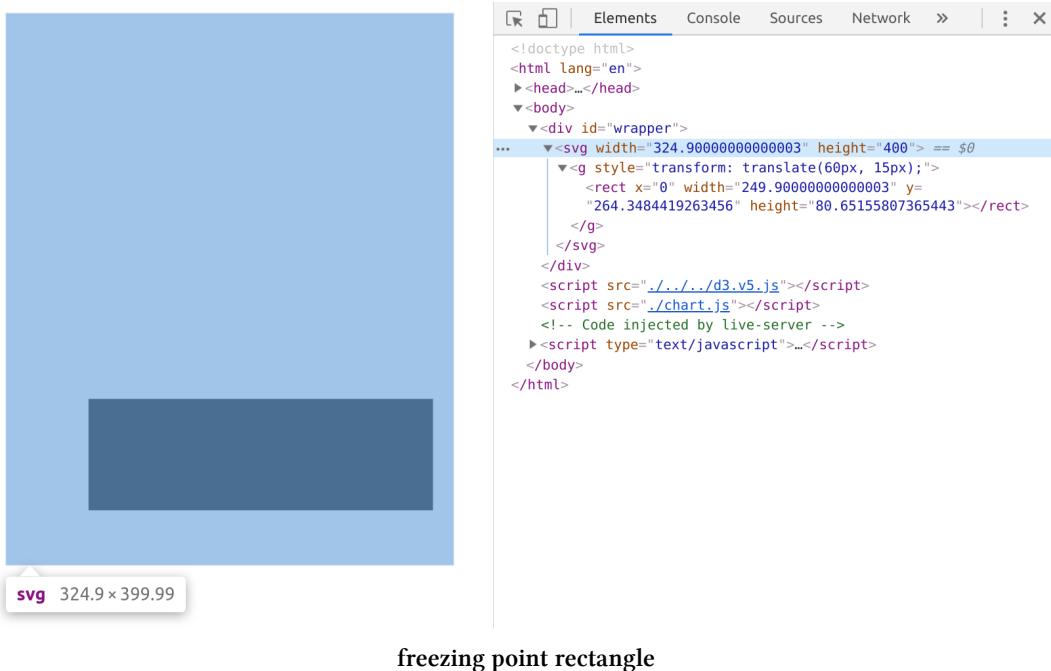
¹⁴<https://developer.mozilla.org/en-US/docs/Web/SVG/Element>

¹⁵<https://developer.mozilla.org/en-US/docs/Web/SVG/Element/rect>

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.



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.

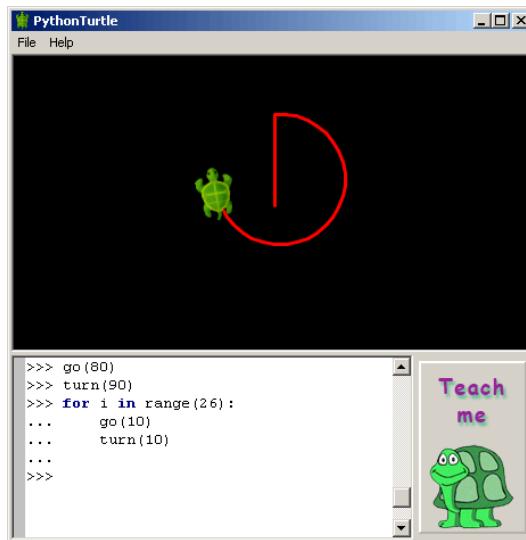
code/01-making-your-first-chart/completed/chart.js

```
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()

Our generator needs two pieces of information:

1. how to find an x axis value, and

¹⁶<https://github.com/d3/d3-shape>

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.

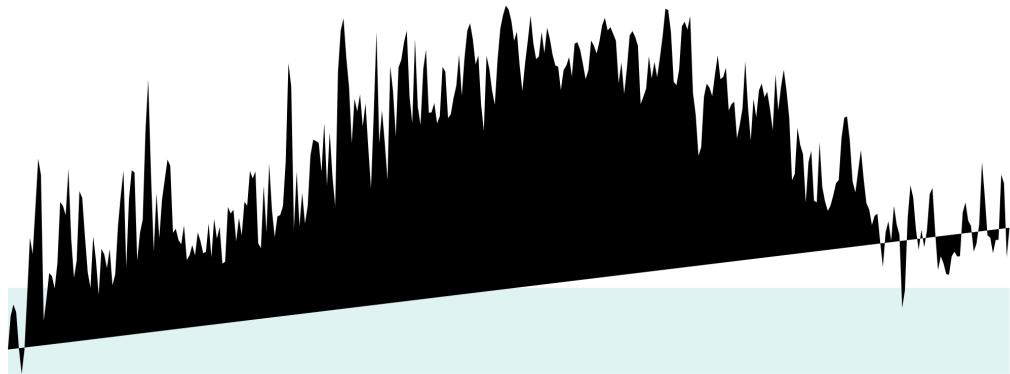
```
const line = bounds.append("path")
```

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:

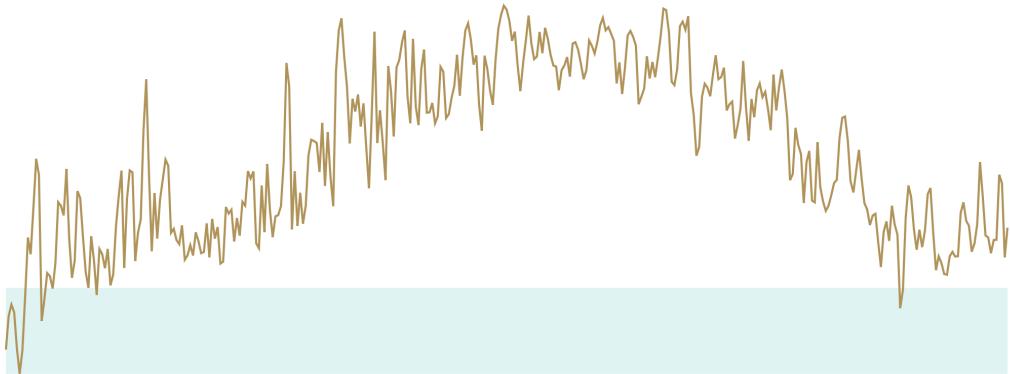


Our line!

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-axis¹⁷](#) 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:

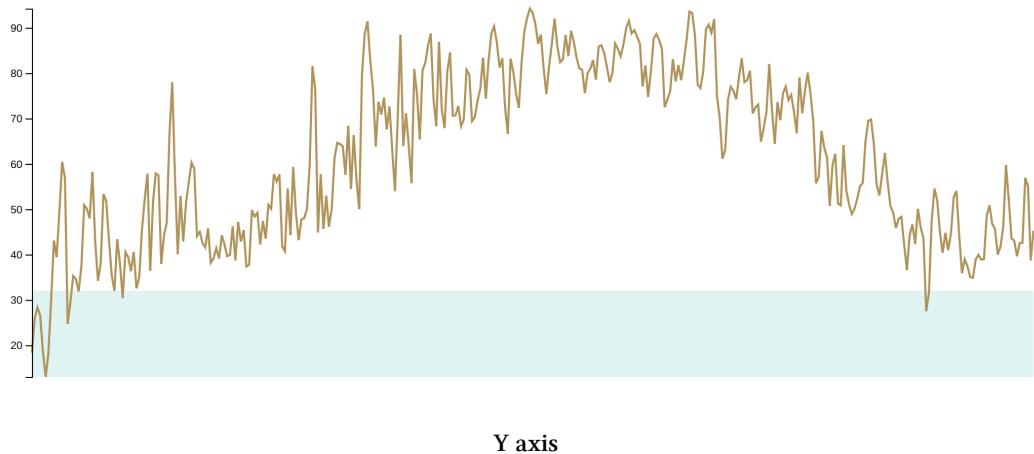
1. prevent saving our selection as a variable, and
2. preserve the selection for additional chaining.

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")
  .call(yAxisGenerator)
80
```

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.

```

    ▼<g fill="none" font-size="10" font-family="sans-serif" text-anchor="end">
      <path class="domain" stroke="currentColor" d="M-6,350.5H0.5V0.5H-6">
    </path>
    ▼<g class="tick" opacity="1" transform="translate(0,337.5558785529716)">
      <line stroke="currentColor" x2="-6"></line>
      <text fill="currentColor" x="-9" dy="0.32em">30</text>
    </g>
    ▶<g class="tick" opacity="1" transform="translate(0,309.2936046511628)">...
    </g>
    ▶<g class="tick" opacity="1" transform="translate(0,281.03133074935397)">...
    </g>
    ▶<g class="tick" opacity="1" transform="translate(0,252.76905684754522)">...
    </g>
    ▶<g class="tick" opacity="1" transform="translate(0,224.5067829457364)">...
    </g>
    ▶<g class="tick" opacity="1" transform="translate(0,196.24450904392762)">...
    </g>
  </g>
  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)
```

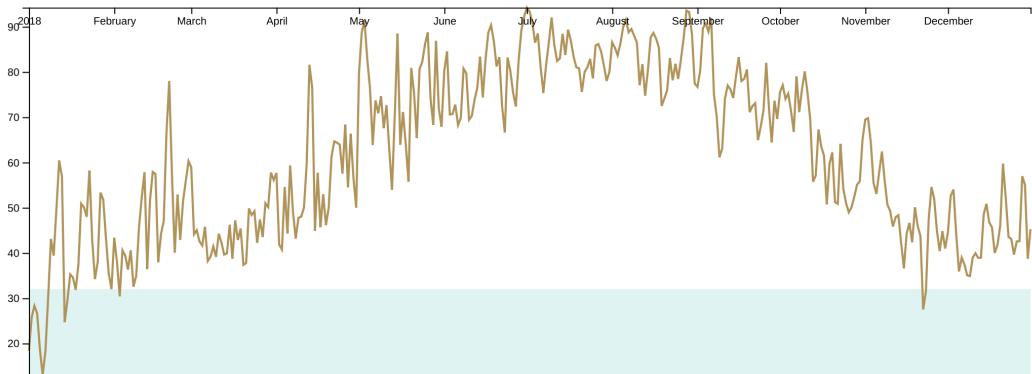
This would create our axis directly under our `bounds` (in the DOM).

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:



x axis on top

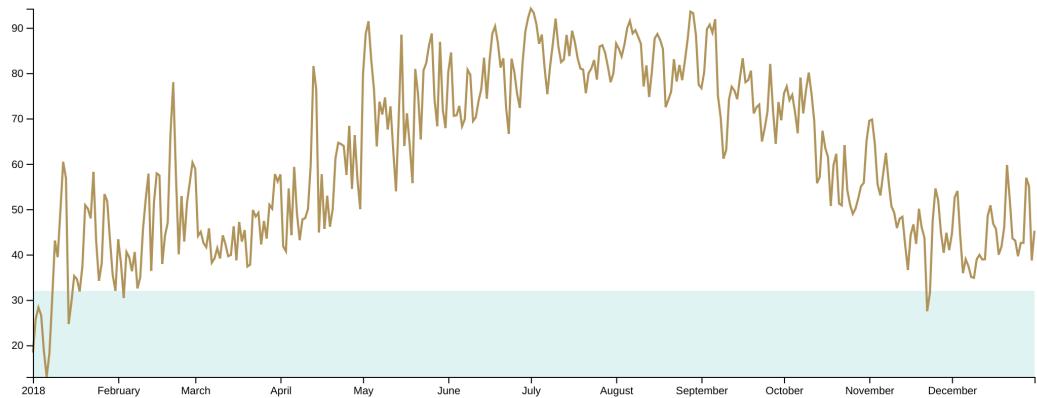
Why didn't `.axisBottom()` draw the axis in the right place? d3's axis generator functions know where to place the tick marks and tick labels relative to the axis line, but they have no idea where to place the axis itself.

To move the x axis to the bottom, we can shift the x axis group, similar to how we shifted our chart bounds using a CSS transform.

[code/01-making-your-first-chart/completed/chart.js](#)

```
85 const xAxis = bounds.append("g")
86   .call(xAxisGenerator)
87   .style("transform", `translateY(${{
88     dimensions.boundedHeight
89   }px})`)
```

And just like that we're done making our first chart!



Finished line graph

Next, let's dive into making a slightly more complex chart and talk more about how d3 works for a deeper understanding of the concepts we just learned.

Making a Scatterplot

Intro

Now that we've created our first chart, let's create another chart that's a little more complex. At the end of this chapter, we'll have a deeper understanding of each step required to make a chart in d3.

There are endless questions we could ask our weather dataset — many of them ask about the relationship between different metrics. Let's investigate these two metrics:

- **dew point** is the highest temperature (°F) at which dew droplets form
- **humidity** is the amount of water vapor in the air

I would expect them to be correlated — high humidity should cause a higher dew point temperature, right? Let's dive in and find out!

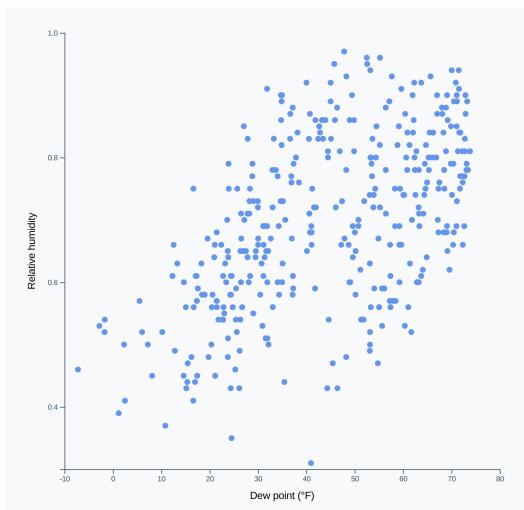
Deciding the chart type

When looking at the relationship between two metrics, a **scatterplot is a good choice**.

A scatterplot includes two axes:

- an **x axis** that displays one metric and
- a **y axis** that displays the other.

We'll plot **each data point** (in this case, a single day) as a dot. If we wanted to involve a third metric, we could even add another dimension by changing the color or the size of each dot.



Our Finished Scatterplot

The great thing about scatterplots is that when we're finished plotting the chart, we'll get a clear view of the relationship between the two metrics. We'll talk more about the potential patterns in [Chapter 8](#).

Steps in drawing any chart

In d3, there are general steps that we need to take every time we make a chart — we briefly went through each of them in [Chapter 1](#) to create our line chart, but now let's create a checklist to give us a roadmap for future charts.

1. Access data

Look at the data structure and declare how to access the values we'll need

2. Create chart dimensions

Declare the physical (i.e. pixels) chart parameters

3. Draw canvas

Render the chart area and bounds element

4. Create scales

Create scales for every data-to-physical attribute in our chart

5. Draw data

Render your data elements

6. Draw peripherals

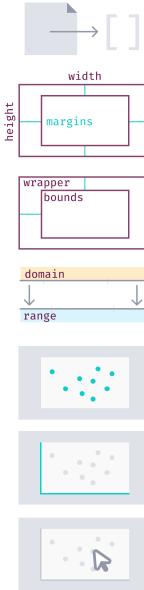
Render your axes, labels, and legends

7. Set up interactions

Initialize event listeners and create interaction behavior - we'll get to this step in Chapter 5

We have a [Chart drawing checklist PDF](#) in the [Advanced package](#) for easy reference. Feel free to print it out or save it somewhere to give you an outline in the future!

Chart drawing checklist



- Access data**
Look at the data structure and declare how to access the values we'll need
- Create chart dimensions**
Declare the physical (i.e. pixels) chart parameters
- Draw canvas**
Render the `wrapper` and `bounds` element
- Create scales**
Create scales for every data-to-physical attribute in our chart
- Draw data**
Render your data elements
- Draw peripherals**
Render your axes, labels, legends, annotations, etc
- Set up interactions**
Initialize event listeners and create interaction behavior

Chart drawing checklist

Let's dig in! To start, open up the

`/code/02-making-a-scatterplot/draft/draw-scatter.js`

file and point your browser at

<http://localhost:8080/02-making-a-scatterplot/draft/>¹⁸

Remember to start your server to prevent CORS errors (live-server).

As usual, the completed code is in the `/code/02-making-a-scatterplot/completed/` folder, for reference.

¹⁸<http://localhost:8080/02-making-a-scatterplot/draft/>

Access data

As we saw in [Chapter 1](#), this step will be quick! We can utilize `d3.json()` to grab the `my_weather_data.json` file.

code/02-making-a-scatterplot/completed/draw-scatter.js

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

The next part of the **Access data** step is to create our accessor functions. Let's log the first data point to the console to look at the available keys.

```
const dataset = await d3.json("./../../my_weather_data.json")
console.table(dataset[0])
```

We can see the metrics we're interested in as `humidity` and `dewPoint`. Let's use those to define our accessor functions.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
6 const xAccessor = d => d.dewPoint
7 const yAccessor = d => d.humidity
```

Perfect! Now that we can access our data, we can move to the next step.

Create chart dimensions

Next up, we need to define the dimensions of our chart. Typically, scatterplots are square, with the x axis as wide as the y axis is tall. This makes it easier to look at the overall shape of the data points once they're plotted by not stretching or squashing one of the scales.

To make a square chart, we want the height to be the same as the width. We could use the same width we used in [Chapter 1](#) (`window.innerWidth * 0.9`), but then the chart might extend down the page, out of view on horizontal screens.

Ideally, the chart will be as large as possible while still fitting on our screen.

To fix this problem, we want to use either the height or the width of the window, whichever one is smaller. And because we want to leave a *little* bit of whitespace around the chart, we'll multiply the value by `0.9` (so 90% of the total width or height). `d3.array` can help us out here with the `d3.min` method. `d3.min` takes two arguments:

1. an array of data points
2. an accessor function to grab the value from each data point

Though in this case we won't need to specify the second parameter because it defaults to an identity function and returns the value.

`code/02-making-a-scatterplot/completed/draw-scatter.js`

```
12 const width = d3.min([
13   window.innerWidth * 0.9,
14   window.innerHeight * 0.9,
15 ])
```

There is a native browser method (`Math.min`) that will also find the lowest number — why wouldn't we use that? `Math.min` is great, but there are a few benefits to `d3.min`:

- `Math.min` will count any `nulls` or `undefineds` in the array as `0`, whereas `d3.min` will ignore them
- `Math.min` will return `NaN` if there is a value in the array that can't be converted into a number, whereas `d3.min` will ignore it
- `d3.min` will prevent the need to create another array of values if we need to use an accessor function
- `Math.min` will return `Infinity` if the dataset is empty, whereas `d3.min` will return `undefined`
- `Math.min` uses numeric order, whereas `d3.min` uses natural order, which allows it to handle strings

You can see how `d3.min` would be preferable when creating charts, especially when using dynamic data.

Now let's use our `width` variable to define the chart dimensions:

```
let dimensions = {  
  width: width,  
  height: width,  
}  
}
```

We were introduced to the concept of **wrapper** and **bounds** in Chapter 1. As a reminder:

- the **wrapper** is your entire SVG element, containing your axes, data elements, and legends
- the **bounds** live inside of the **wrapper**, containing just the data elements

Having margins around the **bounds** allows us to allocate space for our static chart elements (axes and legends) while allowing the charting area to be dynamically sized based on the available space.

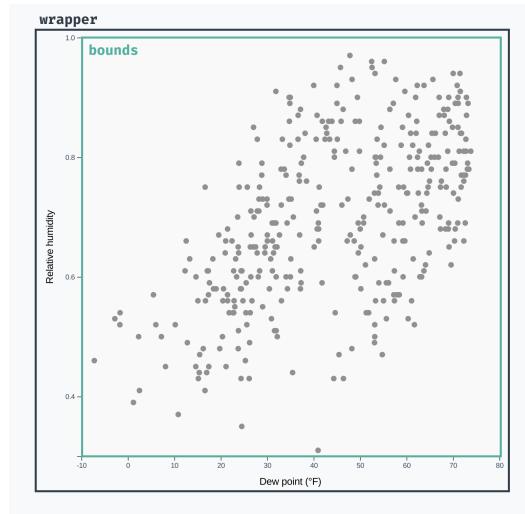


Chart terminology

We want a small `top` and `right` margin to give the chart some space. Dots near the top or right of the chart or the y axis's topmost tick label might overflow our **bounds** (because the position of the dot is technically the *center* of the dot, but the dot has a radius).

We'll want a larger `bottom` and `left` margin to create room for our axes.

[code/02-making-a-scatterplot/completed/draw-scatter.js](#)

```
16 let dimensions = {  
17   width: width,  
18   height: width,  
19   margin: {  
20     top: 10,  
21     right: 10,  
22     bottom: 50,  
23     left: 50,  
24   },  
25 }
```

Lastly, we want to define the width and height of our **bounds**, calculated from the space remaining after we add the margins.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
26 dimensions.boundedWidth = dimensions.width  
27   - dimensions.margin.left  
28   - dimensions.margin.right  
29 dimensions.boundedHeight = dimensions.height  
30   - dimensions.margin.top  
31   - dimensions.margin.bottom
```

You might be asking: why do we have to be explicit about the chart dimensions? Generally when developing for the web we can let elements size themselves to fit their contents or to fill the available space. That's not an option here for a few reasons:

- SVG elements scale in an unfamiliar and inconsistent way
- we need to know the width and height of the chart in order to calculate the scale outputs
- we generally want more control over the size of our chart elements — in this example, we want the width and height to be the same size, we want our dots to be large enough to see, etc

Draw canvas

Let's make some SVG elements! This step will look exactly like our line chart code. First, we find an existing DOM element (`#wrapper`), and append an `<svg>` element.

Then we use `attr` to set the size of the `<svg>` to our `dimensions.width` and `dimensions.height`. Note that these sizes are the size of the “outside” of our plot. Everything we draw next will be within this `<svg>`.

code/02-making-a-scatterplot/completed/draw-scatter.js

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

Next, we create our bounds and shift them to accommodate our top & left margins.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
40 const bounds = wrapper.append("g")
41   .style("transform", `translate(${{
42     dimensions.margin.left
43   }px, ${{
44     dimensions.margin.top
45   }}px)`)
```

Above, we create a `<g>` (think “group”) element and we use the `transform` CSS property to move it to the right and down (note that the `left` margin pushes our bounds to the right, and a top margin pushes our bounds down).

This `bounds` is the “inner” part of our chart that we will use for our data elements.

Create scales

Before we draw our data, we have to figure out how to convert numbers from the data domain to the pixel domain.

Let’s start with the x axis. We want to decide the horizontal position of each day’s dot based on its **dew point**.

To find this position we use a **d3 scale object**, which helps us map our data to pixels. Let’s create a scale that will take a dew point (temperature) and tell us *how far to the right* a dot needs to be.

This will be a *linear* scale because the input (dew point) and the output (pixels) will be numbers that increase linearly.

```
const xScale = d3.scaleLinear()
```

The concept behind scales

Remember, we need to tell our scale:

- what *inputs* it will need to handle (**domain**), and
- what *outputs* we want back (**range**).

For a simple example, let's pretend that the temperatures in our dataset range from 0 to 100 degrees.

In this case, converting from *temperature* to *pixels* is easy: a temperature of 50 degrees maps to 50 pixels because both **range** and **domain** are $[0, 100]$.

But the relationship between our data and the pixel output is rarely so simple. What if our chart was 200 pixels wide? What if we have to handle negative temperatures?

Mapping between metric values and pixels is one of the areas in which d3 scales shine.

Finding the extents

In order to create a scale, we need to pick the smallest and largest values we will handle. These numbers can be anything you want, but the standard practice is to examine your data and extract the minimum and maximum values. This way your chart will “automatically” scale according to the values in your dataset.

D3 has a helper function we can use here: `d3.extent()` that takes two parameters:

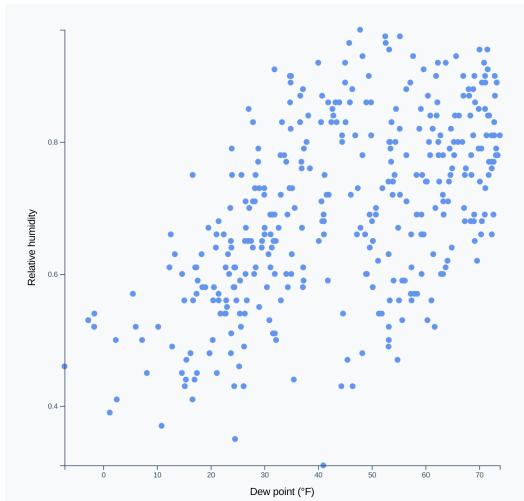
1. an array
2. an accessor function that extracts the metric value from a data point. If not specified, this defaults to an identity function `d => d`.

We'll pass `d3.extent()` our dataset and our `xAccessor()` function and get the min and max temperatures we need to handle (in $[\min, \max]$ format).

code/02-making-a-scatterplot/completed/draw-scatter.js

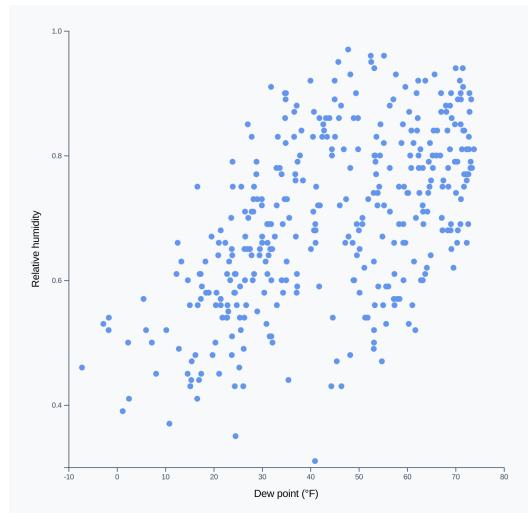
```
49 const xScale = d3.scaleLinear()  
50   .domain(d3.extent(dataset, xAccessor))  
51   .range([0, dimensions.boundedWidth])
```

This scale will create a perfectly useable chart, but we can make it slightly friendlier. With this x scale, our x axis will have a domain of [11.8, 77.26] — the exact min and max values from the dataset. The resulting chart will have dots that extend all the way to the left and right edges.



Finished scatterplot without nice scales

While this works, it would be easier to read the axes if the first and last tick marks were round values. Note that d3 won't even label the top and bottom tick marks of an axis with a strange domain — it might be hard to reason about a chart that scales up to 77.26 degrees. That number of decimal points gives too much unnecessary information to the reader, making them do the next step of rounding the number to a more tangible one.



Finished scatterplot with nice scales

d3 scales have a `.nice()` method that will round our scale's domain, giving our x axis friendlier bounds.

We can look at how `.nice()` modifies our x scale's domain by looking at the values before and after using `.nice()`. *Note that calling `.domain()` without parameters on an existing scale will output the scale's existing domain instead of updating it.*

```
console.log(xScale.domain())
xScale.nice()
console.log(xScale.domain())
```

With the New York City dataset, the domain changes from [11.8, 77.26] to [10, 80] — much friendlier! Let's chain that method when we create our scale.

[code/02-making-a-scatterplot/completed/draw-scatter.js](#)

```
49 const xScale = d3.scaleLinear()
50   .domain(d3.extent(dataset, xAccessor))
51   .range([0, dimensions.boundedWidth])
52   .nice()
```

Creating our y scale will be very similar to creating our x scale. The only differences are:

1. we'll be using our `yAccessor()` to grab the humidity values, and
2. we want to invert the range to make sure the axis runs bottom-to-top.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
54 const yScale = d3.scaleLinear()  
55   .domain(d3.extent(dataset, yAccessor))  
56   .range([dimensions.boundedHeight, 0])  
57   .nice()
```

If we were curious about how `.nice()` modifies our y scale, we could log those values.

```
console.log(d3.extent(dataset, yAccessor))  
console.log(yScale.domain())
```

In this case, the domain changed from `[0.27, 0.97]` to `[0.2, 1]`, which will create a much friendlier chart.

Draw data

Here comes the fun part! Drawing our scatterplot dots will be different from how we drew our timeline. Remember that we had one line that covered all of the data points? For our scatter plot, we want **one element per data point**.

We'll want to use the `<circle>`¹⁹ SVG element, which thankfully doesn't need a `d` attribute string. Instead, we'll give it `cx` and `cy` attributes, which set its x and y coordinates, respectively. These position the center of the circle, and the `r` attribute sets the circle's radius (half of its width or height).

Let's draw a circle in the center of our chart to test it out.

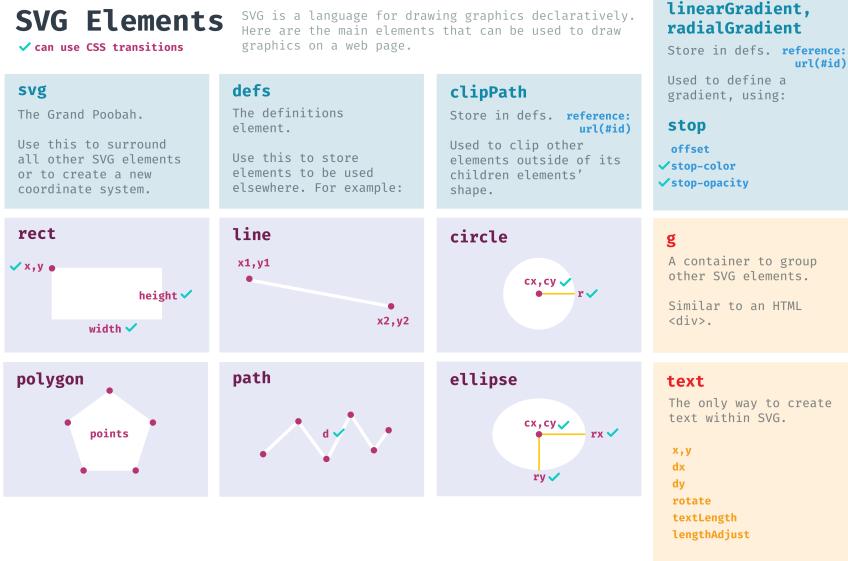
¹⁹<https://developer.mozilla.org/en-US/docs/Web/SVG/Element/circle>

```
bounds.append("circle")
    .attr("cx", dimensions.boundedWidth / 2)
    .attr("cy", dimensions.boundedHeight / 2)
    .attr("r", 5)
```



Test circle

Starting to get SVG elements mixed up? No worries! Our advanced package has an [SVG elements cheat sheet](#) PDF to help remember what elements exist and what attributes they want. Don't worry if you don't recognize any of the elements — we'll cover them all by the end of the book.



SVG elements cheat sheet

Great! Now let's add one of those for each day.

A straightforward way of drawing the dots would be to map over each element in the dataset and append a circle to our bounds.

```
dataset.forEach(d => {
  bounds
    .append("circle")
    .attr("cx", xScale(xAccessor(d)))
    .attr("cy", yScale(yAccessor(d)))
    .attr("r", 5)
})
```

Look at that! Now we're starting to get a better sense of our data.



Dots!

While this method of drawing the dots works for now, there are a few issues we should address.

- We're adding a level of nesting, which makes our code harder to follow.
- If we run this function twice, we'll end up drawing two sets of dots. When we start updating our charts, we will want to draw and update our data with the same code to prevent repeating ourselves.

To address these issues and keep our code clean, let's handle the dots without using a loop.

Data joins

Scratch that last block of code. D3 has functions that will help us address the above problems.

We'll start off by grabbing all `<circle>` elements in a **d3 selection object**. Instead of using `d3.selection's .select()` method, which returns one matching element, we'll use its `.selectAll()` method, which returns an array of matching elements.

```
const dots = bounds.selectAll("circle")
```

This will seem strange at first — we don't have any dots yet, why would we select something that doesn't exist? Don't worry! You'll soon become comfortable with this pattern.

We're creating a d3 selection that is aware of what elements already exist. If we had already drawn part of our dataset, this selection will be aware of what dots were already drawn, and which need to be added.

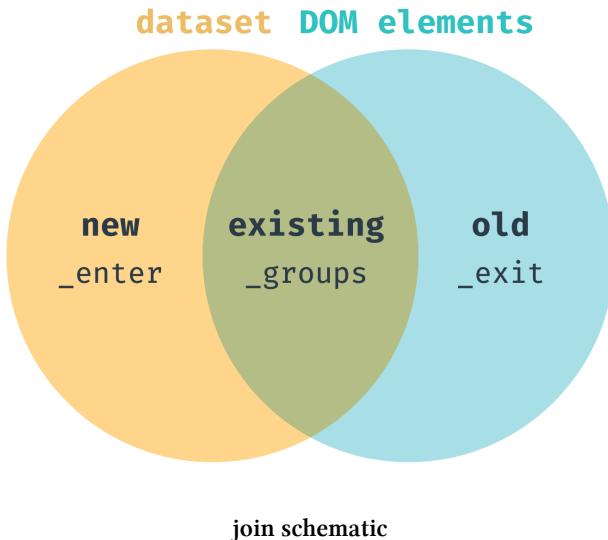
To tell the selection what our data look like, we'll pass our dataset to the selection's `.data()` method.

```
const dots = bounds.selectAll("circle")
  .data(dataset)
```

When we call `.data()` on our selection, we're joining our selected elements with our array of data points. The returned selection will have a list of existing elements, new elements that need to be added, and old elements that need to be removed.

We'll see these changes to our selection object in three ways:

- our selection object is updated to contain any overlap between existing DOM elements and data points
- an `_enter` key is added that lists any data points that don't already have an element rendered
- an `_exit` key is added that lists any data points that are already rendered but aren't in the provided dataset



Let's get an idea of what that updated selection object looks like by logging it to the console.

```
let dots = bounds.selectAll("circle")
console.log(dots)
dots = dots.data(dataset)
console.log(dots)
```

Remember, the currently selected DOM elements are located under the `_groups` key. Before we join our dataset to our selection, the selection just contains an empty array. That makes sense! There are no circles in `bounds` yet.

```
▼ Selection {_groups: Array(1), _parents: Array(1)} ⓘ  
  ▼ _groups: Array(1)  
    ► 0: NodeList []  
      length: 1  
    ► __proto__: Array(0)  
  ► _parents: [g]  
  ► __proto__: Object
```

empty selection

However, the next selection object looks different. We have two new keys: `_enter` and `_exit`, and our `_groups` array has an array with 365 elements — the number of data points in our dataset.

```
draw-scatter.js:65  
▼ Ct {_groups: Array(1), _parents: Array(1), _enter: Array(1), _exit  
  : Array(1)} ⓘ  
  ► _enter: [Array(365)]  
  ► _exit: [Array(0)]  
  ► _groups: [Array(365)]  
  ► _parents: [g]  
  ► __proto__: Object
```

.data selection

Let's take a closer look at the `_enter` key. If we expand the array and look at one of the values, we can see an object with a `__data__` property.

```
          draw-scatter.js:65
Ct {_groups: Array(1), _parents: Array(1), _enter: Array(1), _ex
  it: Array(1)} i
    ▼_enter: Array(1)
      ▼0: Array(365)
        ▼[0 .. 99]
          ▼0: et
            namespaceURI: "http://www.w3.org/2000/svg"
            ►ownerDocument: document
            ▼__data__:
              apparentTemperatureHigh: 17.29
              apparentTemperatureHighTime: 1514844000
              apparentTemperatureLow: 4.51
              apparentTemperatureLowTime: 1514887200
              apparentTemperatureMax: 17.29
              apparentTemperatureMaxTime: 1514844000
              apparentTemperatureMin: -2.19
              apparentTemperatureMinTime: 1514808000
              cloudCover: 0.02
              date: "2018-01-01"
              dewPoint: -1.67
              humidity: 0.54
              icon: "clear-day"
              moonPhase: 0.48
              precipIntensity: 0
              precipIntensityMax: 0
              precipProbability: 0
              pressure: 1028.26
              summary: "Clear throughout the day."
              sunriseTime: 1514809280
            .data selection EnterNode
```

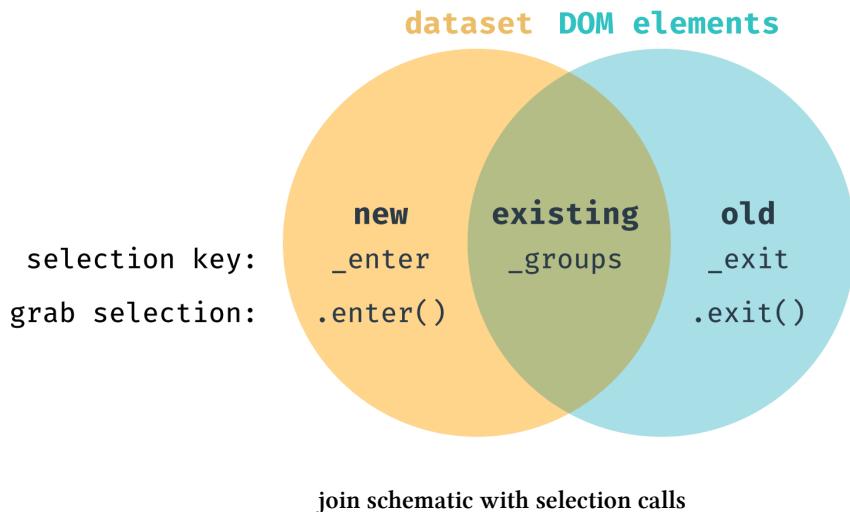
For the curious, the `namespaceURI` key tells the browser that the element is a SVG element and needs to be created in the “`http://www.w3.org/2000/svg`” namespace (SVG), instead of the default “`http://www.w3.org/1999/xhtml`” namespace (XHTML).

If we expand the `__data__` value, we will see one our data points.

Great! We can see that each value in `_enter` corresponds to a value in our dataset. This is what we would expect, since all of the data points need to be added to the DOM.

The `_exit` value is an empty array — if we were removing existing elements, we would see those listed out here.

In order to act on the **new** elements, we can create a d3 selection object containing just those elements with the `enter` method. There is a matching method (`exit`) for **old** elements that we'll need when we go over transitions in Chapter 4.



Let's get a better look at that new selection object:

```
const dots = bounds.selectAll("circle")
  .data(dataset)
  .enter()
console.log(dots)
```

This looks just like any d3 selection object we've manipulated before. Let's append one `<circle>` for each data point. We can use the same `.append()` method we've been using for single-node selection objects and d3 will create one element for each data point.

```
const dots = bounds.selectAll("circle")
  .data(dataset)
  .enter().append("circle")
```

When we load our webpage, we will still have a blank page. However, we will be able to see lots of new empty `<circle>` elements in our **bounds** in the **Elements** panel.

Let's set the position and size of these circles.

```
const dots = bounds.selectAll("circle")
  .data(dataset)
  .enter().append("circle")
  .attr("cx", d => xScale(xAccessor(d)))
  .attr("cy", d => yScale(yAccessor(d)))
  .attr("r", 5)
```

We can write the same code we would write for a single-node selection object. Any attribute values that are functions will be passed each data point individually. This helps keep our code concise and consistent.

Let's make these dots a lighter color to help them stand out.

```
.attr("fill", "cornflowerblue")
```

Data join exercise

Here's a quick example to help visualize the data join concept. We're going to split the dataset in two and draw both parts separately. Temporarily comment out your finished dots code so we have a clear slate to work with. We'll put it back when we're done with this exercise.

Let's add a function called `drawDots()` that mimics our dot drawing code. This function will select all existing circles, join them with a provided dataset, and draw any new circles with a provided color.

```
function drawDots(dataset, color) {  
  const dots = bounds.selectAll("circle").data(dataset)  
  
  dots  
    .enter().append("circle")  
    .attr("cx", d => xScale(xAccessor(d)))  
    .attr("cy", d => yScale(yAccessor(d)))  
    .attr("r", 5)  
    .attr("fill", color)  
}  
}
```

Let's call this function with part of our dataset. The color doesn't matter much — let's go with a dark grey.

```
drawDots(dataset.slice(0, 200), "darkgrey")
```

We should see some of our dots drawn on the page.



grey dots

After one second, let's call the function again with our whole dataset, this time with a blue color. We're adding a timeout to help distinguish between the two sets of dots.

```
setTimeout(() => {
  drawDots(dataset, "cornflowerblue")
}, 1000)
```

When you refresh your webpage, you should see a set of grey dots, then a set of blue dots one second later.



grey + blue dots

Each time we run `drawDots()`, we're setting the color of only **new** circles. This explains why the grey dots stay grey. If we wanted to set the color of all circles, we could re-select all circles and set their fill on the new selection:

```
function drawDots(dataset, color) {
  const dots = bounds.selectAll("circle").data(dataset)

  dots.enter().append("circle")
  bounds.selectAll("circle")
    .attr("cx", d => xScale(xAccessor(d)))
    .attr("cy", d => yScale(yAccessor(d)))
    .attr("r", 5)
    .attr("fill", color)
}
```

In order to keep the chain going, d3 selection objects have a `merge()` method that will combine the current selection with another selection. In this case, we could combine the new `enter` selection with the original `dots` selection, which will return the full list of dots. When we set attributes on the new merged selection, we'll be updating all of the dots.

```
function drawDots(dataset, color) {
  const dots = bounds.selectAll("circle").data(dataset)

  dots
    .enter().append("circle")
    .merge(dots)
    .attr("cx", d => xScale(xAccessor(d)))
    .attr("cy", d => yScale(yAccessor(d)))
    .attr("r", 5)
    .attr("fill", color)
}
```

.join()

Since [d3-selection version 1.4.0²⁰](#), there is a new `.join()`²¹ method that helps to cut down on this code. `.join()` is a shortcut for running `.enter()`, `.append()`, `.merge()`, and some other methods we haven't covered yet. This allows us to write the following code instead:

²⁰<https://github.com/d3/d3-selection/releases/tag/v1.4.0>

²¹https://github.com/d3/d3-selection/#selection_join

```
function drawDots(dataset, color) {  
  const dots = bounds.selectAll("circle").data(dataset)  
  
  dots.join("circle")  
    .attr("cx", d => xScale(xAccessor(d)))  
    .attr("cy", d => yScale(yAccessor(d)))  
    .attr("r", 5)  
    .attr("fill", color)  
}  
}
```

While `.join()` is a great addition to d3, it's still beneficial to understand the `.enter()`, `.append()`, and `.merge()` methods. Most existing d3 code will use these methods, and it's important to understand the basics before getting fancy.

Don't worry if this pattern still feels new — we'll reinforce and build on what we've learned when we talk about transitions. For now, let's delete this example code, uncomment our finished dots code, and move on with our scatter plot!

Draw peripherals

Let's finish up our chart by drawing our axes, starting with the x axis.

We want our x axis to be:

- a line across the bottom
- with spaced “tick” marks that have...
- labels for values per tick
- a label for the axis overall

To do this, we'll create our axis generator using `d3.axisBottom()`, then pass it:

- our x scale so it knows what ticks to make (from the domain) and
- what size to be (from the range).

code/02-making-a-scatterplot/completed/draw-scatter.js

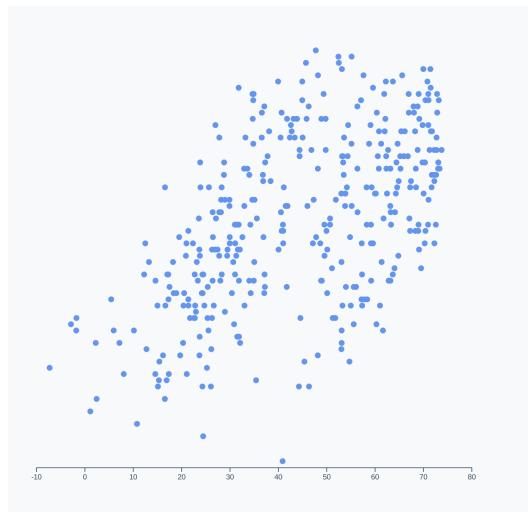
```
76 const xAxisGenerator = d3.axisBottom()  
77   .scale(xScale)
```

Next, we'll use our `xAxisGenerator()` and call it on a new g element. Remember, we need to translate the x axis to move it to the bottom of the chart bounds.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
79 const xAxis = bounds.append("g")  
80   .call(xAxisGenerator)  
81   .style("transform", `translateY(${dimensions.boundedHeight}px)`)
```

When we render our webpage, we should see our scatter plot with an x axis. As a bonus, we can see how using `.nice()` on our scale ensures that our axis ends in round values.



Scatterplot with an x axis

Let's expand on our knowledge and create labels for our axes. Drawing text in an SVG is fairly straightforward - we need a `<text>` element, which can be positioned with an `x` and a `y` attribute. We'll want to position it horizontally centered and slightly above the bottom of the chart.

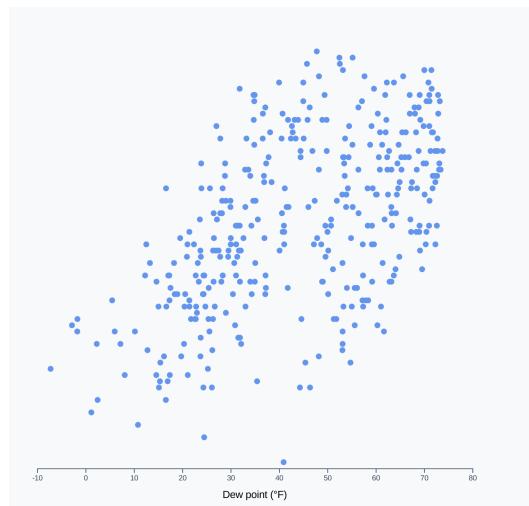
<text> elements will display their children as text — we can set that with our selection's `.html()` method.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
83 const xAxisLabel = xAxis.append("text")
84   .attr("x", dimensions.boundedWidth / 2)
85   .attr("y", dimensions.margin.bottom - 10)
86   .attr("fill", "black")
87   .style("font-size", "1.4em")
88   .html("Dew point (&deg;F)")
```

We need to explicitly set the text fill to black because it inherits a fill value of none that d3 sets on the axis <g> element.

Great! Now we can see a label underneath our x axis.



Scatter plot with an x axis label

Almost there! Let's do the same thing with the y axis. First, we need an axis generator. D3 axes can be customized in many ways. An easy way to cut down on visual clutter

is to tell our axis to aim for a certain number with the `ticks` method. Let's aim for 4 ticks, which should give the viewer enough information.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
90 const yAxisGenerator = d3.axisLeft()  
91   .scale(yScale)  
92   .ticks(4)
```

Note that the resulting axis won't necessarily have exactly 4 ticks. D3 will take the number as a suggestion and aim for that many ticks, but also trying to use friendly intervals. Check out some of the internal logic [in the d3-array code^a](#)—see how it's attempting to use intervals of 10, then 5, then 2?

There are many ways to configure the ticks for a d3 axis — find them all in [the docs^b](#). For example, you can specify their exact values by passing an array of values to `.tickValues()`.

^a<https://github.com/d3/d3-array/blob/master/src/ticks.js#L43>

^bhttps://github.com/d3/d3-axis#axis_ticks

Let's use our generator to draw our y axis.

code/02-making-a-scatterplot/completed/draw-scatter.js

```
94 const yAxis = bounds.append("g")  
95   .call(yAxisGenerator)
```

To finish up, let's draw the y axis label in the middle of the y axis, just inside the left side of the chart wrapper. d3 selection objects also have a `.text()` method that operates similarly to `.html()`. Let's try using that here.

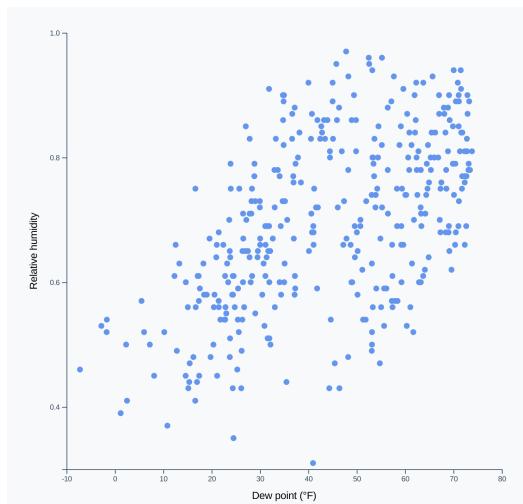
code/02-making-a-scatterplot/completed/draw-scatter.js

```
97 const yAxisLabel = yAxis.append("text")
98   .attr("x", -dimensions.boundedHeight / 2)
99   .attr("y", -dimensions.margin.left + 10)
100  .attr("fill", "black")
101  .style("font-size", "1.4em")
102  .text("Relative humidity")
```

We'll need to rotate this label to fit next to the y axis. To rotate it around its center, we can set its CSS property `text-anchor` to `middle`.

```
.style("transform", "rotate(-90deg)")
.style("text-anchor", "middle")
```

And just like that, we've drawn our scatter plot!



Finished scatterplot

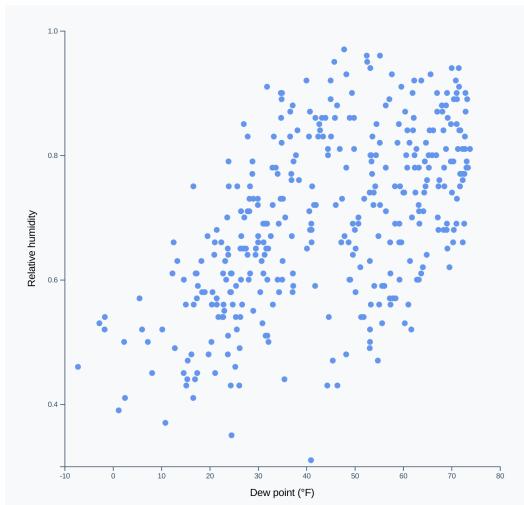
Initialize interactions

The next step in our chart-drawing checklist is setting up interactions and event listeners. We'll go over this in detail in **Chapter 5**.

Looking at our chart

Now that we've finished drawing our scatter plot, we can step back and see what we can learn from displaying the data in this manner. Without running statistical analyses (such as the Pearson Correlation Coefficient or Mutual Analyses), we won't be able to make any definitive statements about whether or not our metrics are correlated. However, we can still get a sense of how they relate to one another.

Looking at the plotted dots, they do seem to group around an invisible line from the bottom left to the top right of the chart.



Finished scatterplot

Generally, it seems like we were correct in guessing that a high humidity would likely coincide with a high dew point. We'll discuss the different types of patterns we might see in a scatter plot in Chapter 10.

Extra credit: adding a color scale

Remember when I said that we could introduce another metric? Let's bring in the amount of cloud cover for each day. In our dataset, each datapoint records the

cloud cover for that day. Let's show how the amount of cloud cover varies based on humidity and dew point by adding a color scale.

(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"

Our dataset

Looking at a value in our dataset, we can see that the amount of cloud cover exists at the key `cloudCover`. Let's add another data accessor function near the top of our file:

```
const colorAccessor = d => d.cloudCover
```

Next up, let's create another scale at the bottom of our `Create scales` step.

So far, we've only looked at linear scales that convert numbers to other numbers. Scales can also convert a number into a color — we just need to replace the `domain` with a range of colors.

Let's make low cloud cover days be light blue and very cloudy days dark blue - that's a good semantic mapping.

```
const colorScale = d3.scaleLinear()  
  .domain(d3.extent(dataset, colorAccessor))  
  .range(["skyblue", "darkslategrey"])
```

Let's test it out - if we log `colorScale(0.1)` to the console, we should see a color value, such as `rgb(126, 193, 219)`. Perfect!

Choosing colors is a complicated topic! We'll learn about color spaces, good color scales, and picking colors in [Chapter 7](#).

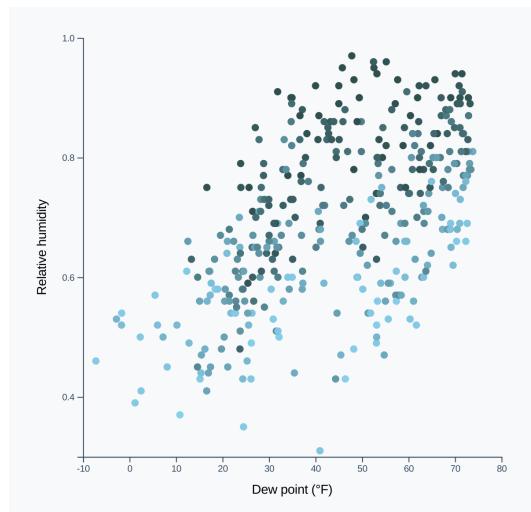
All that's left to do is to update how we set the `fill` of each dot. Let's find where we're doing that now.

```
.attr("fill", "cornflowerblue")
```

Instead of making every dot blue, let's use our `colorAccessor()` to grab the precipitation value, then pass that into our `colorScale()`.

```
.attr("fill", d => colorScale(colorAccessor(d)))
```

When we refresh our webpage, we should see our finished scatter plot with dots of various blues.



Scatterplot with a color scale

For a complete, accessible chart, it would be a good idea to add a legend to explain what our colors mean. Stay tuned! We'll learn how to add a color scale legend in [Chapter 6](#).

This chapter was jam-packed with new concepts — we learned about data joins, `<text>` SVG elements, color scales, and more. Give yourself a pat on the back for making it through! Next up, we'll create a bar chart and learn some new concepts.

Making a Bar Chart

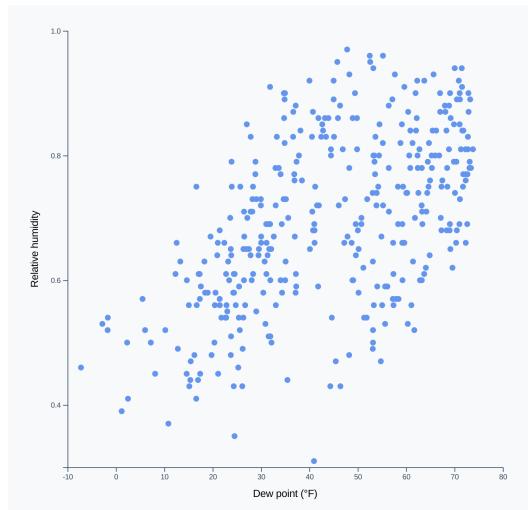
We'll walk through one last "basic" chart — once finished, you'll feel very comfortable with each step and we'll move on to even more exciting concepts like animations and interactions.

Deciding the chart type

Another type of question that we can ask our dataset is: what does the *distribution* of a metric look like? For example:

- What kinds of humidity values do we have?
- Does the humidity level generally stay around one value, with a few very humid days and a few very dry days?
- Or does it vary consistently, with no standard value?

Looking at the scatter plot we just made, we can see the daily humidity values from the dots' vertical placement.



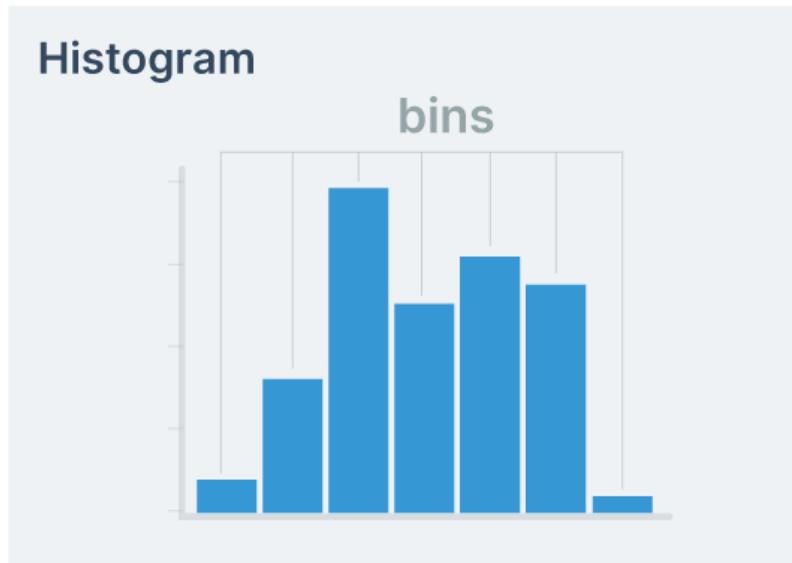
Finished scatter plot

But it's hard to answer our questions - do most of our dots fall close to the middle of the chart? We're not entirely sure.

Instead, let's make a histogram.

Histogram

A *histogram* is a bar chart that shows the *distribution* of one metric, with the metric values on the x axis and the frequency of values on the y axis.



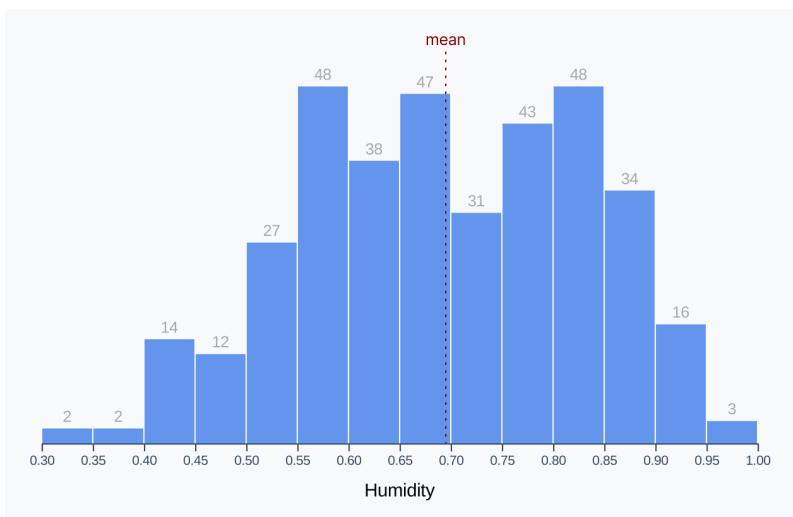
Histogram graphic

In order to show the frequency, values are placed in **equally-sized bins** (visualized as individual bars). For example, we could make bins for dew point temperatures that span 10 degrees - these would look something like [0-10, 10-20, 20-30, ...]. A dew point of 15 degrees would be counted in the second bin: 10-20.

The number of and size of bins is up to the implementor - you could have a histogram with only 3 bins or one with 100 bins! There are standards that can be followed (feel free to [check out d3's built-in formulas²²](#)), but we can generally decide the number based on what suits the data and what's easy to read.

Our goal is to make a histogram of humidity values. This will show us the distribution of humidity values and help answer our questions. Do most days stay around the same level of humidity? Or are there two types of days: humid and dry? Are there crazy humid days?

²²<https://github.com/d3/d3-array#bin-thresholds>



Finished humidity histogram



To interpret the above histogram, it shows that we have **48** days in our dataset with a humidity value between **0.55** and **0.6**

For extra credit, we'll generalize our histogram function and loop through eight metrics in our dataset - creating many histograms to compare!

**Many histograms**

Let's dig in.

Chart checklist

To start, let's look over our chart-making checklist to remind ourselves of the necessary steps.

1. Access data
2. Create dimensions
3. Draw canvas
4. Create scales
5. Draw data
6. Draw peripherals
7. Set up interactions

We'll breeze through most of these steps, reinforcing what we've already learned.

Access data

As usual, make sure your node server is running (`live-server`) and point your browser at `http://localhost:8080/03-making-a-bar-chart/draft/`. Inside the

`/code/03-making-a-bar-chart/draft/draw-bars.js`

file, let's grab the data from our JSON file, waiting until it's loaded to continue.

`code/03-making-a-bar-chart/completed/draw-bars.js`

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

As usual, the completed chart code is available if you need a hint, this time at `/code/03-making-a-bar-chart/completed/draw-bars.js`.

This time, we're only interested in one metric for the whole chart. Remember, the y axis is plotting the *frequency* (i.e. the number of occurrences) of the metric whose values are on the x axis. So instead of an `xAccessor()` and `yAccessor()`, we define a single `metricAccessor()`.

`const metricAccessor = d => d.humidity`

Create dimensions

Histograms are easiest to read when they are wider than they are tall. Let's set the width before defining the rest of our dimensions so we can use it to calculate the height. We'll also be able to quickly change the width later and keep the same aspect ratio for our chart.



Chart design tip: Histograms are easiest to read when they are wider than they are tall.

Instead of filling the whole window, let's prepare for multiple histograms and keep our chart small. That way, the charts can stack horizontally and vertically, depending on the screen size.

code/03-making-a-bar-chart/completed/draw-bars.js

```
11 const width = 600
```

Alright! Let's use the `width` to set the `width` and `height` of our chart. We'll leave a larger margin on the top to account for the bar labels, which we'll position above each bar.

code/03-making-a-bar-chart/completed/draw-bars.js

```
12 let dimensions = {  
13   width: width,  
14   height: width * 0.6,  
15   margin: {  
16     top: 30,  
17     right: 10,  
18     bottom: 50,  
19     left: 50,  
20   },  
21 }
```

Remember, our **wrapper** encompasses the whole chart. If we subtract our margins, we'll get the size of our **bounds** which contain any data elements.

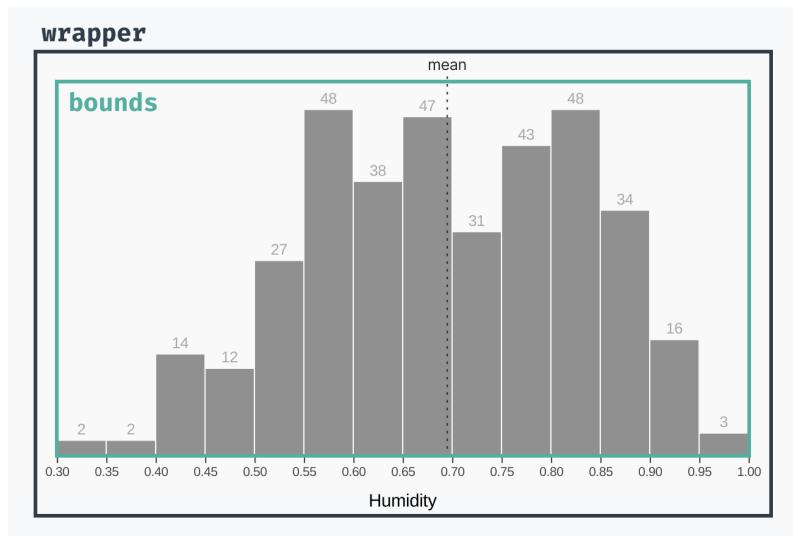


Chart terminology

Now that we know the size of our **wrapper** and **margins**, we can calculate the size of our **bounds**.

code/03-making-a-bar-chart/completed/draw-bars.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
```

Draw canvas

Let's create our **wrapper** element. Try to write this code without looking first. Like we've done before, we want to select the existing element, add a new `<svg>` element, and set its **width** and **height**.

code/03-making-a-bar-chart/completed/draw-bars.js

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

How far did you get without looking? Let's try that again for this next part: creating our **bounds**. As a reminder, our **bounds** are a `<g>` element that will contain our main chart bits and be shifted to respect our **top** and **left** margins.

code/03-making-a-bar-chart/completed/draw-bars.js

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

Perfect! Let's make our scales.

Create scales

Our x scale should look familiar to the ones we've made in the past. We need a scale that will convert humidity levels into pixels-to-the-right. Since both the **domain** and the **range** are continuous numbers, we'll use our friend `d3.scaleLinear()`.

Let's also use `.nice()`, which we learned in **Chapter 2**, to make sure our axis starts and ends on round numbers.

code/03-making-a-bar-chart/completed/draw-bars.js

```
45 const xScale = d3.scaleLinear()  
46   .domain(d3.extent(dataset, metricAccessor))  
47   .range([0, dimensions.boundedWidth])  
48   .nice()
```

Rad. Now we need to create our `yScale`.

But wait a minute! We can't make a `y` scale without knowing the range of frequencies we need to cover. Let's create our data bins first.

Creating Bins

How can we split our data into bins, and what size should those bins be? We could do this manually by looking at the domain and organizing our days into groups, but that sounds tedious.

Thankfully, we can use `d3-array`'s `d3.histogram()` method to create a bin generator. This generator will convert our dataset into an array of bins - we can even choose how many bins we want!

Let's create a new generator:

code/03-making-a-bar-chart/completed/draw-bars.js

```
50 const binsGenerator = d3.histogram()
```

Similar to making a scale, we'll pass a `domain` to the generator to tell it the range of numbers we want to cover.

code/03-making-a-bar-chart/completed/draw-bars.js

```
50 const binsGenerator = d3.histogram()  
51   .domain(xScale.domain())
```

Next, we'll need to tell our generator how to get the `humidity` value, since our dataset contains objects instead of values. We can do this by passing our `metricAccessor()` function to the `.value()` method.

code/03-making-a-bar-chart/completed/draw-bars.js

```
50 const binsGenerator = d3.histogram()  
51   .domain(xScale.domain())  
52   .value(metricAccessor)
```

We can also tell our generator that we want it to aim for a specific number of bins. When we create our bins, we won't necessarily get this exact amount, but it should be close.

Let's aim for 13 bins — this should make sure we have enough granularity to see the shape of our distribution without too much noise. Keep in mind that the number of bins is the number of **thresholds + 1**.

code/03-making-a-bar-chart/completed/draw-bars.js

```
50 const binsGenerator = d3.histogram()  
51   .domain(xScale.domain())  
52   .value(metricAccessor)  
53   .thresholds(12)
```

Great! Our bin generator is ready to go. Let's create our bins by feeding it our data.

code/03-making-a-bar-chart/completed/draw-bars.js

```
55 const bins = binsGenerator(dataset)
```

Let's take a look at these bins by logging them to the console: `console.log(bins)`.

```
draw-bars.js:48
(15) [Array(0), Array(2), Array(2), Array(14), Array(12), Array(27), Array(48),
▼ Array(38), Array(47), Array(31), Array(43), Array(48), Array(34), Array(16), Arr
ay(3)] 1
▶ 0: [x0: 0.3, x1: 0.3000000000000004]
▶ 1: (2) [{}], {}, x0: 0.3000000000000004, x1: 0.3500000000000003]
▶ 2: (2) [{}], {}, x0: 0.3500000000000003, x1: 0.4]
▶ 3: (14) [{}], {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, ..., ...
▶ 4: (12) [{}], {}, {}, {}, {}, {}, {}, {}, {}, {}, {}, ..., x0: 0.45...
▶ 5: (27) [{}], {}, {}, {}, {}, {}, {}, {}, {}, {}, ..., ..., ..., ...
▶ 6: (48) [{}], {}, {}, {}, {}, {}, {}, {}, ..., ..., ..., ..., ...
▶ 7: (38) [{}], {}, {}, {}, {}, {}, {}, ..., ..., ..., ..., ..., ...
▶ 8: (47) [{}], {}, {}, {}, {}, ..., ..., ..., ..., ..., ..., ...
▶ 9: (31) [{}], {}, {}, {}, ..., ..., ..., ..., ..., ..., ..., ...
▶ 10: (43) [{}], {}, {}, ..., ..., ..., ..., ..., ..., ..., ..., ...
▶ 11: (48) [{}], {}, {}, ..., ..., ..., ..., ..., ..., ..., ..., ...
▶ 12: (34) [{}], {}, {}, ..., ..., ..., ..., ..., ..., ..., ...
▶ 13: (16) [{}], {}, {}, ..., ..., ..., ..., ..., ..., ..., ...
▶ 14: (3) [{}], {}, ..., x0: 0.9500000000000001, x1: 1]
length: 15
▶ __proto__: Array(0)
```

logged bins

Each bin is an array with the following structure:

- each item is a matching data point. For example, the first bin has no matching days — this is likely because we used `.nice()` to round out our x scale.
- there is an `x0` key that shows the lower bound of included humidity values (inclusive)
- there is an `x1` key that shows the upper bound of included humidity values (exclusive). For example, a bin with a `x1` value of `1` will include values **up to 1**, but not `1` itself

Note how there are 16 bins in my example — our bin generator was aiming for 13 bins but decided that 16 bins were more appropriate. This was a good decision, creating bins with a sensible size of `0.05`. If our bin generator had been more strict about the number of bins, our bins would have ended up with a size of `0.06666667`, which is harder to reason about. To extract insights from a chart,

readers will mentally convert awkward numbers into rounder numbers to make sense of them. Let's do that work for them.

If we want, we can specify an exact number of bins by instead passing an array of thresholds. For example, we could specify 5 bins with `.thresholds([0, 0.2, 0.4, 0.6, 0.8, 1])`.

Creating the y scale

Okay great, now we can use these bins to create our y scale. First, let's create a y accessor function and throw it at the top of our file. Now that we know the shape of the data that we'll use to create our data elements, we can specify how to access the y value in one place.

`code/03-making-a-bar-chart/completed/draw-bars.js`

```
7 const yAccessor = d => d.length
```

Let's use our new accessor function and our bins to create that y scale. As usual, we'll want to make a linear scale. This time, however, we'll want to start our y axis at zero.

Previously, we wanted to represent the extent of our data since we were plotting metrics that had no logical bounds (temperature and humidity level). But the number of days that fall in a bin is bounded at 0 — you can't have negative days in a bin!

Instead of using `d3.extent()`, we can use another method from `d3-array`: `d3.max()`. This might sound familiar — we've used its counterpart, `d3.min()` in Chapter 2. `d3.max()` takes the same arguments: an array and an accessor function.

Note that we're passing `d3.max()` our `bins` instead of our original dataset — we want to find the maximum number of days in a bin, which is only available in our computed `bins` array.

code/03-making-a-bar-chart/completed/draw-bars.js

```
57 const yScale = d3.scaleLinear()  
58   .domain([0, d3.max(bins, yAccessor)])  
59   .range([dimensions.boundedHeight, 0])
```

Let's use `.nice()` here as well to give our bars a round top number.

code/03-making-a-bar-chart/completed/draw-bars.js

```
57 const yScale = d3.scaleLinear()  
58   .domain([0, d3.max(bins, yAccessor)])  
59   .range([dimensions.boundedHeight, 0])  
60   .nice()
```

Draw data

Here comes the fun part! Our plan is to create one bar for each bin, with a label on top of each bar.

We'll need one bar for each item in our `bins` array — this is a sign that we'll want to use the **data bind concept** we learned in [Chapter 2](#).

Let's first create a `<g>` element to contain our bins. This will help keep our code organized and isolate our bars in the DOM.

code/03-making-a-bar-chart/completed/draw-bars.js

```
64 const binsGroup = bounds.append("g")
```

Because we have more than one element, we'll bind each data point to a `<g>` SVG element. This will let us group each bin's **bar** and **label**.

To start, we'll select all existing `<g>` elements within our `binsGroup` (*there aren't any yet, but we're creating a selection object that points to the right place*). Then we'll use `.data()` to bind our `bins` to the selection.

code/03-making-a-bar-chart/completed/draw-bars.js

```
66 const binGroups = binsGroup.selectAll("g")
67   .data(bins)
```

Next, we'll create our `<g>` elements, using `.enter()` to target the new bins (*hint: all of them*).

code/03-making-a-bar-chart/completed/draw-bars.js

```
66 const binGroups = binsGroup.selectAll("g")
67   .data(bins)
68   .enter().append("g")
```

The above code will create one new `<g>` element for each bin. We're going to place our bars within this group.

Next up we'll draw our bars, but first we should calculate any constants that we'll need. Like a warrior going into battle, we want to prepare our weapons before things heat up.

In this case, the only constant that we can set ahead of time is **the padding between bars**. Giving them some space helps distinguish individual bars, but we don't want them too far apart - that will make them hard to compare and take away from the overall shape of the distribution.



Chart design tip: putting a space between bars helps distinguish individual bars

code/03-making-a-bar-chart/completed/draw-bars.js

```
70 const barPadding = 1
```

Now we are armed warriors and are ready to charge into battle! Each bar is a rectangle, so we'll append a `<rect>` to each of our `<g>` elements.

code/03-making-a-bar-chart/completed/draw-bars.js

```
71 const barRects = binGroups.append("rect")
72   .attr("x", d => xScale(d.x0) + barPadding / 2)
73   .attr("y", d => yScale(yAccessor(d)))
```

Remember, <rect>s need four attributes: **x**, **y**, **width**, and **height**.

Let's start with the **x** value, which will corresponds to the *left* side of the bar. The bar will start at the lower bound of the bin, which we can find at the **x0** key.

But **x0** is a humidity level, not a pixel. So let's use **xScale()** to convert it to pixel space.

Lastly, we need to offset it by the **barPadding** we set earlier.

code/03-making-a-bar-chart/completed/draw-bars.js

```
72   .attr("x", d => xScale(d.x0) + barPadding / 2)
73   .attr("y", d => yScale(yAccessor(d)))
74   .attr("width", d => d3.max([
```

We could create accessor functions for the **x0** and **x1** properties of each bin if we were concerned about the structure of our bins changing. In this case, it would be overkill since:

1. we didn't specify the structure of each bin, **d3.histogram()** did
2. we're not going to change the way we access either of these values since they're built in to **d3.histogram()**
3. the way we access these properties is very straightforward. If the values were more nested or required computation, we could definitely benefit from accessor functions.

Next, we'll specify the <rect>'s **y** attribute which corresponds to the top of the bar. We'll use our **yAccessor()** to grab the frequency and use our scale to convert it into pixel space.

code/03-making-a-bar-chart/completed/draw-bars.js

```
73     .attr("y", d => yScale(yAccessor(d)))  
74     .attr("width", d => d3.max([  
75         0,
```

To find the width of a bar, we need to subtract the x0 position of the left side of the bar from the x1 position of the right side of the bar.

We'll need to subtract the bar padding from the total width to account for spaces between bars. Sometimes we'll get a bar with a width of 0, and subtracting the barPadding will bring us to -1. To prevent passing our `<rect>`s a negative width, we'll wrap our value with `d3.max([0, width])`.

code/03-making-a-bar-chart/completed/draw-bars.js

```
74     .attr("width", d => d3.max([  
75         0,  
76         xScale(d.x1) - xScale(d.x0) - barPadding  
77     ]))  
78     .attr("height", d => dimensions.boundedHeight  
79         - yScale(yAccessor(d)))
```

Lastly, we'll calculate the bar's height by subtracting the y value from the bottom of the y axis. Since our y axis starts from 0, we can use our `boundedHeight`.

code/03-making-a-bar-chart/completed/draw-bars.js

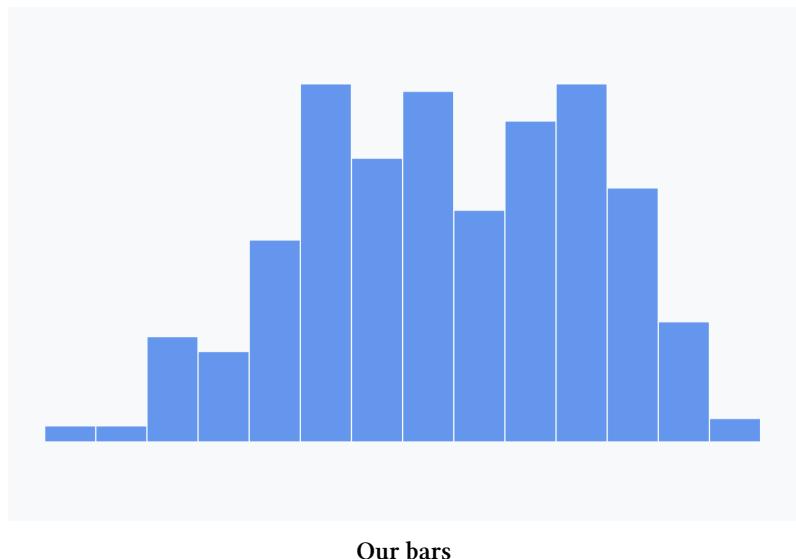
```
78     .attr("height", d => dimensions.boundedHeight  
79         - yScale(yAccessor(d)))  
80     )
```

Let's put that all together and change the bar `fill` to blue.

code/03-making-a-bar-chart/completed/draw-bars.js

```
71 const barRects = binGroups.append("rect")
72   .attr("x", d => xScale(d.x0) + barPadding / 2)
73   .attr("y", d => yScale(yAccessor(d)))
74   .attr("width", d => d3.max([
75     0,
76     xScale(d.x1) - xScale(d.x0) - barPadding
77   ]))
78   .attr("height", d => dimensions.boundedHeight
79     - yScale(yAccessor(d))
80   )
81   .attr("fill", "cornflowerblue")
```

Alright! When we refresh our webpage, we'll see the beginnings of our histogram!



Adding Labels

Let's add labels to show the count for each of these bars.

We can keep our chart clean by only adding labels to bins with any relevant days — having `0`s in empty spaces is unhelpful visual clutter. We can identify which bins have no data by their lack of a bar, no need to call it out with a label.

d3 selections have a `.filter()` method that acts the same way the native Array method does. `.filter()` accepts one parameter: a function that accepts one data point and returns a value. Any items in our dataset who return a **falsy** value will be removed.

By “falsy”, we’re referring to any value that evaluates to `false`. Maybe surprisingly, this includes values other than `false`, such as `0`, `null`, `undefined`, `""`, and `NaN`. Keep in mind that empty arrays `[]` and object `{}` evaluate to **truthy**. If you’re curious, [read more here^a](https://developer.mozilla.org/en-US/docs/Glossary/Falsy).

^a<https://developer.mozilla.org/en-US/docs/Glossary/Falsy>

We can use `yAccessor()` as shorthand for `d => yAccessor(d) != 0` because `0` is **falsy**.

code/03-making-a-bar-chart/completed/draw-bars.js

83

```
const barText = binGroups.filter(yAccessor)
```

Since these labels are just text, we’ll want to use the SVG `<text>` element we’ve been using for our axis labels.

code/03-making-a-bar-chart/completed/draw-bars.js

83

```
const barText = binGroups.filter(yAccessor)  
    .append("text")
```

Remember, `<text>` elements are positioned with `x` and `y` attributes. The label will be centered horizontally above the bar — we can find the center of the bar by adding half of the bar’s width (*the right side minus the left side*) to the left side of the bar.

code/03-making-a-bar-chart/completed/draw-bars.js

```
83 const barText = binGroups.filter(yAccessor)  
84   .append("text")  
85     .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)
```

Our <text>'s y position will be similar to the <rect>'s y position, but let's shift it up by 5 pixels to add a little gap.

code/03-making-a-bar-chart/completed/draw-bars.js

```
83 const barText = binGroups.filter(yAccessor)  
84   .append("text")  
85     .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)  
86     .attr("y", d => yScale(yAccessor(d)) - 5)
```

Next, we'll display the count of days in the bin using our `yAccessor()` function.
Note: again, we can use `yAccessor()` as shorthand for `d => yAccessor(d)`.

code/03-making-a-bar-chart/completed/draw-bars.js

```
83 const barText = binGroups.filter(yAccessor)  
84   .append("text")  
85     .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)  
86     .attr("y", d => yScale(yAccessor(d)) - 5)  
87     .text(yAccessor)
```

We can use the CSS `text-anchor` property to horizontally align our text — this is a much simpler solution than compensating for text width when we set the `x` attribute.

code/03-making-a-bar-chart/completed/draw-bars.js

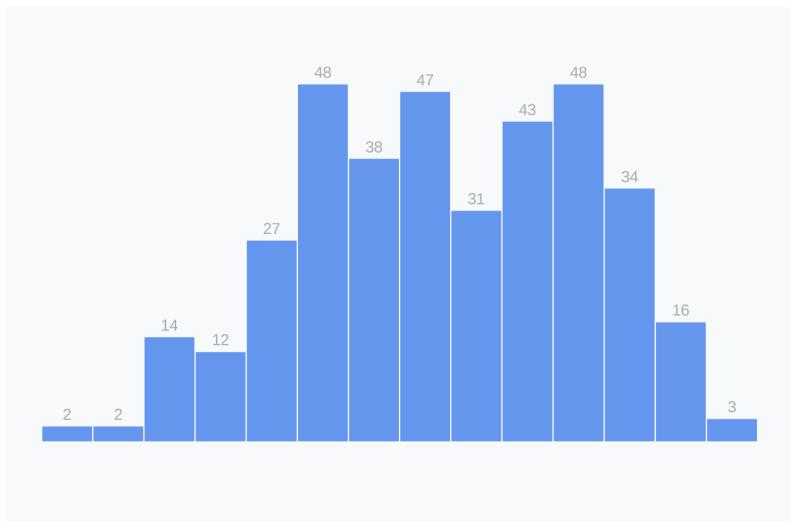
```
83 const barText = binGroups.filter(yAccessor)
84   .append("text")
85     .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)
86     .attr("y", d => yScale(yAccessor(d)) - 5)
87     .text(yAccessor)
88     .style("text-anchor", "middle")
```

After adding a few styles to decrease the visual importance of our labels...

code/03-making-a-bar-chart/completed/draw-bars.js

```
83 const barText = binGroups.filter(yAccessor)
84   .append("text")
85     .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)
86     .attr("y", d => yScale(yAccessor(d)) - 5)
87     .text(yAccessor)
88     .style("text-anchor", "middle")
89     .attr("fill", "darkgrey")
90     .style("font-size", "12px")
91     .style("font-family", "sans-serif")
```

...we should see the count of days for each of our bars!



Our bars with labels

Extra credit

When looking at the shape of a distribution, it can be helpful to know where the mean is.

The mean is just the average — the center of a set of numbers. To calculate the mean, you would divide the sum by the number of values. For example, the mean of [1, 2, 3, 4, 5] would be $(1 + 2 + 3 + 4 + 5) / 5 = 3$.

Instead of calculating the mean by hand, we can use `d3.mean()` to grab that value. Like many d3 methods we've used, we pass the dataset as the first parameter and an optional accessor function as the second.

[code/03-making-a-bar-chart/completed/draw-bars.js](#)

```
const mean = d3.mean(dataset, metricAccessor)
```

Great! Let's see how comfortable we are with drawing an unfamiliar SVG element: `<line>`. A `<line>` element will draw a line between two points: `[x1, y1]` and `[x2, y2]`. Using this knowledge, let's add a line to our bounds that is:

- at the mean humidity level,

- starting 20px above our chart, and
- ending at our x axis.

How close can you get before looking at the following code?

code/03-making-a-bar-chart/completed/draw-bars.js

```
94 const meanLine = bounds.append("line")
95   .attr("x1", xScale(mean))
96   .attr("x2", xScale(mean))
97   .attr("y1", -15)
98   .attr("y2", dimensions.boundedHeight)
```

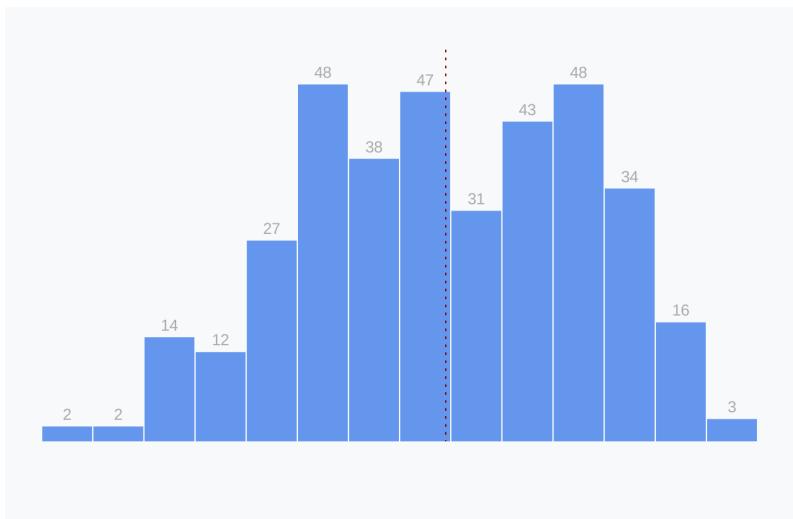
Let's add some styles to the line so we can see it (by default, `<line>`s have no stroke color) and to distinguish it from an axis. SVG element strokes can be split into dashes with the `stroke-dasharray` attribute. The lines alternate between the stroke color and transparent, starting with transparent. We define the line lengths with a space-separated list of values (which will be repeated until the line is drawn).

Let's make our lines dashed with a 2px long maroon dash and a 4px long gap.

code/03-making-a-bar-chart/completed/draw-bars.js

```
94 const meanLine = bounds.append("line")
95   .attr("x1", xScale(mean))
96   .attr("x2", xScale(mean))
97   .attr("y1", -15)
98   .attr("y2", dimensions.boundedHeight)
99   .attr("stroke", "maroon")
100  .attr("stroke-dasharray", "2px 4px")
```

Give yourself a pat on the back for drawing your first `<line>` element!



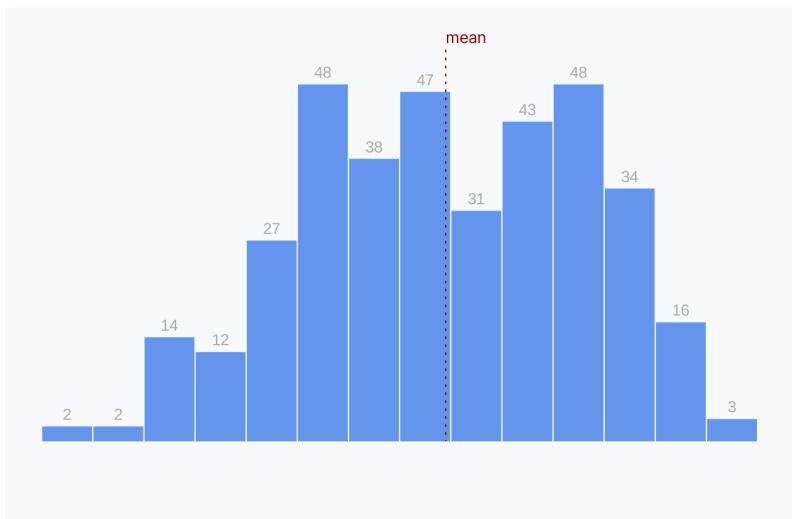
Our bars with labels and the mean

Let's label our line to clarify to readers what it represents. We'll want to add a `<text>` element in the same position as our line, but 5 pixels higher to give a little gap.

`code/03-making-a-bar-chart/completed/draw-bars.js`

```
102 const meanLabel = bounds.append("text")
103   .attr("x", xScale(mean))
104   .attr("y", -20)
105   .text("mean")
106   .attr("fill", "maroon")
107   .style("font-size", "12px")
```

Hmm, we can see the text but it isn't horizontally centered with our line.



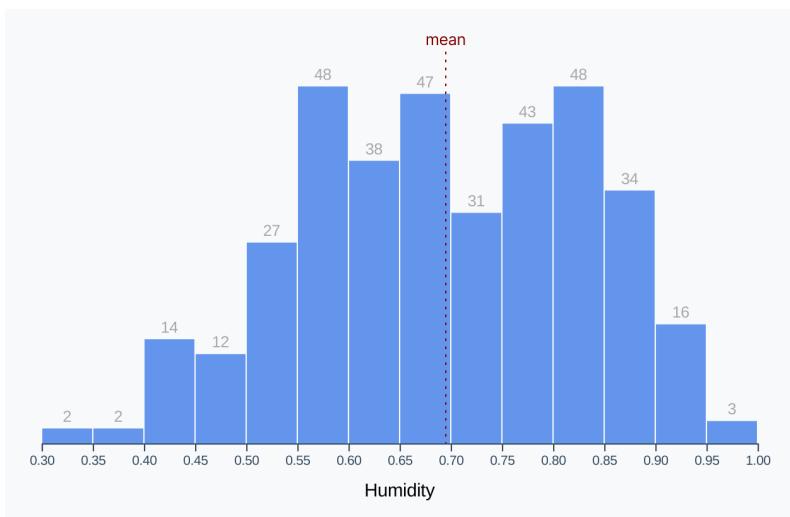
Our bars with a mean label

Let's center our text by adding the CSS property `text-anchor: middle`. This is a property specifically for setting the horizontal alignment of text in SVG.

[code/03-making-a-bar-chart/completed/draw-bars.js](#)

```
102 const meanLabel = bounds.append("text")
103   .attr("x", xScale(mean))
104   .attr("y", -20)
105   .text("mean")
106   .attr("fill", "maroon")
107   .style("font-size", "12px")
108   .style("text-anchor", "middle")
```

Perfect! Now our mean line is clear to our readers.



Our bars with a mean label, centered horizontally

Draw peripherals

As usual, our last task here is to draw our axes. But we're in for a treat! Since we're labeling the y value of each of our bars, we won't need a y axis. We just need an x axis and we're set!

We'll start by making our axis generator — our axis will be along the bottom of the chart so we'll be using `d3.axisBottom()`.

`code/03-making-a-bar-chart/completed/draw-bars.js`

```
112 const xAxisGenerator = d3.axisBottom()  
113   .scale(xScale)
```

Then we'll use our new axis generator to create an axis, then shift it below our bounds.

code/03-making-a-bar-chart/completed/draw-bars.js

```
115 const xAxis = bounds.append("g")
116   .call(xAxisGenerator)
117     .style("transform", `translateY(${dimensions.boundedHeight}px)`)
```

And lastly, let's throw a label on there to make it clear what the tick labels represent.

```
const xAxisLabel = xAxis.append("text")
  .attr("x", dimensions.boundedWidth / 2)
  .attr("y", dimensions.margin.bottom - 10)
  .attr("fill", "black")
  .style("font-size", "1.4em")
  .text("Humidity")
```

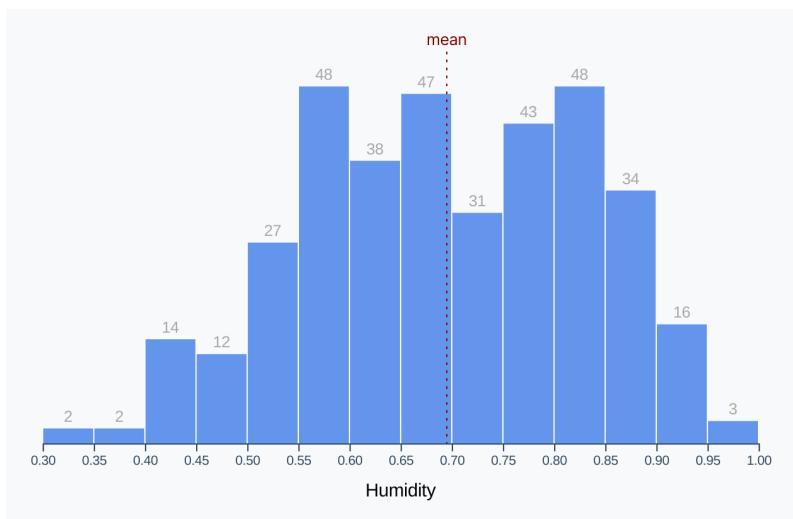
And voila, we're done drawing our peripherals!

Set up interactions

Next, we would set up any chart interactions. We don't have any interactions for this chart, but stay tuned — we'll cover this in the next chapter.

Looking at our chart

Chart finished! Let's take a look at our distribution.



Finished humidity histogram

Our histogram looks somewhere in-between a normal and bimodal distribution. Don't worry if those terms make no sense right now — we'll cover distribution shapes in detail in **Chapter 8**. If you used your own weather data, your histogram could have a very different shape!

Extra credit

Let's generalize our histogram drawing function and create a chart for each weather metric we have access to! This will make sure that we understand what every line of code is doing.

Generalizing our code will also help us to start thinking about handling dynamic data — a core concept when building a dashboard. Drawing a graph with a specific dataset can be difficult, but you get to rely on values being the same every time your code runs. When handling data from an API, your charting functions need to be more robust and able to handle very different datasets.



Finished histogram

Here's the good news: we won't need to rewrite the majority of our code! The main difference is that we'll wrap most of the chart drawing into a new function called `drawHistogram()`.

Which steps do we need to repeat for every chart? Let's look at our checklist again.

1. Access data
2. Create dimensions
3. Draw canvas
4. Create scales
5. Draw data
6. Draw peripherals
7. Set up interactions

All of the histograms will use the same dataset, so we can skip step 1. And every chart will be the same size, so we don't need to repeat step 2 either. However, we want each chart to have its own `svg` element, so we'll need to wrap everything after step 2 .

In the next section, we'll cover ways to make our chart more accessible. We'll be working on the current version of our histogram - make a copy of your current finished histogram in order to come back to it later.

Let's do that — we'll create a new function called `drawHistogram()` that contains all of our code, starting at the point we create our svg. Note that the finished code for this step is in the `/code/03-making-a-bar-chart/completed-multiple/draw-bars.js` file if you're unsure about any of these steps.

```
const drawHistogram = () => {
  const wrapper = d3.select("#wrapper")
  // ... the rest of our chart code
```

What parameters does our function need? The only difference between these charts is the metric we're plotting, so let's add that as an argument.

```
const drawHistogram = metric => {
  // ...
```

But wait, we need to use the metric to update our `metricAccessor()`. Let's grab our accessor functions from our **Access data** step and throw them at the top of our new function. We'll also need our `metricAccessor()` to return the provided metric, instead of hard-coding `d.humidity`.

```
const drawHistogram = metric => {
  const metricAccessor = d => d[metric]
  const yAccessor = d => d.length

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

Great, let's give it a go! At the bottom of our `drawBars()` function, let's run through some of the available metrics (see code example for a list) and pass each of them to our new generalized function.

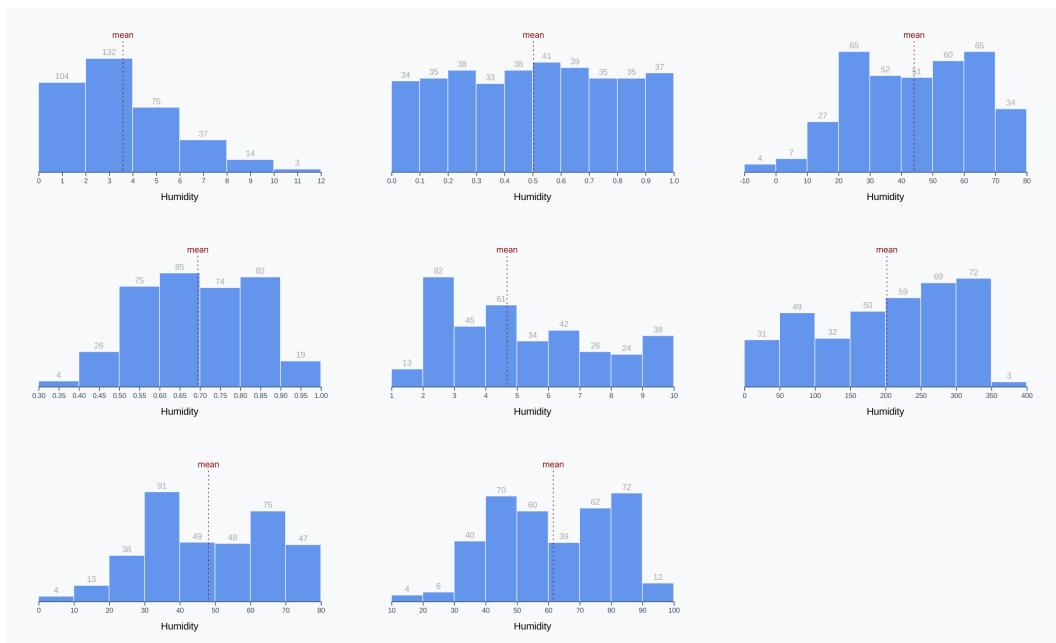
```

const metrics = [
  "windSpeed",
  "moonPhase",
  "dewPoint",
  "humidity",
  "uvIndex",
  "windBearing",
  "temperatureMin",
  "temperatureMax",
]

metrics.forEach(drawHistogram)

```

Alright! Let's see what happens when we refresh our webpage.



Finished histograms, wrong labels

We see multiple histograms, but something is off. Not all of these charts are showing **Humidity!** Let's find the line where we set our x axis label and update that to show our metric instead. Here it is:

```
const xAxisLabel = xAxis.append("text")
// ...
.text("Humidity")
```

We'll set the text to our metric instead, and we can also add a CSS `text-transform` value to help format our metric names. For a production dashboard, we might want to look up a proper label in a metric-to-label map, but this will work in a pinch.

```
const xAxisLabel = xAxis.append("text")
// ...
.text(metric)
.style("text-transform", "capitalize")
```

When we refresh our webpage, we should see our finished histograms.



Finished histogram

Wonderful!

Take a second and observe the variety of shapes of these histograms. What are some insights we can discover when looking at our data in this format?

- the **moon phase** distribution is flat - this makes sense because it's cyclical, consistently going through the same steps all year.
- our **wind speed** is usually around 3 mph, with a long tail to the right that represents a few very windy days. Some days have no wind at all, with an average wind speed of 0.
- our **max temperatures** seem almost bimodal, with the mean falling in between two humps. Looks like New York City spends more days with relatively extreme temperatures (30°F - 50°F or 70°F - 90°F) than with more temperate weather (60°F).

Accessibility

The main goal of any data visualization is for it to be readable. This generally means that we want our elements to be easy to see, text is large enough to read, colors have enough contrast, etc. But what about users who access web pages through screen readers?

We can actually make our charts accessible at a basic level, without putting a lot of effort in. Let's update our histogram so that it's accessible with a screen reader.

If you want to test this out, download the [ChromeVox²³](#) extension for chrome (or use any other screen reader application). If we test it out on our histogram, you'll notice that it doesn't give much information, other than reading all of the text in our chart. That's not an ideal experience.

The main standard for making websites accessible is from **WAI-ARIA**, the Accessible Rich Internet Applications Suite. WAI-ARIA roles, set using a `role` attribute, tell the screen reader what *type of content* an element is.

We'll be updating our completed *single* histogram in this section. If you completed the previous **Extra credit** section, either find your backup of your code or use the completed code in the `/03-making-a-bar-chart/completed/` folder.

²³<https://chrome.google.com/webstore/detail/chromevox/kgejglhpjiefppelpmljglcjboiplfn?hl=en>

The first thing we can do is to give our `<svg>` element a `role` of `figure`²⁴, to alert it that this element is a chart. (This code can go at the bottom of the **Draw canvas** step).

```
wrapper.attr("role", "figure")
```

Next, we can make our chart *tabbable*, by adding a `tabindex` of `0`. This will make it so that a user can hit `tab` to highlight our chart.

There are only two `tabindex` values that you should use:

1. `0`, which puts an element in the `tab` flow, in DOM order
2. `-1`, which takes an element out of the `tab` flow.

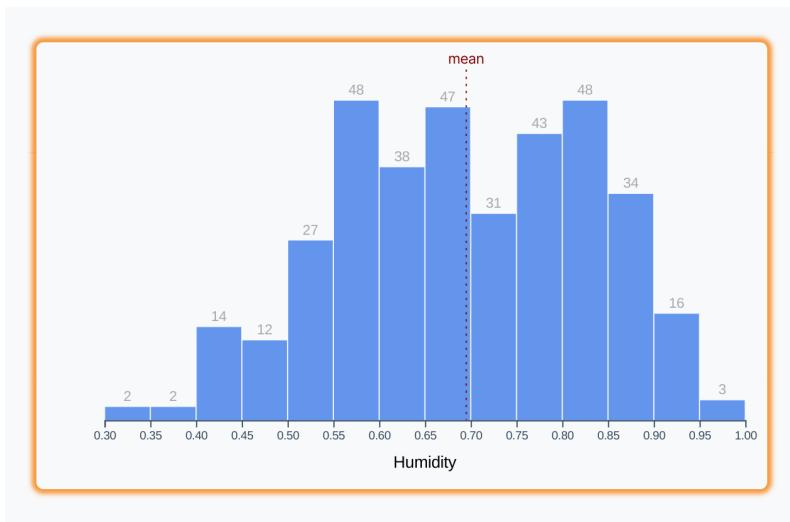
```
wrapper.attr("role", "figure")
    .attr("tabindex", "0")
```

When a user *tabs* to our chart, we want the screen reader to announce the basic layout so the user knows what they’re “looking” at. To do this, we can add a `<title>` SVG component with a short description.

```
wrapper.append("title")
    .text("Histogram looking at the distribution of humidity in 2016")
```

If you have a screen reader set up, you’ll notice that it will read our `<title>` when we tab to our chart. The “highlighted” state will look something like this:

²⁴https://developer.mozilla.org/en-US/docs/Web/Accessibility/ARIA/Roles/Figure_Role



Accessibility highlight

Next, we'll want to make our `binsGroup` selectable by also giving it a `tabindex` of `0`. If the user presses `tab` after the `wrapper` is focused, the browser will focus on the `binsGroup` because it's the next element (in DOM order) that is focusable.

```
const binsGroup = bounds.append("g")
  .attr("tabindex", "0")
```

We can also give our `binsGroup` a `role` of `"list"`, which will make the screen reader announce the number of items within the list. And we'll let the user know *what* the list contains by adding an `aria-label`.

```
const binsGroup = bounds.append("g")
  .attr("tabindex", "0")
  .attr("role", "list")
  .attr("aria-label", "histogram bars")
```

Now when our `binsGroup` is highlighted, the screen reader will announce: `"histogram bars. List with 15 items"`. Perfect!

Let's annotate each of our `"list items"`. After we create our `binGroups`, we'll add a few attributes to each group:

1. make it focusable with a `tabindex` of `0`
2. give it a `role` of `"listitem"`
3. give it an `aria-label` that the screen reader will announce when the item is *focused*.

```
const binGroups = binsGroup.selectAll("g")
  .data(bins)
  .enter().append("g")
    .attr("tabindex", "0")
    .attr("role", "listitem")
    .attr("aria-label", d => `There were ${yAccessor(d)} days between ${d.x0.toString().slice(0, 4)} and ${d.x1.toString().slice(0, 4)} humidity levels.`)
```

Now when we *tab* out of our `binsGroup`, it will focus the first bar group (and subsequent ones when we *tab*) and announce our `aria-label`.

We'll tackle one last issue — you might have noticed that the screen reader reads each of our x-axis tick labels once it's done reading our `<title>`. This is pretty annoying, and not giving the user much information. Let's prevent that.

At the bottom of our `drawBars()` function, let's select all of the text within our chart and give it an `aria-hidden` attribute of `"true"`.

```
wrapper.selectAll("text")
  .attr("role", "presentation")
  .attr("aria-hidden", "true")
```

Great! Now our screen reader will read only our labels and ignore any `<text>` elements within our chart.

With just a little effort, we've made our chart accessible to any users who access the web through a screen reader. That's wonderful, and more than most online charts can say!

Next up, we'll get fancy with animations and transitions.

Animations and Transitions

When we update our charts, we can animate elements from their old to their new positions. These animations can be visually exciting, but more importantly, they have functional benefits. When a bar animates from one height to another, the viewer has a better idea of whether it's getting larger or smaller. If we're animating several bars at once, we're drawing the viewer's eye to the bar making the biggest change because of its fast speed.

By analogy, imagine if track runners teleported from the start to the finish line. For one, it would be terribly boring to watch, but it would also be hard to tell who was fastest.

There are multiple ways we can animate changes in our graphs:

- using SVG `<animate>`
- using CSS `transition`
- using `d3.transition()`

Let's introduce each of these options.

SVG `<animate>`

`<animate>` is a native SVG element that can be defined inside of the animated element.

```
<svg width="120" height="120">
  <rect x="10" y="10" width="100" height="100" fill="cornflowerblue">
    <animate
      attributeName="x"
      values="0;20;0"
      dur="2s"
      repeatCount="indefinite"
    />
    <animate
      attributeName="fill"
      values="cornflowerBlue;maroon;cornflowerBlue"
      dur="6s"
      repeatCount="indefinite"
    />
  </rect>
</svg>
```

With your server running (`live-server`), navigate to the `4-animations-and-transitions/1-svg-animate/` folder in your browser to see this animation in action.



SVG animate

Unfortunately this won't work for our charts. For one, `<animate>` is **unsupported in Internet Explorer**. But the bigger issue is that we would have to set a static start and end value. We don't want to define static animations, instead we want our elements to animate changes between two dynamic values. Luckily, we have other options.

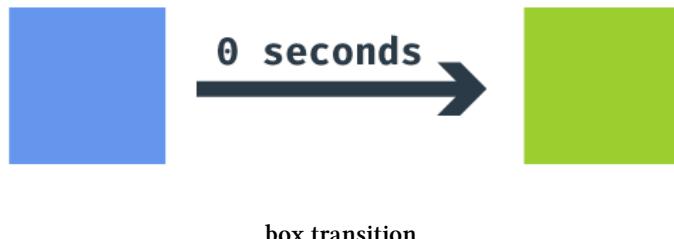
CSS transitions

Many of our chart changes can be transitioned with the CSS `transition` property. When we update a `<rect>` to have a fill of **red** instead of **blue**, we can specify that the color change take 10 seconds instead of being instantaneous. During those 10 seconds, the `<rect>` will continuously re-draw with intermediate colors on the way to **red**.

Not all properties can be animated. For example, how would you animate changing a label from **Humidity** to **Dew point**? However most properties can be animated, so feel free to operate under the assumption that a property can be animated until proven otherwise.

Let's try out an example! Navigate to the

`4-animations-and-transitions/2-css-transition-playground/` folder in the browser. You'll see a blue box that moves and turns green on hover.



Let's open up the

`4-animations-and-transitions/2-css-transition-playground/styles.css` file to take a look at what's going on. We can see our basic styles for the box.

```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
}
```

And our styles that apply to our box when it is hovered (change the background color and move it 30 pixels to the right).

```
.box:hover {  
  background: yellowgreen;  
  transform: translateX(30px);  
}
```

To create CSS a transition, we need to specify how long we want the animation to take with the `transition-duration` property. The property value accepts **time CSS data types** — a number followed by either `s` (seconds) or `ms` (milliseconds).

Let's make our box changes animate over one second.

```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
  transition-duration: 1s;  
}
```

When we refresh our webpage and hover over the box, we can see it slowly move to the right and turn green. Smooth!

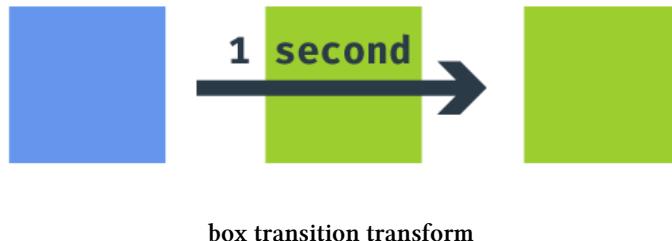


box transition all

Now let's say that we only want to animate our box's movement, but we want the color change to happen instantaneously. This is possible by specifying the `transition-property` CSS property. By default, `transition-property` is set to `all`, which animates all transitions. Instead, let's override the default and specify a specific CSS property name (`transform`).

```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
  transition-duration: 1s;  
  transition-property: transform;  
}
```

Now our box instantly turns green, but still animates to the right.



box transition transform

Instead of setting `transition-duration` and `transition-property` separately, we can use the shorthand property: `transition`. Shorthand CSS properties let you set multiple properties in one line. When we give `transition` a CSS property name and duration (separated by a space), we are setting both `transition-duration` and `transition-property`. Let's try it out.

```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
  transition: transform 1s;  
}
```

transition will accept a third property (transition-timing-function) that sets the acceleration curve for the animation. The animation could be linear (the default), slow then fast (ease-in), in steps (steps(6)), or even a custom function (cubic-bezier(0.1, 0.7, 1.0, 0.1)), among other options. Let's see what steps(6) looks like — it should break the animation into 6 discrete steps.

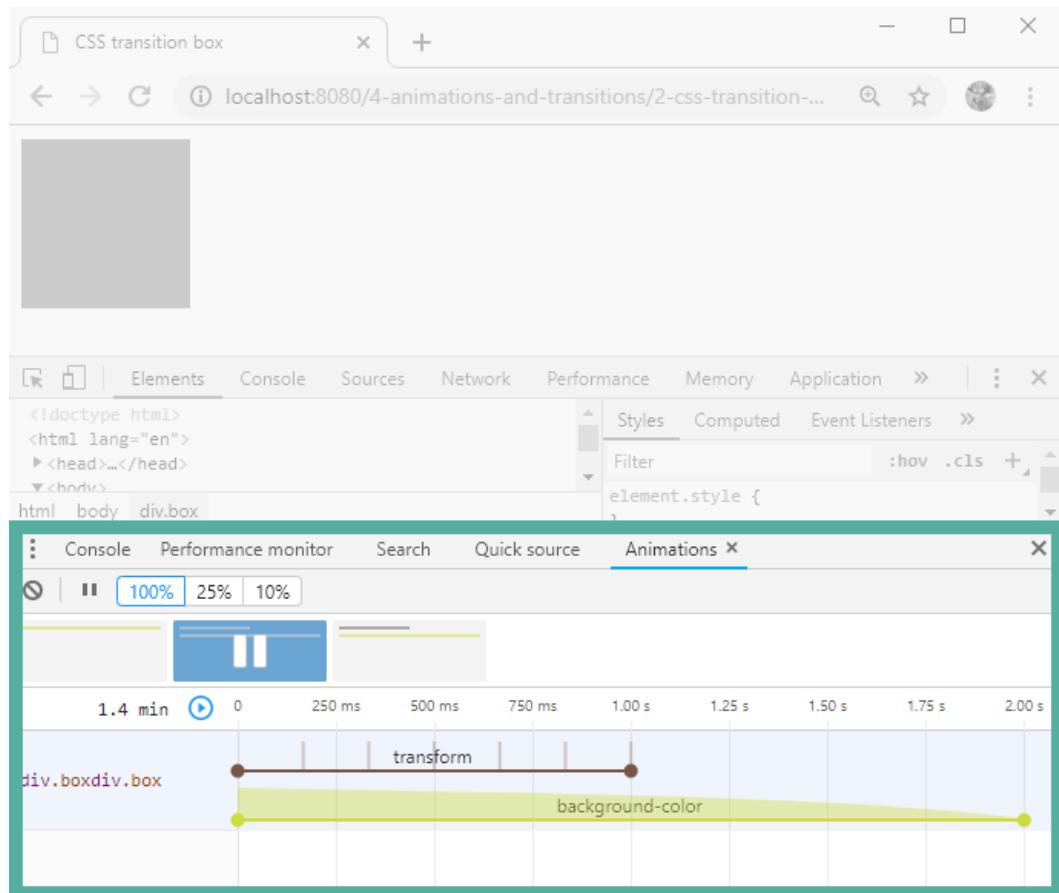
```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
  transition: transform 1s steps(6);  
}
```

What if we wanted to animate the color change, but finish turning green *while* our box is shifting to the right? transition will accept multiple transition statements, we just need to separate them by a comma. Let's add a transition for the background color.

```
.box {  
  background: cornflowerblue;  
  height: 100px;  
  width: 100px;  
  transition: transform 1s steps(6),  
             background 2s ease-out;  
}
```

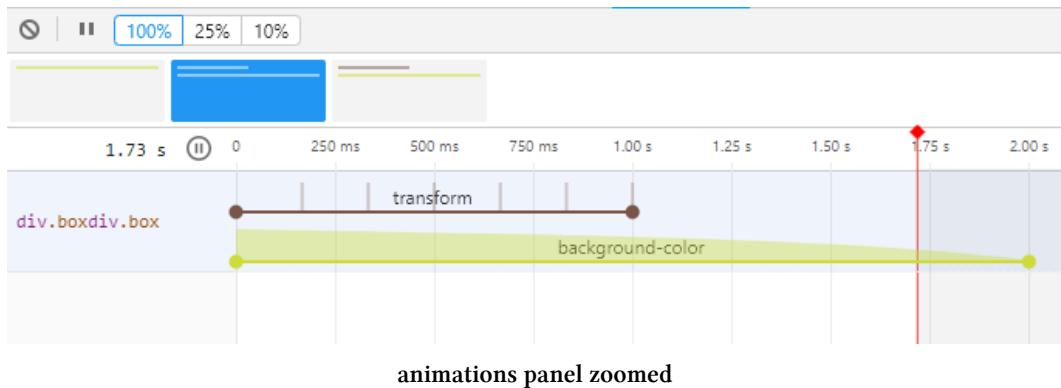
Nice! Now our box transitions by stepping to the right, while turning green over two seconds. Chrome's dev tools have a great way to visualize this transition. Press `esc` when looking at the **Elements** panel to bring up the bottom panel. In the bottom panel, we can open up the **Animations** tab.

If you don't see the **Animations** tab, click on the kebab menu on the left and select it from the dropdown options.



animations panel

Once we've triggered our box transition by hovering, we can inspect the animation.



We can see the `transform` transition on top, with six discrete changes, and the background animation on the bottom, easing gradually from one color to the next. The background transition diagram is twice as wide as the `transform` transition diagram, indicating that it takes twice as long.

This view can be very handy when inspecting, tweaking, and debugging transitions. Now that we're comfortable with CSS transition, let's see how we might use it to animate our charts.

Using CSS transition with a chart

Navigate to the `4-animated-and-transitions/3-draw-bars-with-css-transition` folder. The `index.html` is importing a CSS stylesheet (`styles.css`) and the `updating-bars.js` file, which is an updated version of our histogram drawing code from [Chapter 3](#).

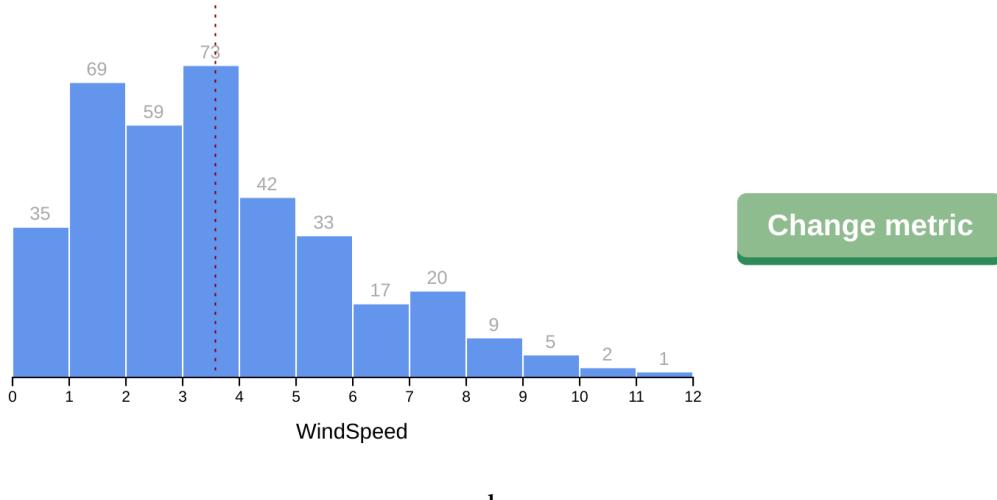
The code should look mostly familiar, but you might notice a few changes. Don't worry about those changes at the moment — they're not important to our main mission.

Let's look inside that `styles.css` file. We can see that we have already set the basic styles for our bars (`.bin rect`), bar labels (`.bin text`), mean line (`.mean`), and x axis label (`.x-axis-label`).

Great! Now that we know the lay of the land, let's point our browser at <http://localhost:8080/04-animated-and-transitions/3-draw-bars-with-css-transition>²⁵ — we should see our

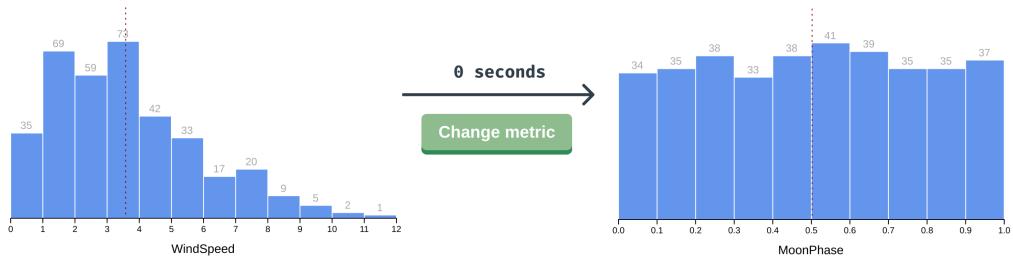
²⁵<http://localhost:8080/04-animated-and-transitions/3-draw-bars-with-css-transition/>

histogram and a **Change metric** button.



bars

When we click the button, our chart re-draws with the next metric, but the change is instantaneous.



instant change

Does this next metric have fewer bars than the previous one? Does our mean line move to the left or the right? These questions can be answered more easily if we transition gradually from one view to the other.

Let's add an animation whenever our bars update (`.bin_rect`).

```
.bin rect {  
    fill: cornflowerblue;  
    transition: all 1s ease-out;  
}
```

Note that a CSS transition won't work here in Firefox. This is because Firefox uses SVG 1.1, which sets `height` as an attribute. Chrome has implemented the part of the [SVG 2 spec^a](#) which allows `height` as a CSS property, letting us animate the transition.

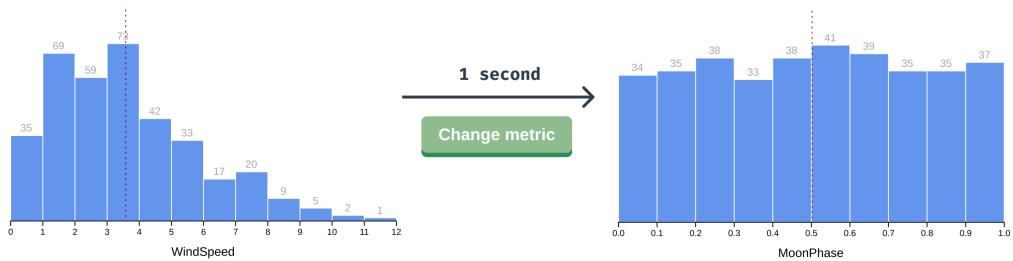
^a<https://www.w3.org/TR/SVG/styling.html#StylingUsingCSS>

Now when we update our metric, our bars shift slowly to one side while changing height — we can see that their `width`, `height`, `x`, and `y` values are animating. This may be fun to watch, but it doesn't really represent our mental model of bar charts. It would make more sense for the bars to change position instantaneously and animate any height differences. Let's only transition the `height` and `y` values.

```
.bin rect {  
    fill: cornflowerblue;  
    transition: height 1s ease-out,  
                y 1s ease-out;  
}
```

`ease-out` is a good starting point for CSS transitions — it starts quickly and slows down near the end of the animation to ease into the final value. It won't be ideal in every case, but it's generally a good choice.

That's better! Now we can see whether each bar is increasing or decreasing.



bars - slow change

Our transitions are still looking a bit disjointed with our text changing position instantaneously. Let's try to animate our text position, too.

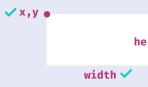
```
.bin text {
  transition: y 1s ease-out;
}
```

Hmm, our text position is still not animating — it seems as if `y` isn't a transition-able property. Thankfully there is a workaround here — we can position the text using a CSS property instead of changing its `y` attribute.

Our advanced package has a handy SVG element cheat sheet with green check marks to show us what SVG elements' attributes are animate-able with CSS transitions. Keep it around for a quick reference when you're transitioning your own elements!

SVG Elements

✓ can use CSS transitions

svg	SVG is a language for drawing graphics declaratively. Here are the main elements that can be used to draw graphics on a web page.	linearGradient, radialGradient
The Grand Poobah.	Store in defs. <code>reference: url(#id)</code>	Used to define a gradient, using:
Use this to surround all other SVG elements or to create a new coordinate system.		stop
defs	The definitions element.	<code>offset</code>
	Use this to store elements to be used elsewhere. For example:	<code>stop-color</code>
		<code>stop-opacity</code>
clipPath	Store in defs. <code>reference: url(#id)</code>	
	Used to clip other elements outside of its children elements' shape.	
rect		circle
<code>x, y</code>		<code>cx, cy</code>
<code>width</code>		<code>r</code>
line		g
<code>x1, y1</code>		A container to group other SVG elements.
<code>x2, y2</code>		Similar to an HTML <div>.
polygon		text
<code>points</code>		The only way to create text within SVG.
path		<code>x, y</code>
<code>d</code>		<code>dx</code>
		<code>dy</code>
		<code>rotate</code>
		<code>textLength</code>
		<code>lengthAdjust</code>
ellipse		
<code>cx, cy</code>		
<code>rx</code>		
<code>ry</code>		

SVG elements cheat sheet

Switching over to our `updating-bars.js` file, let's position our bar labels using `translateY()`.

```
const barText = binGroups.select("text")
  .attr("x", d => xScale(d.x0) + (xScale(d.x1) - xScale(d.x0)) / 2)
  .attr("y", 0)
  .style("transform", d => `translateY(${yScale(yAccessor(d)) - 5}px)`)
  .text(d => yAccessor(d) || "")
```

Note that we're filling our `<text>` elements with empty strings instead of `0` (with `.text(d => yAccessor(d) || "")`) to prevent labeling empty bars.

We'll also need to change the transition property to target `transform`.

```
.bin text {  
  transition: transform 1s ease-out;  
}
```

Now our bar labels are animating with our bars. Perfect!

Let's make one last change - we want our dashed mean line to animate when it moves left or right. We could try to `transition` changes to `x1` and `x2`, but those aren't CSS properties, they're SVG attributes. Let's position the line's horizontal position with the `transform` property.

```
const meanLine = bounds.selectAll(".mean")  
  .attr("y1", -20)  
  .attr("y2", dimensions.boundedHeight)  
  .style("transform", `translateX(${xScale(mean)}px)`)
```

We'll also add the `transition` CSS property in our `styles.css` file:

```
.mean {  
  stroke: maroon;  
  stroke-dasharray: 2px 4px;  
  transition: transform 1s ease-out;  
}
```

These updates are looking great!

There are some animations that aren't possible with CSS transitions. For example, transitioning the x axis changes would help us see if the values for our new metric have increased or decreased. Using a CSS transition won't help here — CSS has no way of knowing whether a tick mark with the text `10` is larger than a tick mark with the text `100`. Let's bring out the heavier cavalry.

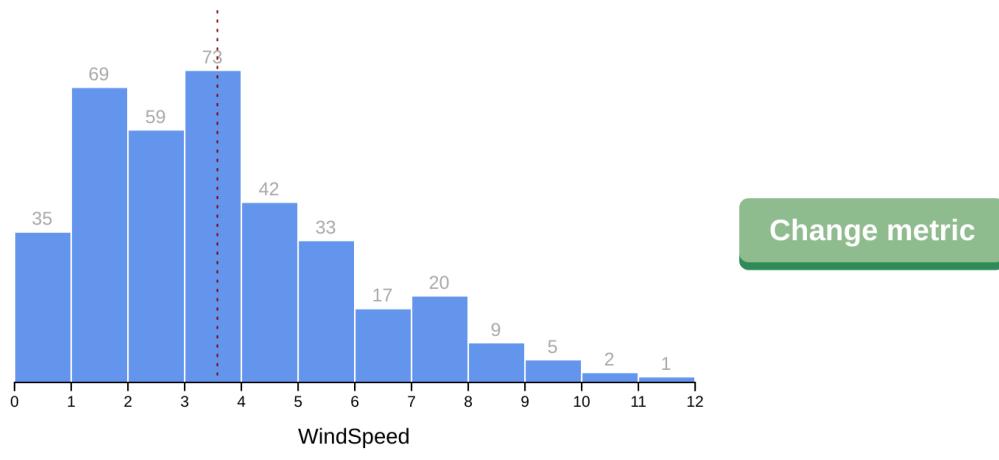
d3.transition

CSS transitions have our back for simple property changes, but for more complex animations we'll need to use `d3.transition()` from the `d3-transition`²⁶ module. When would we want to use `d3.transition()` instead of CSS transitions?

²⁶<https://github.com/d3/d3-transition>

- When we want to ensure that multiple animations line up
- When we want to do something when the animation ends (for example starting another animation)
- When the property we want to animate isn't a CSS property (remember when we tried to animate our bars' heights but had to use `transform` instead? `d3.translate` can animate non-CSS property changes.)
- When we want to synchronize adding and removing elements with animations
- When we might interrupt halfway through a transition
- When we want a custom animation (for example, we could write a custom interpolator for changing text that adds new letters one-by-one)

Let's get our hands dirty by re-implementing the CSS transitions for our histogram. Navigate to the `/04-animations-and-transitions/3-draw-bars-with-d3-transition/` folder — you'll see the same setup with our histogram and a big old **Change metric** button.



Let's again start by animating any changes to our bars. Instead of adding a `transition` property to our `styles.css` file, we'll start in the `updating-bars.js` file where we set our `barRects` attributes.

As a reminder, when we run:

```
const barRects = binGroups.select("rect")
```

we're creating a d3 selection object that contains all `<rect>` elements. Let's log that to the console as a refresher of what a selection object looks like.

```
const barRects = binGroups.select("rect")
  .attr("x", d => xScale(d.x0) + barPadding)
  .attr("y", d => yScale(yAccessor(d)))
  .attr("height", d => dimensions.boundedHeight
    - yScale(yAccessor(d)))
)
.attr("width", d => d3.max([
  0,
  xScale(d.x1) - xScale(d.x0) - barPadding
]))
```



```
console.log(barRects)
```

```
updating-bars.js:88
▼ Selection {_groups: Array(1), _parents: Array(1)} ⓘ
  ▼ _groups: Array(1)
    ► 0: (18) [rect, rect, ...
      length: 1
    ► __proto__: Array(0)
  ▼ _parents: [g]
  ► __proto__: Object
```

bars selection

We can use the `.transition()` method on our d3 selection object to transform our selection object into a d3 transition object.

```
const barRects = binGroups.select("rect")
  .transition()
    .attr("x", d => xScale(d.x0) + barPadding)
    .attr("y", d => yScale(yAccessor(d)))
    .attr("height", d => dimensions.boundedHeight
      - yScale(yAccessor(d)))
  )
  .attr("width", d => d3.max([
    0,
    xScale(d.x1) - xScale(d.x0) - barPadding
  ]))

console.log(barRects)
```

```
updating-bars.js:85
▼ Transition {_groups: Array(1), _parents: Array(1), _name: null, _id: 1} ⓘ
  ▼ _groups: Array(1)
    ► 0: (18) [rect, rect, ...
      length: 1
    ► __proto__: Array(0)
    _id: 1
    _name: null
  ► _parents: [g]
  ► __proto__: Object
```

bars transition

d3 transition objects look a lot like selection objects, with a `_groups` list of relevant DOM elements and a `_parents` list of ancestor elements. They have two additional keys: `_id` and `_name`, but that's not all that has changed.

Let's expand the `__proto__` of our transition object.

`__proto__` is a native property of JavaScript objects that exposes methods and values that this specific object has inherited. If you're unfamiliar with JavaScript Prototypal Inheritance and want to read up, the [MDN docs^a](https://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain) are a good place to start.

^ahttps://developer.mozilla.org/en-US/docs/Web/JavaScript/Inheritance_and_the_prototype_chain

In this case, we can see that the `__proto__` property contains d3-specific methods, and the nested `__proto__` object contains native object methods, such as `toString()`.

```

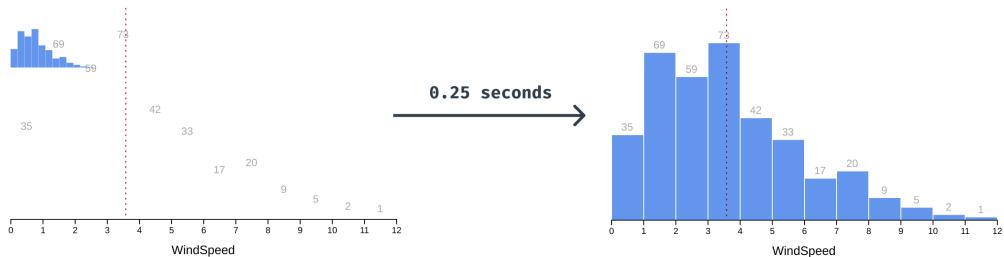
updating-bars.js:85
▼ Transition {_groups: Array(1), _parents: Array(1), _name: null, _id: 1} ⓘ
  ► _groups: [Array(18)]
    _id: 1
    _name: null
  ► _parents: [g]
  ▼ __proto__:
    ► attr: f transition_attr(name, value)
    ► attrTween: f transition_attrTween(name, value)
    ► call: f selection_call()
    ► constructor: f Transition(groups, parents, name, id)
    ► delay: f transition_delay(value)
    ► duration: f transition_duration(value)
    ► each: f selection_each(callback)
    ► ease: f transition_ease(value)
    ► empty: f selection_empty()
    ► filter: f transition_filter(match)
    ► merge: f transition_merge(transition$$1)
    ► node: f selection_node()
    ► nodes: f selection_nodes()
    ► on: f transition_on(name, listener)
    ► remove: f transition_remove()
    ► select: f transition_select(select$$1)
    ► selectAll: f transition_selectAll(select$$1)
    ► selection: f transition_selection()
    ► size: f selection_size()
    ► style: f transition_style(name, value, priority)
    ► styleTween: f transition_styleTween(name, value, priority)
    ► text: f transition_text(value)
    ► transition: f transition_transition()
    ► tween: f transition_tween(name, value)
  ► __proto__: Object

```

Transition object, expanded

We can see that some methods are inherited from d3 selection objects (eg. `.call()` and `.each()`), but most are overwritten by new transition methods. When we click the **Change metric** button now, we can see that our bar changes are animated. This makes sense — any `.attr()` updates chained after a `.transition()` call will use transition's `.attr()` method, which attempts to interpolate between old and new values.

Something looks strange though - our new bars are flying in from the top left corner.



Note that d3 transitions animate over 0.25 seconds — we'll learn how to change that in a minute!

Knowing that `<rect>`s are drawn in the top left corner by default, this makes sense. But how do we prevent this?

Remember how we can isolate new data points with `.enter()`? Let's find the line where we're adding new `<rect>`s and set their initial values. We want them to start in the right horizontal location, but be 0 pixels tall so we can animate them “growing” from the x axis.

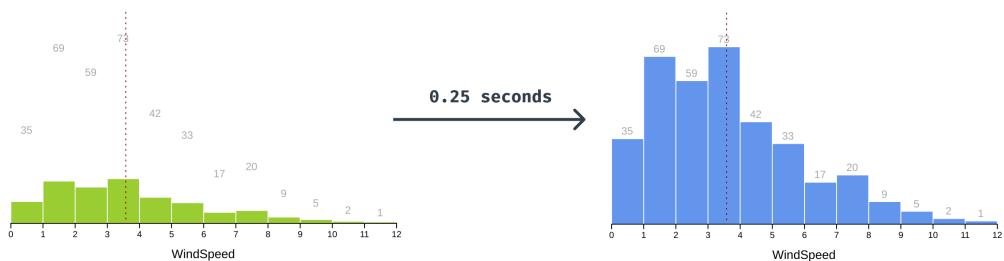
Let's also have them be green to start to make it clear which bars we're targeting. We'll need to set the fill using an inline style using `.style()` instead of setting the attribute in order to override the CSS styles in `styles.css`.

```
newBinGroups.append("rect")
  .attr("height", 0)
  .attr("x", d => xScale(d.x0) + barPadding)
  .attr("y", dimensions.boundedHeight)
  .attr("width", d => d3.max([
    0,
    xScale(d.x1) - xScale(d.x0) - barPadding
  ]))
  .style("fill", "yellowgreen")
```

Why are we using `.style()` instead of `.attr()` to set the fill? We need the `fill` value to be an inline style instead of an SVG attribute in order to override the CSS styles in `styles.css`. The way CSS specificity works means that inline styles override class selector styles, which override SVG attribute styles.

Once our bars are animated in, they won't be new anymore. Let's transition their fill to blue. Luckily, chaining d3 transitions is really simple — to add a new transition that starts after the first one ends, add another `.transition()` call.

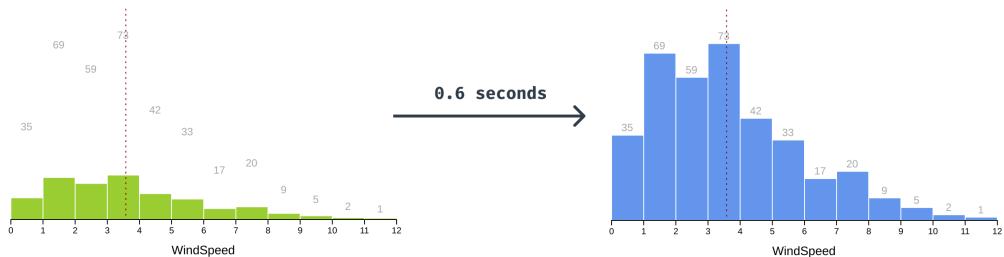
```
const barRects = binGroups.select("rect")
  .transition()
    .attr("x", d => xScale(d.x0) + barPadding)
    .attr("y", d => yScale(yAccessor(d)))
    .attr("height", d => dimensions.boundedHeight
      - yScale(yAccessor(d)))
  )
  .attr("width", d => d3.max([
    0,
    xScale(d.x1) - xScale(d.x0) - barPadding
  ]))
  .transition()
    .style("fill", "cornflowerblue")
```



green bars on load

Let's slow things down a bit so we can bask in these fun animations. d3 transitions default to 0.25 seconds, but we can specify how long an animation takes by chaining `.duration()` with a number of milliseconds.

```
const barRects = binGroups.select("rect")
  .transition().duration(600)
    .attr("x", d => xScale(d.x0) + barPadding)
  // ...
```



green bars on load, 1 second

Smooth! Now that our bars are nicely animated, it's jarring when our text moves to its new position instantly. Let's add another transition to make our text transition with our bars.

```
const barText = binGroups.select("text")
  .transition().duration(600)
    .attr("x", d => xScale(d.x0)
      + (xScale(d.x1) - xScale(d.x0)) / 2
    )
    .attr("y", d => yScale(yAccessor(d)) - 5)
    .text(d => yAccessor(d) || "")
```

We'll also need to set our labels' initial position (higher up in our code) to prevent them from flying in from the left.

```
newBinGroups.append("text")
    .attr("x", d => xScale(d.x0)
        + (xScale(d.x1) - xScale(d.x0)) / 2
    )
    .attr("y", dimensions.boundedHeight)
```

Here's a fun tip: we can specify a `timing` function (similar to CSS's `transition-timing-function`) to give our animations some life. They can look super fancy, but we only need to chain `.ease()` with a d3 easing function. Check out the full list at [the d3-ease repo²⁷](#).



```
const barRects = binGroups.select("rect")
    .transition().duration(600).ease(d3.easeBounceOut)
    .attr("x", d => xScale(d.x0) + barPadding)
// ...
```

That's looking groovy, but our animation is out of sync with our labels again. We could ease our other transition, but there's an easier (no pun intended) way to sync multiple transitions.

By calling `d3.transition()`, we can make a transition on the root document that can be used in multiple places. Let's create a root transition — we'll need to place this definition above our existing transitions, for example after we define `barPadding`. Let's also log it to the console to take a closer look.

```
const barPadding = 1

const updateTransition = d3.transition()
    .duration(600)
    .ease(d3.easeBackIn)

console.log(updateTransition)
```

If we expand the `_groups` array, we can see that this transition is indeed targeting our root `<html>` element.

²⁷<https://github.com/d3/d3-ease>

```
updating-bars-with-d3-transition.js:70
▼ Transition {_groups: Array(1), _parents: Array(1), _name: null, _id: 1} ⓘ
  ▼ _groups: Array(1)
    ► 0: [html]
      length: 1
    ► __proto__: Array(0)
    _id: 1
    _name: null
  ► _parents: [null]
  ► __proto__: Object
Root transition object
```

You'll notice errors in the dev tools console that say `Error: <rect> attribute height: A negative value is not valid..` This happens with the `d3.easeBackIn` easing we're using, which causes the bars to bounce below the x axis when they animate.

Let's update our bar transition to use `updateTransition` instead of creating a new transition. We can do this by passing the existing transition in our `.transition()` call.

```
const barRects = binGroups.select("rect")
  .transition(updateTransition)
    .attr("x", d => xScale(d.x0) + barPadding)
  // ...
```

Let's use `updateTransition` for our text, too.

```
const barText = binGroups.select("text")
  .transition(updateTransition)
    .attr("x", d => xScale(d.x0)
      + (xScale(d.x1) - xScale(d.x0)) / 2
    )
  // ...
```

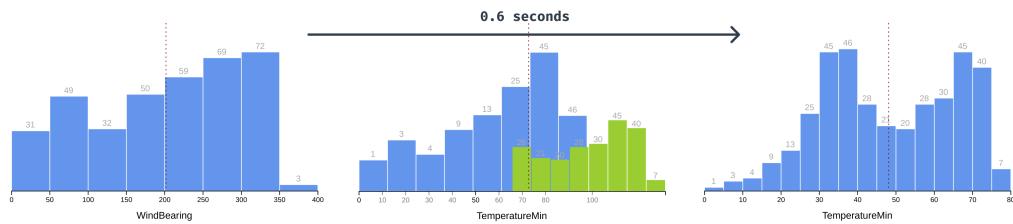
We can use this transition as many times as we want — let's also animate our mean line when it updates.

```
const meanLine = bounds.selectAll(".mean")
  .transition(updateTransition)
    .attr("x1", xScale(mean))
  // ...
```

And our x axis:

```
const xAxis = bounds.selectAll(".x-axis")
  .transition(updateTransition)
    .call(xAxisGenerator)
```

Remember that we couldn't animate our x axis with CSS transition? Our transition objects are built to handle axis updates — we can see our tick marks move to fit the new **domain** before the new tick marks are drawn.



Bars - transition all elements

But what about animating our bars when they leave? Good question - exit animations are often difficult to implement because they involve delaying element removal. Thankfully, d3 transition makes this pretty simple.

Let's start by creating a transition right before we create updateTransition. Let's also take out the easing we added to updateTransition since it's a little distracting.

```
const exitTransition = d3.transition().duration(600)
const updateTransition = d3.transition().duration(600)
```

We can target only the bars that are exiting using our `.exit()` method. Let's turn them red before they animate to make it clear which bars are leaving. Then we can use our `exitTransition` and animate the `y` and `height` values so the bars shrink into the x axis.

Don't look at the browser just yet, we'll need to finish our exit transition first.

```
const oldBinGroups = binGroups.exit()
oldBinGroups.selectAll("rect")
  .style("fill", "red")
  .transition(exitTransition)
    .attr("y", dimensions.boundedHeight)
    .attr("height", 0)
```

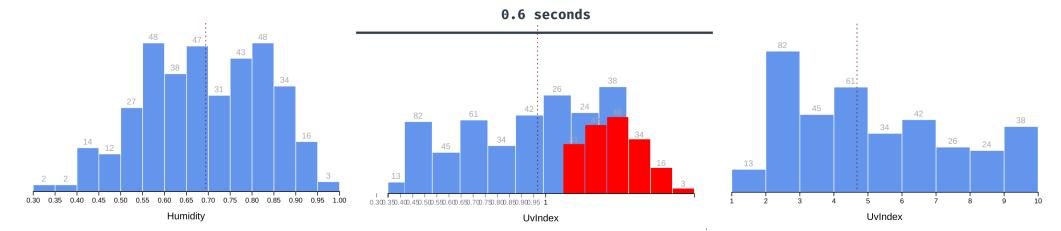
And we'll remember to also transition our text this time:

```
oldBinGroups.selectAll("text")
  .transition(exitTransition)
    .attr("y", dimensions.boundedHeight)
```

Last, we need to actually remove our bars from the DOM. We'll use a new transition here — not because we can animate removing the elements, but to delay their removal until the transition is over.

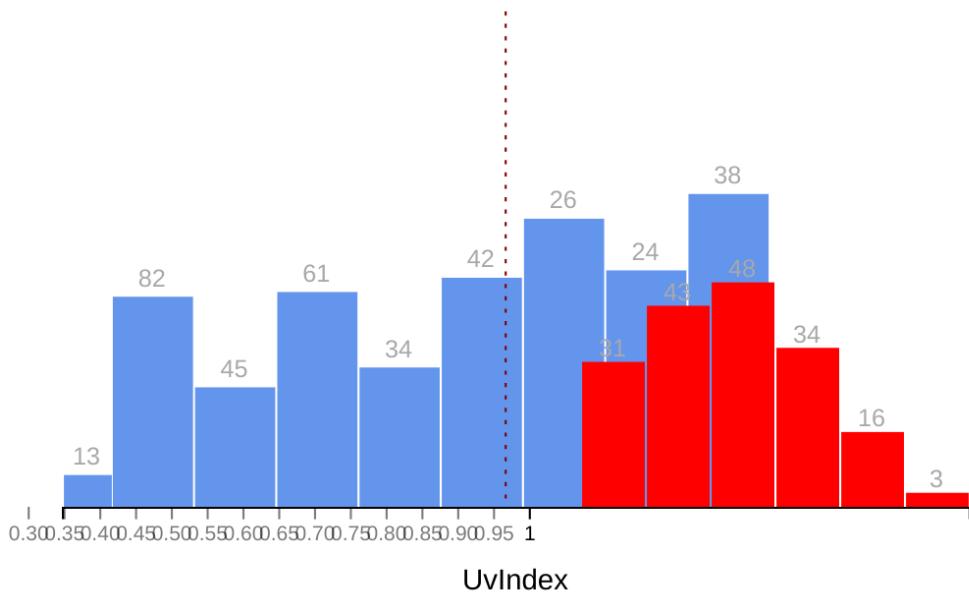
```
oldBinGroups
  .transition(exitTransition)
    .remove()
```

Now we can look at the browser, and we can see our bars animating in and out!



Bars - transition in and out

There is one issue, though: our bars are moving to their new positions while the bars are still exiting and we end up with intermediate states like this one:



Transition - intermediate

To fix this, we'll delay the update transition until the exit transition is finished. Instead of creating a our `updateTransition` as a root transition, we can chain it on our existing `exitTransition`.

```
const exitTransition = d3.transition().duration(600)
const updateTransition = exitTransition.transition().duration(600)
```

We're chaining transitions here to run them one after the other — d3 transitions also have a `.delay()` method if you need to delay a transition for a certain amount of time. Check out [the docs^a](#) for more information.

^ahttps://github.com/d3/d3-transition#transition_delay

Wonderful! Now that we've gone through the three different ways we can animate changes, let's recap when each method is appropriate.

SVG `<animate>` is only appropriate for static animations.

CSS `transition` is useful for animating CSS properties. A good rule of thumb is to use these mainly for stylistic polish — that way we can keep simpler transitions in our stylesheets, with the main goal of making our visualizations feel smoother.

`d3.transition()` is what we want to use for more complex animations: whenever we need to chain or synchronize with another transition or with DOM changes.

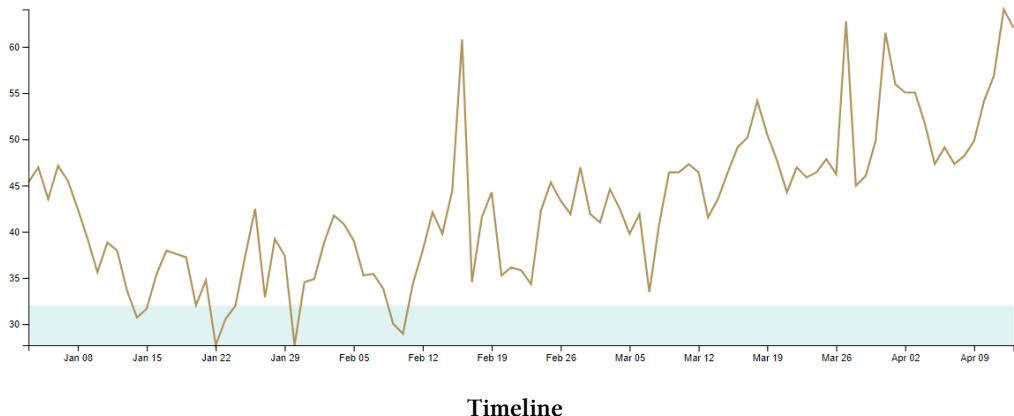
Lines

After animating bars, animating line transitions should be easy, right? Let's find out!

Let's navigate to the `/04-animations-and-transitions/4-draw-line/` folder and open the `updating-line.js` file.

Look familiar? This is our timeline drawing code from [Chapter 1](#) with some small updates.

You might need to update your “freezing” temperatures, if you live in a warm place and are getting errors in the console.



One of the main changes is an `addNewDay()` function at the bottom of the script. This exact code isn't important — what is good to know is that `addNewDay()` shifts our dataset one day in the future. To simulate a live timeline, `addNewDay()` runs every 1.5 seconds.

If you read through the `addNewDay()` code and were confused by the `...dataset.slice(1)`, syntax, the `...` is using ES6 spread syntax to expand the dataset (minus the first point) in place. Read more about it in [the MDN docs^a](#).

^ahttps://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Spread_syntax

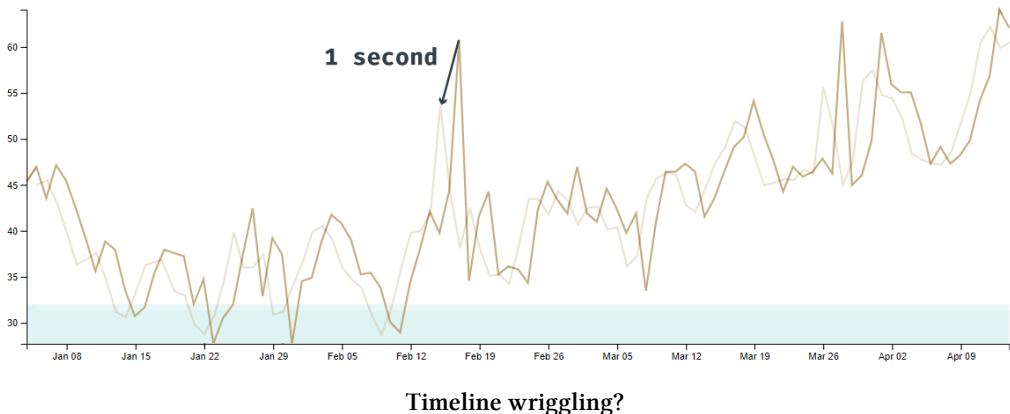
We can see our timeline updating when we load our webpage, but it looks jerky. We know how to smooth the axis transitions, let's make them nice and slow.

```
const xAxis = bounds.select(".x-axis")
  .transition().duration(1000)
  .call(xAxisGenerator)
```

Great! Now let's transition the line.

```
const line = bounds.select(".line")
  .transition().duration(1000)
    .attr("d", lineGenerator(dataset))
```

What's going on here? Why is our line wriggling around instead of adding a new point at the end?



Remember when we talked about how path **d** attributes are a string of draw-to values, like a learn-coding turtle? d3 is transitioning each point to **the next point at the same index**. Our transition's `.attr()` function has no idea that we've just shifted our points down one index. It's guessing how to transition to the new **d** value, animating each point to the next day's y value.

Pretend you're the `.attr()` function - how would you transition between these two **d** values?

```
<path d= "M 0 50 L 1 60 L 2 70 L 3 80 Z" />
<path d= "M 0 60 L 1 70 L 2 80 L 3 90 Z" />
```

It would make the most sense to transition each point individually, interpolating from **0 50** to **0 60** instead of moving each point to the left.

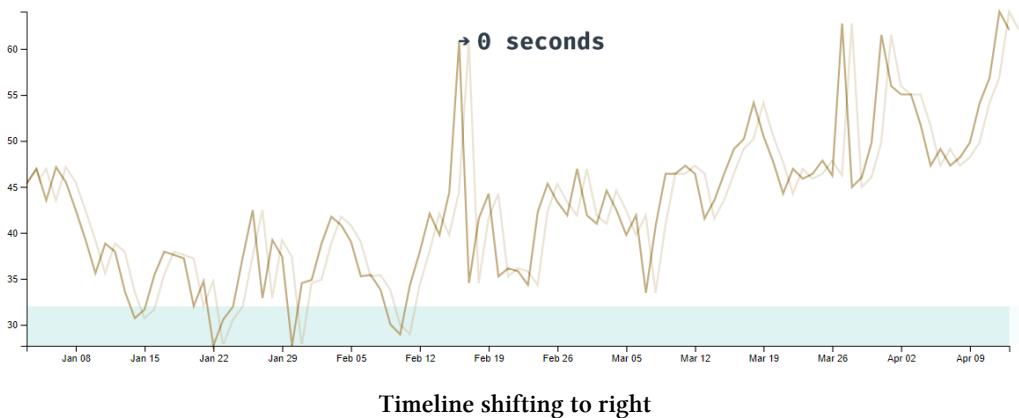
Great, we understand *why* our line is wriggling, but how do we shift it to the left instead?

Let's start by figuring out **how far we need to shift our line** to the left. Before we update our line, let's grab the last two points in our dataset and find the difference between their x values.

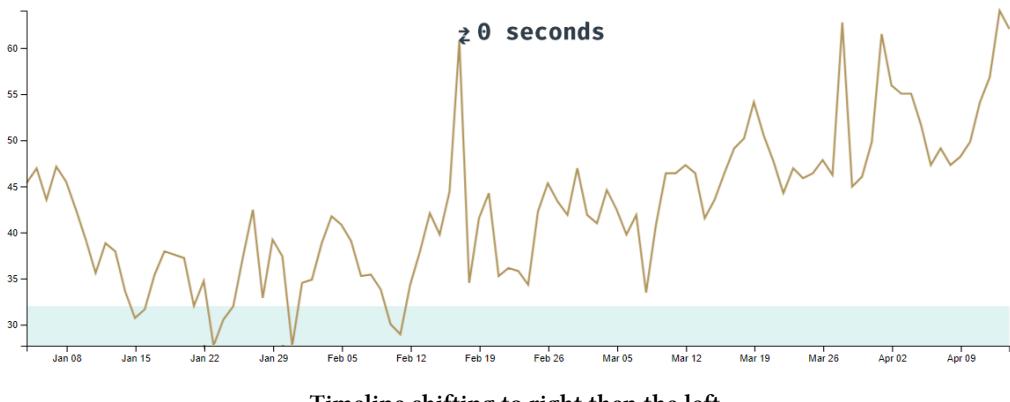
```
const lastTwoPoints = dataset.slice(-2)
const pixelsBetweenLastPoints = xScale(xAccessor(lastTwoPoints[1]))
  - xScale(xAccessor(lastTwoPoints[0]))
```

Now when we update our line, we can instantly shift it to the right to match the old line.

```
const line = bounds.select(".line")
  .attr("d", lineGenerator(dataset))
  .style("transform", `translateX(${pixelsBetweenLastPoints}px)`)
```

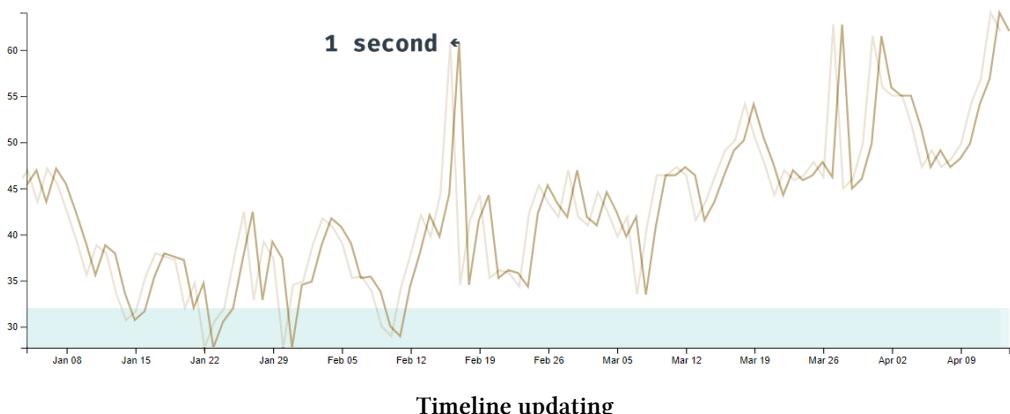


This shift should be invisible because **at the same time we're shifting our x scale to the left by the same amount**.



Then we can animate un-shifting the line to the left, to its normal position on the x axis.

```
const line = bounds.select(".line")
    .attr("d", lineGenerator(dataset))
    .style("transform", `translateX(${pixelsBetweenLastPoints}px}`)
    .transition().duration(1000)
    .style("transform", "none")
```



Okay great! We can see the line updating before it animates to the left, but we don't want to see the new point until it's within our bounds. **The easiest way to hide out-of-bounds data is to add a <clipPath>**.

A `<clipPath>` is an SVG element that:

- is sized by its children. If a `<clipPath>` contains a `circle`, it will only paint content within that circle's bounds.
- can be referenced by another SVG element, using the `<clipPath>`'s `id`.

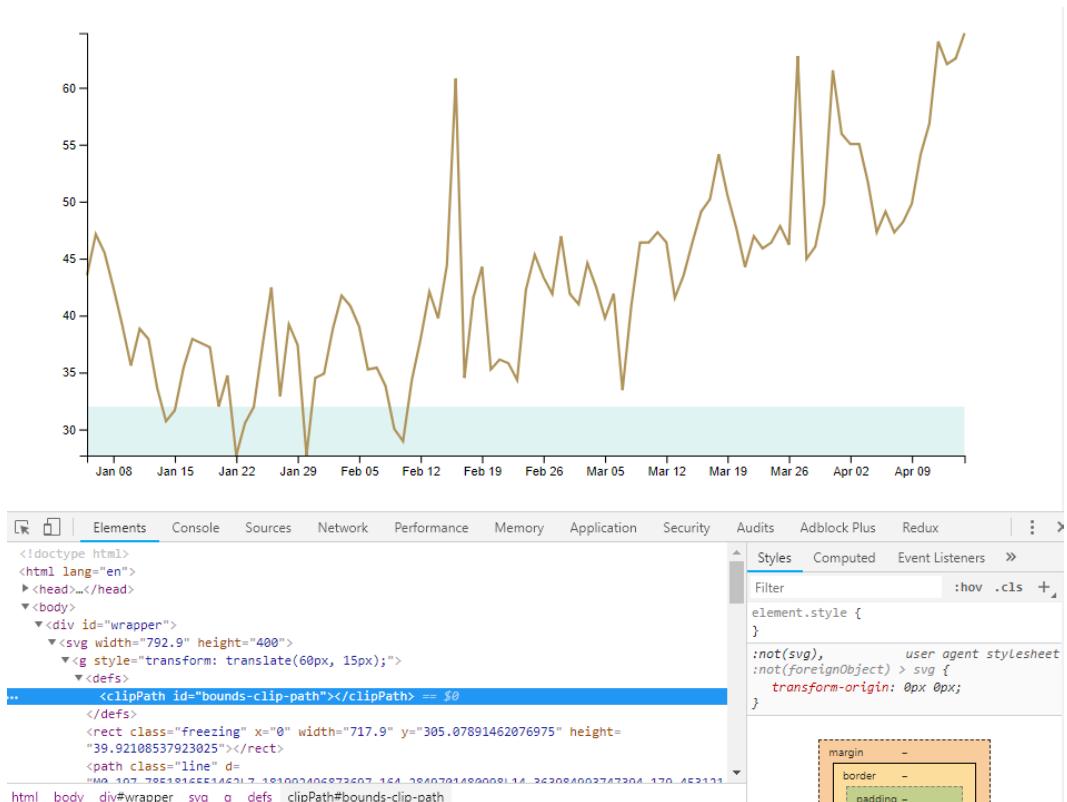
Before we test it out, we need to learn one important SVG convention: using `<defs>`. The SVG `<defs>` element is used to store any re-usable definitions that are used later in the `<svg>`. By placing any `<clipPath>`s or gradients in our `<defs>` element, we'll make our code more accessible. We'll also know where to look when we're debugging, similar to defining constants in one place before we use them.

Now that we know this convention, let's create our `<defs>` element and add our `<clipPath>` inside. We'll want to put this definition right after we define our `bounds`. Let's also give it an `id` that we can reference later.

```
const bounds = wrapper.append("g")
  .style("transform", `translate(${{
    dimensions.margin.left
  }px, ${{
    dimensions.margin.top
  }px})`)

bounds.append("defs")
  .append("clipPath")
    .attr("id", "bounds-clip-path")
```

If we inspect our `<clipPath>` in the Elements panel, we can see that it's not rendering at all.



clipPath

Remember, the `<clipPath>` element's shape depends on its children, and it has no children yet. Let's add a `<rect>` that covers our bounds.

```
bounds.append("defs")
.append("clipPath")
  .attr("id", "bounds-clip-path")
.append("rect")
  .attr("width", dimensions.boundedWidth)
  .attr("height", dimensions.boundedHeight)
```

To use our `<clipPath>` we'll create a group with the attribute `clip-path` pointing to our `<clipPath>`'s id. The order in which we draw SVG elements determines their "z-index". Keeping that in mind, let's add our new group *after* we draw the `freezing` `<rect>`.

```

bounds.append("rect")
    .attr("class", "freezing")
const clip = bounds.append("g")
    .attr("clip-path", "url(#bounds-clip-path)")

```

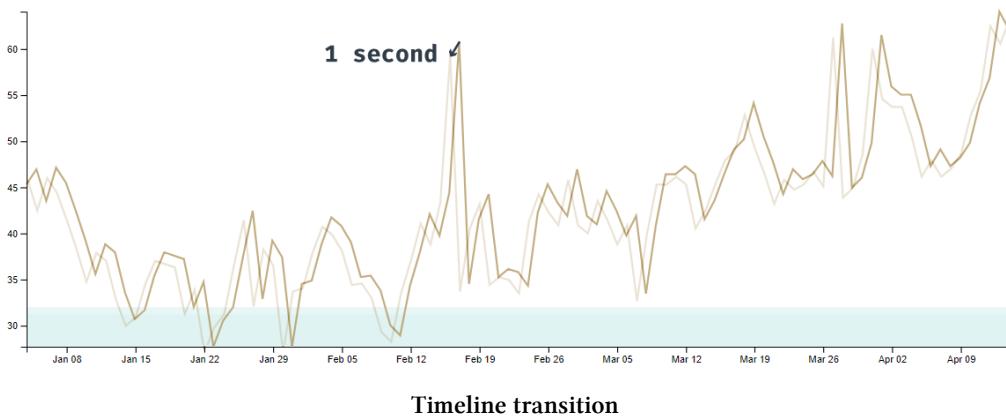
Now we can update our path to sit inside of our new group, instead of the bounds.

```

clip.append("path")
    .attr("class", "line")

```

Voila! When we reload our webpage, we can see that our line's new point isn't fully visible until it has finished un-shifting.



We can see that the first point of our dataset is being removed before our line unshifts. I bet you could think of a few ways around this — feel free to implement one or two! We could save the old dataset and preserve that extra point until our line is unshifted, or we could slice off the first data point when we define our x scale. In a production graph, the solution would depend on how our data is updating and what's appropriate to show.

Now that we have the tools needed to make our chart transitions lively, we'll learn how to let our users interact with our charts!

Interactions

The biggest advantage of creating charts with JavaScript is the ability to respond to user input. In this chapter, we'll learn what ways users can interact with our graphs and how to implement them.

d3 events

Browsers have native **event listeners** — using `addEventListener()`, we can listen for events from a user's:

- mouse
- keyboard
- scroll wheel
- touch
- resize
- ... and more.

For example:

```
function onClick(event) {  
    // do something here...  
}  
addEventListener("click", onClick)
```

After running this code, the browser will trigger `onClick()` when a user clicks anywhere on the page.

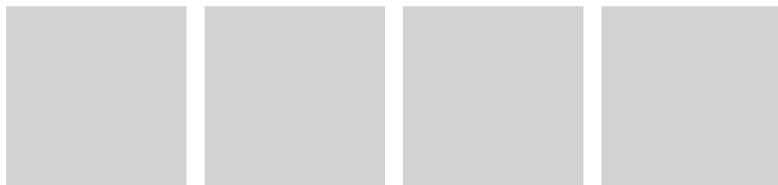
These event listeners have tons of functionality and are simple to use. We can get even more functionality using d3's event listener wrappers!

Our d3 selection objects have an `.on()` method that will create event listeners on our selected DOM elements. Let's take a look at how to implement d3 event listeners. First, we'll need to get our server up and running `live-server` and navigate to `/code/05-interactions/1-events/draft/`.

If we open the `events.js` file, we can see a few things happening:

1. We define `rectColors` as an array of colors.
2. We grab all `.rect` elements inside of the `#svg` element (created in `index.html`) and bind our selection to the `rectColors` array.
3. We use `.enter()` to isolate all new data points (every row in `rectColors`) and `.append()` a `<rect>` for each row.
4. Lastly, we set each `<rect>`'s size to 100 pixels by 100 pixels and shift each item 110 pixels to the right (multiplied by its index). We also make all of our boxes light grey.

In our browser, we can see our four boxes.



grey boxes

They don't do much right now, let's make it so they change to their designated color on hover.

To add a d3 event listener, we pass the type of event we want to listen for as the first parameter of `.on()`. Any DOM event type will work — see the full list of event types on [the MDN docs²⁸](#). To mimic a hover start, we'll want to target `mouseenter`.

²⁸https://developer.mozilla.org/en-US/docs/Web/Events#Standard_events

```
rect.on("mouseenter")
```

The second parameter `.on()` receives is a callback function that will be executed when the specified event happens. This function will receive three parameters:

1. the *bound datum*
2. the current *index*
3. the *currently selected nodes*

Let's log these parameters to the console to get a better look.

```
rects.on("mouseenter", function(datum, index, nodes) {  
  console.log({datum, index, nodes})  
})
```



It can often be helpful to use ES6 object property shorthand for logging multiple variables. This way, we can see the name and value of each variable!

When we hover over a box, we can see that our `mouseenter` event is triggered! The parameters passed to our function(in order) are:

1. the matching *data point* bound from the `rectColors` array (in this case, the color)
2. the rect's *index*
3. the *nodes* in the current selection (in this case, the list of `<rect>`s)

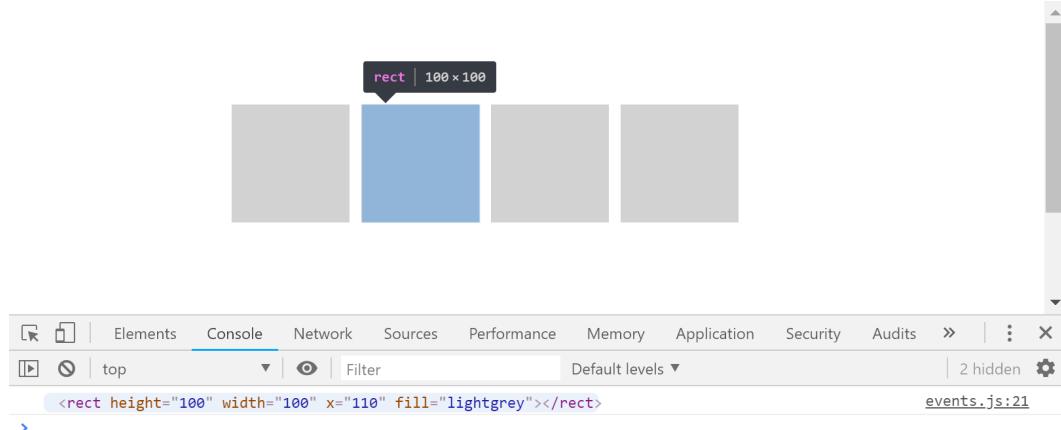
```
▼ {datum: "cornflowerblue", index: 1, nodes: Array(4)} ⓘ events.js:21  
  datum: "cornflowerblue"  
  index: 1  
 ▶ nodes: (4) [rect, rect, rect, rect]  
 ▶ __proto__: Object
```

function parameters

In order to change the color of the current box, we'll need to create a d3 selection targeting only that box. We *could* find it in the list of nodes using its index, but there's an easier way. Let's take a look at what `this` looks like in our function.

```
rects.on("mouseenter", function(datum, index, nodes) {  
  console.log(this)  
})
```

Perfect! It looks like the `this` keyword points at the DOM element that triggered the event.



function this

We can use `this` to create a d3 selection and set the box's fill using the datum.

```
rects.on("mouseenter", function(datum, index, nodes) {  
  d3.select(this).style("fill", datum)  
})
```

Now when we refresh our webpage, we can change our boxes to their related color on hover!



hovered boxes

Hmm, we're missing something. We want our boxes to turn back to grey when our mouse leaves them. Let's chain another event listener that triggers on `mouseout` and make our box grey again.

```
rects.on("mouseenter", function(datum, index, nodes) {  
  d3.select(this).style("fill", datum)  
})  
.on("mouseout", function() {  
  d3.select(this).style("fill", "lightgrey")  
})
```



`mouseenter` is often seen as interchangeable with `mouseover`. The two events are very similar, but `mouseenter` is usually closer to the wanted behavior. They are both triggered when the mouse enters the targeted container, but `mouseover` is also triggered when the mouse moves between nested elements.

This same distinction applies to `mouseleave` (preferred) and `mouseout`.

Don't use fat arrow functions

You might be wondering why we're not using ES6 fat arrow functions here. Let's look at what happens to our `this` keyword when we use an arrow function:

```
rects.on("mouseenter", () => {
  console.log(this)
})
```

Oh right, `this` in arrow functions refer to the *lexical scope*, meaning that `this` will always refer to the same thing inside and outside of a function.

```
▼ Window {postMessage: f, blur: f, focus: f, close: f, parent: Window, ...} ⓘ
  ► GetParams: f (t)
  ► alert: f alert()
  ► applicationCache: ApplicationCache {status: 0, oncached: null, onchecking: n
  ► atob: f atob()
  ► blur: f ()
  ► btoa: f btoa()
  ► caches: CacheStorage {}
  ► cancelAnimationFrame: f cancelAnimationFrame()
  ► cancelIdleCallback: f cancelIdleCallback()
  ► captureEvents: f captureEvents()
```

function this (arrow)

This is why we use normal `function() {}` declarations when creating d3 event listeners, so we can access the targeted element with `this`.

Destroying d3 event listeners

Before we look at adding events to our charts, let's learn how to destroy our event handlers. Removing old event listeners is important for updating charts and preventing memory leaks, among other things.

Let's add a 3 second timeout at the end of our code so we can test that our mouse events are working before we destroy them.

```
setTimeout(() => {
}, 3000)
```

Removing a d3 event listener is easy — all we need to do is call `.on()` with `null` as the triggered function.

```
setTimeout(() => {
  rects
    .on("mouseenter", null)
    .on("mouseout", null)
}, 3000)
```

Perfect! Now our hover events will stop working after 3 seconds. You might have noticed that a box might be stuck with its hovered color if it was hovered over when the mouse events were deleted.



hovered boxes - stuck!

Luckily, there's an easy fix!

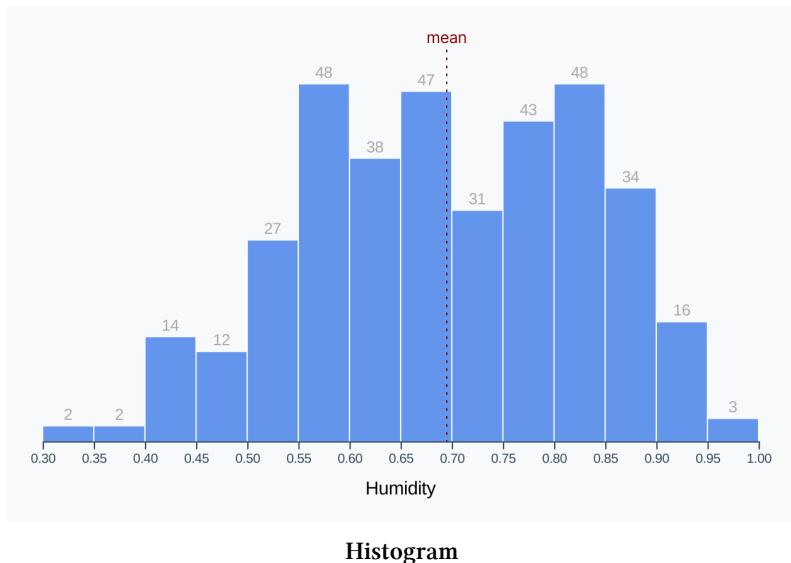
D3 selections have a `.dispatch()` method that will programatically trigger an event — no mouseout needed. We can trigger a `mouseout` event right before we remove it to ensure that our boxes finish in their “neutral” state.

```
setTimeout(() => {
  rects
    .dispatch("mouseout")
    .on("mouseenter", null)
    .on("mouseout", null)
}, 3000)
```

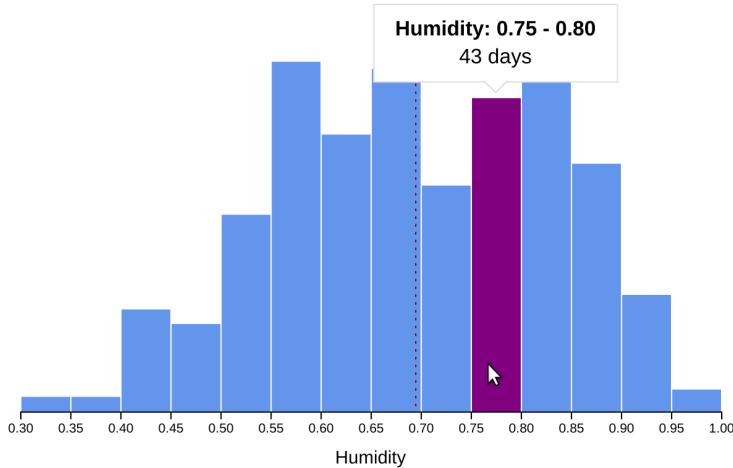
Perfect! Now that we have a good handle on using d3 event listeners, let's use them to make our charts interactive.

Bar chart

Let's add interactions to our histogram — navigate to </code/05-interactions/2-bars/draft/> in the browser and open the `/code/05-interactions/2-bars/draft/bars.js` file in your text editor. We should see the histogram we created in **Chapter 3**.



Our goal in the section is to add an informative tooltip that shows the humidity range and day count when a user hovers over a bar.



histogram finished

We could use d3 event listeners to change the bar's color on hover, but there's an alternative: CSS hover states. To add CSS properties that only apply when an element is hovered over, add `:hover` after the selector name. It's good practice to place this selector immediately after the non-hover styles to keep all bar styles in one place.

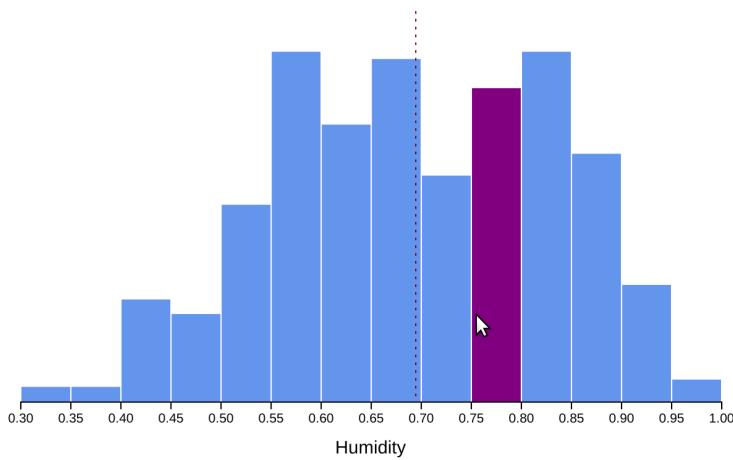
Let's add a new selector to the `/code/05-interactions/2-bars/draft/styles.css` file.

```
.bin rect:hover {  
}
```

Let's have our bars change their fill to purple when we hover over them.

```
.bin rect:hover {  
    fill: purple;  
}
```

Great, now our bars should turn purple when we hover over them and back to blue when we move our mouse out.



histogram with hover state

Now we know how to implement hover states in two ways: CSS hover states and event listeners. Why would we use one over the other?

CSS hover states are good to use for more *stylistic* updates that don't require DOM changes. For example, changing colors or opacity. If we're using a CSS preprocessor like SASS, we can use any color variables instead of duplicating them in our JavaScript file.

JavaScript event listeners are what we need to turn to when we need a more complicated hover state. For example, if we want to update the text of a tooltip or move an element, we'll want to do that in JavaScript.

Since we need to update our tooltip text and position when we hover over a bar, let's add our `mouseenter` and `mouseleave` event listeners at the bottom of our `bars.js` file. We can set ourselves up with named functions to keep our chained code clean and concise.

```
binGroups.select("rect")
  .on("mouseenter", onMouseEnter)
  .on("mouseleave", onMouseLeave)

function onMouseEnter(datum) {
}

function onMouseLeave(datum) {
}
```

Starting with our `onMouseEnter()` function, we'll start by grabbing our tooltip element. If you look in our `index.html` file, you can see that our template starts with a tooltip with two children: a div to display the range and a div to display the value. We'll follow the common convention of using **ids** as hooks for JavaScript and **classes** as hooks for CSS. There are two main reasons for this distinction:

1. We can use classes in multiple places (if we wanted to style multiple elements at once) but we'll only use an id in one place. This ensures that we're selecting the correct element in our chart code
2. We want to separate our chart manipulation code and our styling code — we should be able to move our chart hook without affecting the styles.

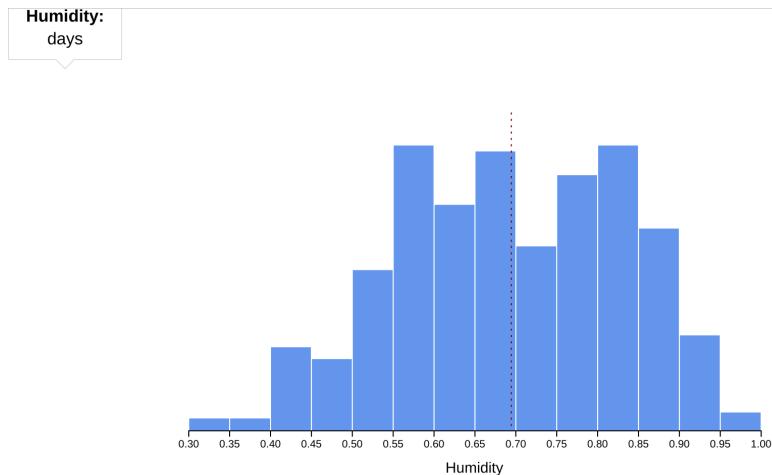
We could create our tooltip in JavaScript, the same way we have been creating and manipulating SVG elements with d3. We have it defined in our HTML file here, which is generally easier to read and maintain since the tooltip layout is static.

If we open up our `styles.css`, we can see our basic tooltip styles, including using a pseudo-selector `.tooltip:before` to add an arrow pointing down (at the hovered bar). Also note that the tooltip is hidden (`opacity: 0`) and will transition any property changes (`transition: all 0.2s ease-out`). It also will not receive any mouse events (`pointer-events: none`) to prevent from stealing the mouse events we'll be implementing.

Let's comment out the `opacity: 0` property so we can get a look at our tooltip.

```
.tooltip {  
  /* opacity: 0; */
```

We can see that our tooltip is positioned in the top left of our page.



histogram with visible tooltip - far away!

If we position it instead at the top left of our chart, we'll be able to shift it based on the hovered bar's position in the chart.

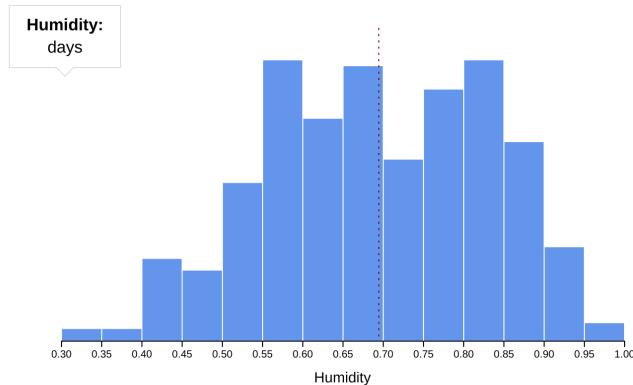
We can see that our tooltip is absolutely positioned all the way to the left and 12px above the top (to offset the bottom triangle). So why isn't it positioned at the top left of our chart?

Absolutely positioned elements are placed relative to their **containing block**. The default containing block is the `<html>` element, but will be overridden by certain ancestor elements. The main scenario that will create a new containing block is if the element has a **position** other than the default (`static`). There are other scenarios, but they are much more rare (for example, if a `transform` is specified).

This means that our tooltip will be positioned at the top left of the nearest ancestor element that has a set position. Let's give our `.wrapper` element a position of `relative`.

```
.wrapper {  
  position: relative;  
}
```

Perfect! Now our tooltip is located at the top left of our chart and ready to be shifted into place when a bar is hovered over.



histogram with visible tooltip

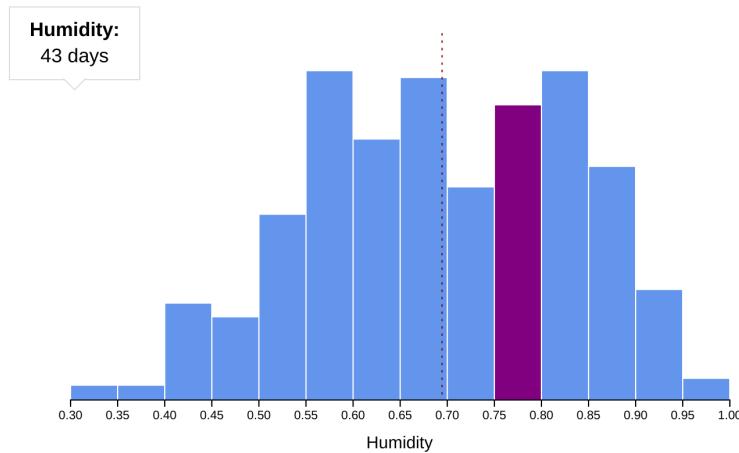
Let's start adding our mouse events in `bars.js` by grabbing the existing tooltip using its id (`#tooltip`). Our tooltip won't change once we load the page, so let's define it outside of our `onMouseEnter()` function.

```
const tooltip = d3.select("#tooltip")  
function onMouseEnter(datum) {  
}
```

Now let's start fleshing out our `onMouseEnter()` function by updating our tooltip text to tell us about the hovered bar. Let's select the nested `#count` element and update it to display the y value of the bar. Remember, in our histogram the y value is the number of days in our dataset that fall in that humidity level range.

```
const tooltip = d3.select("#tooltip")
function onMouseEnter(datum) {
    tooltip.select("#count")
        .text(yAccessor(datum))
}
```

Looking good! Now our tooltip updates when we hover over a bar to show that bar's count.

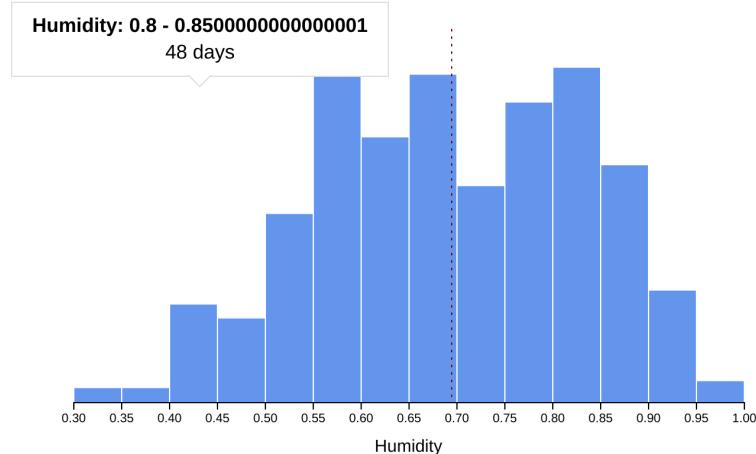


histogram tooltip with count

Next, we can update our range value to match the hovered bar. The bar is covering a range of humidity values, so let's make an array of the values and join them with a `-` (which can be easier to read than a template literal).

```
tooltip.select("#range")
    .text([
        datum.x0,
        datum.x1
    ].join(" - "))
```

Our tooltip now updates to display both the count *and* the range, but it might be a bit *too* precise.



histogram tooltip with count and range

We could convert our range values to strings and slice them to a certain precision, but there's a better way. It's time to meet `d3.format()`.

The `d3-format`²⁹ module helps turn numbers into nicely formatted strings. Usually when we display a number, we'll want to parse it from its raw format. For example, we'd rather display 32,000 than 32000 — the former is easier to read and will help with scanning a list of numbers.

If we pass `d3.format()` a **format specifier string**, it will create a formatter function. That formatter function will take one parameter (a number) and return a formatted string. There are many possible format specifier strings — let's go over the format for the options we'll use the most often.

`[,][.precision][type]`

Each of these specifiers is optional — if we use an empty string, our formatter will just return our number as a string. Let's talk about what each specifier tells our formatter.

`,: add commas every 3 digits to the left of the decimal place`

`.precision: give me this many numbers after the decimal place.`

`type: each specific type is declared by using a single letter or symbol. The most handy types are:`

²⁹<https://github.com/d3/d3-format>

- **f**: fixed point notation — give me `precision` many decimal points
- **r**: decimal notation — give me `precision` many significant digits and pad the rest until the decimal point
- **%**: percentage — multiply my number by 100 and return `precision` many decimal points

Run through a few examples in your terminal to get the hang of it.

```
d3.format( ".2f")(11111.111) // "11111.11"
d3.format(",.2f")(11111.111) // "11,111.11"
d3.format(",.0f")(11111.111) // "11,111"
d3.format(",.4r")(11111.111) // "11,110"
d3.format( ".2%")(0.111)     // "11.10%"
```

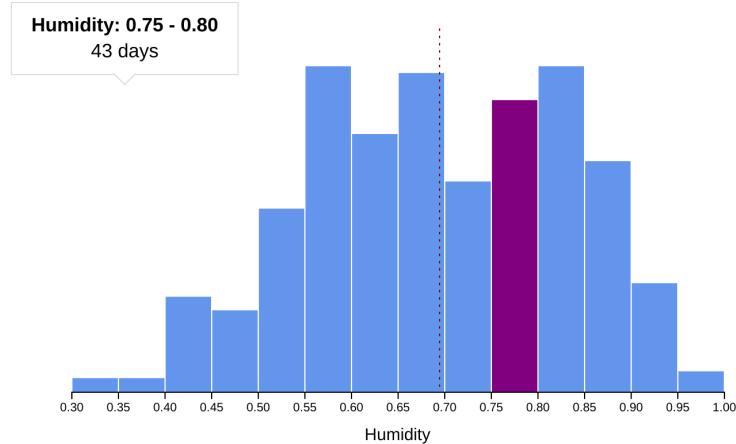
Let's create a formatter for our humidity levels. Two decimal points should be enough to differentiate between ranges without overwhelming our user with too many 0s.

```
const formatHumidity = d3.format(".2f")
```

Now we can use our formatter to clean up our humidity level numbers.

```
const formatHumidity = d3.format(".2f")
tooltip.select("#range")
  .text([
    formatHumidity(datum.x0),
    formatHumidity(datum.x1)
  ].join(" - "))
```

Nice! An added benefit to our number formatting is that our range numbers are the same width for every value, preventing our tooltip from jumping around.



histogram tooltip with count and formatted range

Next, we want to position our tooltip *horizontally centered* above a bar when we hover over it. To calculate our tooltip's x position, we'll need to take three things into account:

- the bar's x position in the chart (`xScale(datum.x0)`),
- half of the bar's width (`((xScale(datum.x1) - xScale(datum.x0)) / 2)`), and
- the margin by which our **bounds** are shifted right (`dimensions.margin.left`).

Remember that our tooltip is located at the top left of our **wrapper** - the outer container of our chart. But since our bars are within our **bounds**, they are shifted by the margins we specified.

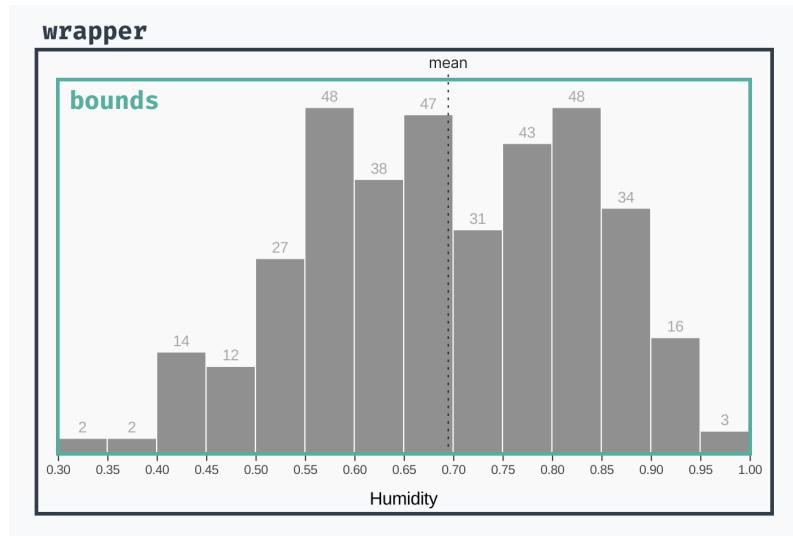


Chart terminology

Let's add these numbers together to get the x position of our tooltip.

```
const x = xScale(datum.x0)
+ (xScale(datum.x1) - xScale(datum.x0)) / 2
+ dimensions.margin.left
```

When we calculate our tooltip's y position, we don't need to take into account the bar's dimensions because we want it placed above the bar. That means we'll only need to add two numbers:

1. the bar's y position (`yScale(yAccessor(datum))`), and
2. the `margin` by which our **bounds** are shifted `down` (`dimensions.margin.top`)

```
const y = yScale(yAccessor(datum))
+ dimensions.margin.top
```

Let's use our x and y positions to shift our tooltip. Because we're working with a normal xHTML `div`, we'll use the CSS `translate` property.

```
tooltip.style("transform", `translate(`  
+ `${x}px,`  
+ `${y}px`  
+ `)`)
```

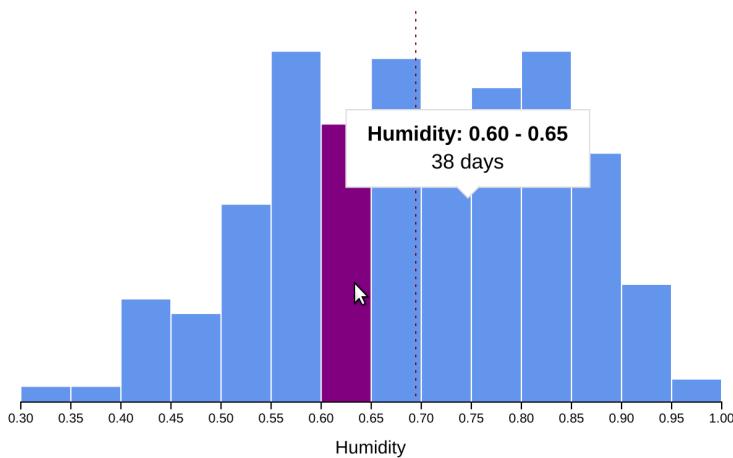
Why are we setting the **transform** CSS property and not **left** and **top**? A good rule of thumb is to avoid changing (and especially animating) CSS values other than **transform** and **opacity**. When the browser styles elements on the page, it runs through several steps:

1. calculate style
2. layout
3. paint, and
4. layers

Most CSS properties affect steps 2 or 3, which means that the browser has to perform that step and the subsequent steps every time that property is changed. **Transform** and **opacity** only affect step 4, which cuts down on the amount of work the browser has to do. Read more about each step and this distinction at <https://www.html5rocks.com/en/tutorials/speed/high-performance-animations/>^a.

^a<https://www.html5rocks.com/en/tutorials/speed/high-performance-animations/>

Hmm, why is our tooltip in the wrong position? It looks like we're positioning the top left of the tooltip in the right location (above the hovered bar).



histogram tooltip, unshifted

We want to position the **bottom, center** of our tooltip (the tip of the arrow) above the bar, instead. We *could* find the tooltip size by calling the `.getBoundingClientRect()` method, but there's a computationally cheaper way.

There are a few ways to shift absolutely positioned elements using CSS properties:

- `top, left, right, and bottom`
- `margins`
- `transform: translate()`

All of these properties can receive percentage values, but some of them are based on different dimensions.

- `top and bottom: percentage of the parent's height`
- `left and right: percentage of the parent's width`
- `margins: percentage of the parent's width`
- `transform: translate(): percentage of the specified element`

We're interested in shifting the tooltip based on its own height and width, so we'll need to use `transform: translate()`. But we're *already* applying a `translate` value — **how can we set the `translate` value using a pixel amount and a width?**

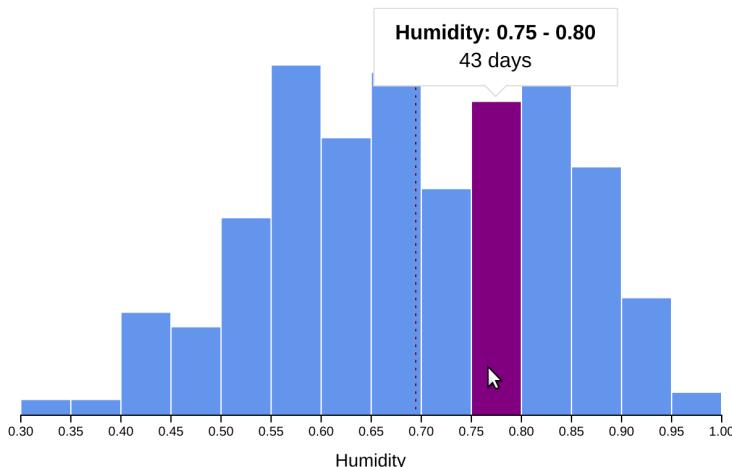
CSS `calc()` comes to the rescue here! We can tell CSS to calculate an offset based on values with different units. For example, the following CSS rule would cause an element to be 20 pixels wider than its container.

```
width: calc(100% + 20px);
```

Let's use `calc()` to offset our tooltip **up** half of its own width (`-50%`) and **left** `-100%` of its own height. This is in addition to our calculated `x` and `y` values.

```
tooltip.style("transform", `translate(`  
+ `calc( -50% + ${x}px),`  
+ `calc(-100% + ${y}px)`  
+ `)`)
```

Perfect! Now our tooltip moves to the exact location we want.



histogram finished

We have one last task to do — hide the tooltip when we're not hovering over a bar. Let's un-comment the `opacity: 0` rule in `styles.css` so its hidden to start.

```
.tooltip {  
  opacity: 0;
```

Jumping back to our bars.js file, we need to make our tooltip visible at the end of our onMouseEnter() function.

```
tooltip.style("opacity", 1)
```

Lastly, we want to make our tooltip invisible again whenever our mouse leaves a bar. Let's add that to our onMouseLeave() function.

```
function onMouseLeave() {  
  tooltip.style("opacity", 0)  
}
```

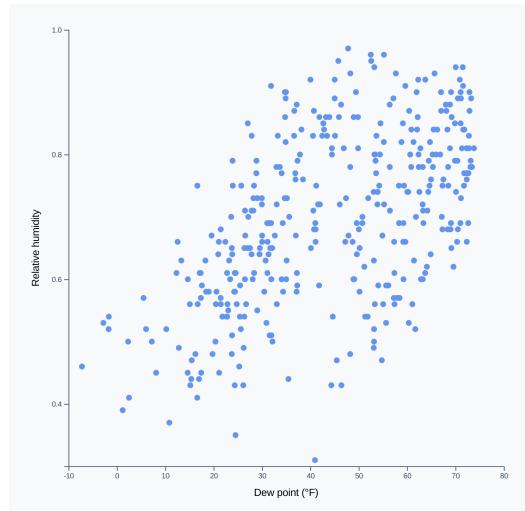
Look at that! You just made an interactive chart that gives users more information when they need it. Positioning tooltips is not a simple feat, so give yourself a pat on the back! Next up, we'll learn an even fancier method for making it easy for users to get tooltips even for small, close-together elements.

Scatter plot

Let's level up and add tooltips to a scatter plot. Navigate to

</code/05-interactions/3-scatter/draft/> in your browser and open up the </code/05-interactions/3-scatter/draft/scatter.js> file.

You should see the scatter plot we made in **Chapter 2** — no interactions here yet.



Scatter plot

We want a tooltip to give us more information when we hover over a point in our chart. Let's go through the steps from last chapter.

At the bottom of the file, we'll select all of our `<circle>` elements and add a `mouseenter` and a `mouseleave` event.

```
bounds.selectAll("circle")
  .on("mouseenter", onMouseEnter)
  .on("mouseleave", onMouseLeave)
```

We know that we'll need to modify our `#tooltip` element, so let's assign that to a variable. Let's also define our `onMouseEnter()` and `onMouseLeave()` functions.

```
const tooltip = d3.select("#tooltip")
function onMouseEnter(datum, index) {
}

function onMouseLeave() {
```

Let's first fill out our `onMouseEnter()` function. We want to display two values:

- the metric on our x axis (**dew point**), and
- the metric on our y axis (**humidity**).

For both metrics, we'll want to define a string formatter using `d3.format()`. Then we'll use that formatter to set the `text` value of the relevant `` in our tooltip.

```
function onMouseEnter(datum) {  
  const formatHumidity = d3.format(".2f")  
  tooltip.select("#humidity")  
    .text(formatHumidity(yAccessor(datum)))  
  
  const formatDewPoint = d3.format(".2f")  
  tooltip.select("#dew-point")  
    .text(formatDewPoint(xAccessor(datum)))  
  
}
```

Let's add an extra bit of information at the bottom of this function — users will probably want to know the date of the hovered point. Our data point's date is formatted as a string, but not in a very human-readable format (for example, "2019-01-01"). Let's use `d3.timeParse` to turn that string into a date that we can re-format.

```
const dateParser = d3.timeParse("%Y-%m-%d")  
console.log(dateParser(datum.date))
```

Now we need to turn our date object into a friendlier string. The **d3-time-format**³⁰ module can help us out here! `d3.timeFormat()` will take a date formatter string and return a formatter function.

The date formatter string uses the same syntax as `d3.timeParse` — it follows four rules:

1. it will return the string verbatim, other than specific directives,
2. these directives contain a percent sign and a letter,

³⁰<https://github.com/d3/d3-time-format>

3. usually the letter in a directive has two formats: lowerbase (abbreviated) and uppercase (full), and
4. a dash (-) between the percent sign and the letter prevents padding of numbers.

For example, `d3.timeFormat("%Y") (new Date())` will return the current year.

Let's learn a few handy directives:

- `%Y`: the full year
- `%y`: the last two digits of the year
- `%m`: the padded month (eg. "01")
- `%-m`: the non-padded month (eg. "1")
- `%B`: the full month name
- `%b`: the abbreviated month name
- `%A`: the full weekday name
- `%a`: the abbreviated weekday name
- `%d`: the day of the month

See the full list of directives at <https://github.com/d3/d3-time-format³¹>.

Now, let's create a formatter string that prints out a friendly date.

```
const dateParser = d3.timeParse("%Y-%m-%d")
const formatDate = d3.timeFormat("%B %A %-d, %Y")
console.log(formatDate(dateParser(datum.date)))
```

Much better! Let's plug that in to our tooltip.

```
const dateParser = d3.timeParse("%Y-%m-%d")
const formatDate = d3.timeFormat("%B %A %-d, %Y")
tooltip.select("#date")
  .text(formatDate(dateParser(datum.date)))
```

Next, we'll grab the `x` and `y` value of our dot, offset by the `top` and `left` margins.

³¹<https://github.com/d3/d3-time-format>

```
const x = xScale(xAccessor(datum))  
+ dimensions.margin.left  
const y = yScale(yAccessor(datum))  
+ dimensions.margin.top
```

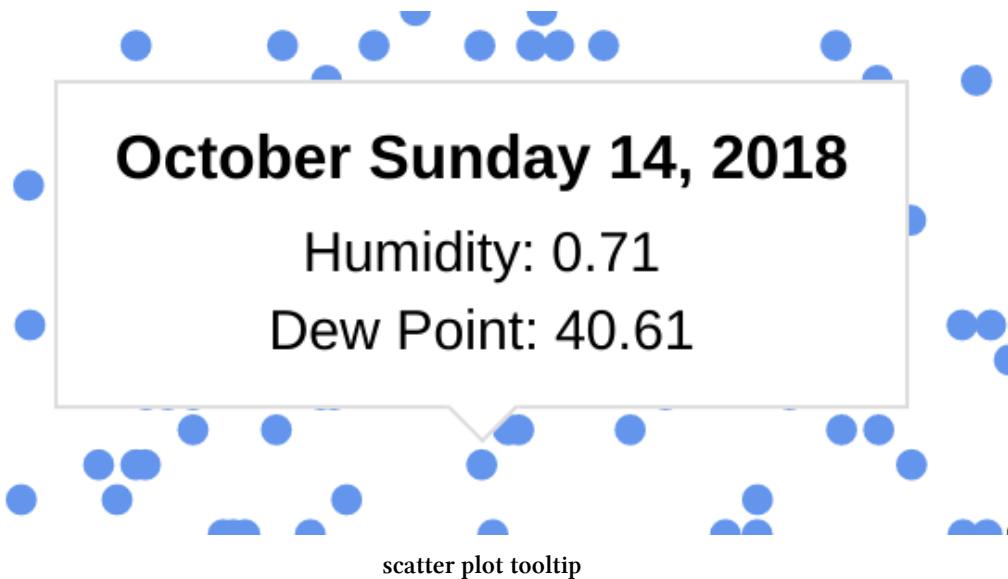
Just like with our bars, we'll use `calc()` to add these values to the percentage offsets needed to shift the tooltip. Remember, this is necessary so that we're positioning its arrow, not the top left corner.

```
tooltip.style("transform", `translate(`  
+ `calc( -50% + ${x}px),`  
+ `calc(-100% + ${y}px)`  
+ `)`)
```

Lastly, we'll make our tooltip visible and hide it when we mouse out of our dot.

```
tooltip.style("opacity", 1)  
}  
  
function onMouseLeave() {  
  tooltip.style("opacity", 0)  
}
```

Nice! Adding a tooltip was much faster the second time around, wasn't it?



Those tiny dots are hard to hover over, though. The small hover target makes us focus really hard to move our mouse *exactly* over a point. To make things worse, our tooltip disappears when moving between points, making the whole interaction a little jerky.

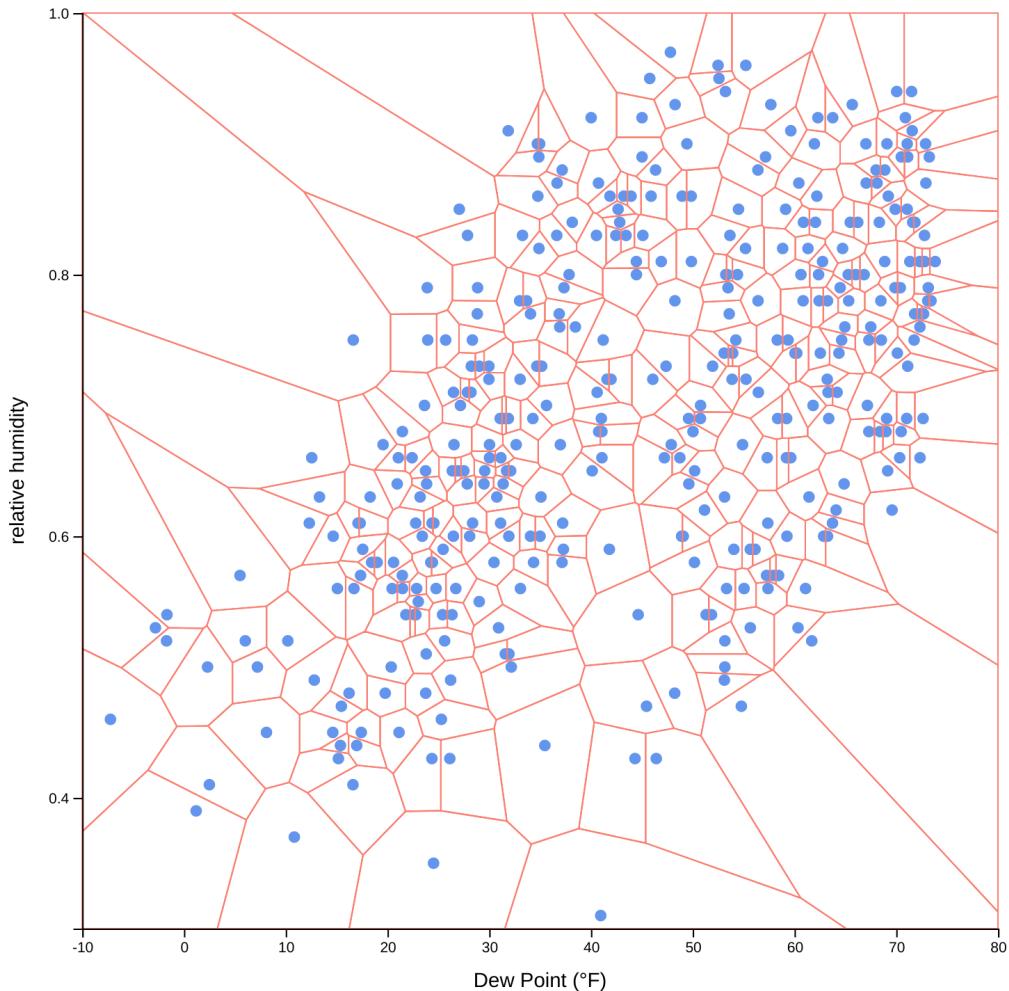
Don't worry! We have a very clever solution to this problem.

Voronoi

Let's talk briefly about **voronoi diagrams**. For every location on our scatter plot, there is a dot that is the closest. A voronoi diagram partitions a plane into regions based on the closest point. Any location within each of these parts agrees on the closest point.

Voronoi are useful in many fields — from creating art to detecting neuromuscular diseases to developing predictive models for forest fires.

Let's look at what our scatter plot would look like when split up with a voronoi diagram.



scatter plot with voronoi

See how each point in our scatter plot is inside of a cell? If you chose any location in that cell, that point would be the closest.

There is a voronoi generator built into the main d3 bundle: `d3-voronoi`³². However, this module is deprecated and the speedier `d3-delaunay`³³ is recommended instead.

³²<https://github.com/d3/d3-voronoi>

³³<https://github.com/d3/d3-delaunay>

We've included the minified version of this module at `/3-scatter/d3-delaunay.min.js` and loaded the script in our `index.html` file, so we'll already have access to `d3.Delaunay`.

Enough talking, let's create our own diagram! Let's add some code at the end of the `Draw data` step, right before the `Draw peripherals` step. Because this is an external library, the API is a bit different from other d3 code. Instead of creating a voronoi generator, we'll create a new **Delaunay triangulation**. A delaunay triangulation³⁴ is a way to join a set of points to create a triangular mesh. To create this, we can pass `d3.Delaunay.from()`³⁵ three parameters:

1. our dataset,
2. an x accessor function, and
3. a y accessor function.

```
const delaunay = d3.Delaunay.from(  
  dataset,  
  d => xScale(xAccessor(d)),  
  d => yScale(yAccessor(d)),  
)
```

Now we want to turn our **delaunay triangulation** into a voronoi diagram – thankfully our triangulation has a `.voronoi()` method.

```
const voronoi = delaunay.voronoi()
```

Let's bind our data and add a `<path>` for each of our data points with a class of "voronoi" (for styling with our `styles.css` file).

³⁴https://en.wikipedia.org/wiki/Delaunay_triangulation

³⁵<https://github.com/d3/d3-delaunay#from>

```
bounds.selectAll(".voronoi")
  .data(dataset)
  .enter().append("path")
    .attr("class", "voronoi")
```

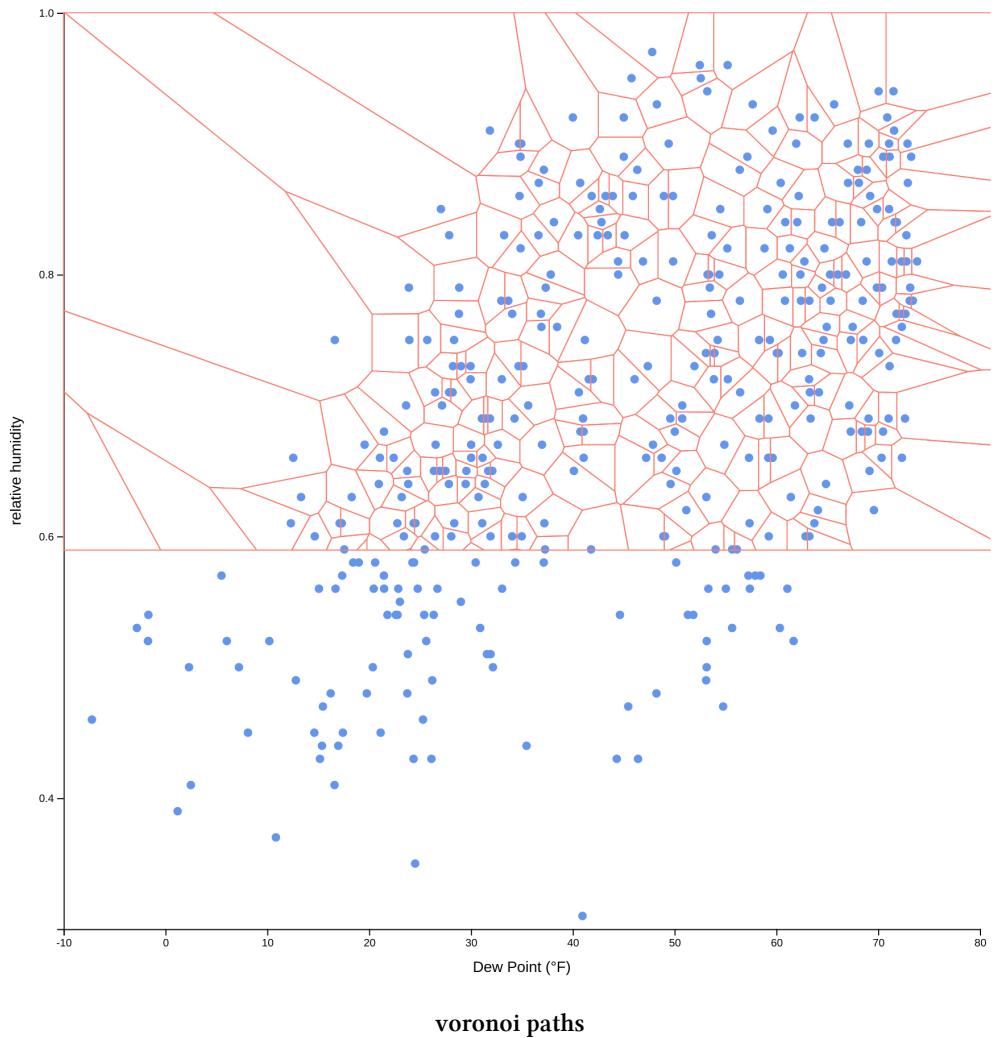
We can create each path's `d` attribute string by passing `voronoi.renderCell()` the *index of our data point*.

```
bounds.selectAll(".voronoi")
  // ...
  .attr("d", (d,i) => voronoi.renderCell(i))
```

Lastly, let's give our paths a `stroke` value of `salmon` so that we can look at them.

```
bounds.selectAll(".voronoi")
  // ...
  .attr("stroke", "salmon")
```

Now when we refresh our webpage, our scatter plot will be split into voronoi cells!



Hmm, our voronoi diagram is wider and shorter than our chart. This is because it has no concept of the size of our bounds, and is using the default size of 960 pixels wide and 500 pixels tall, which we can see if we log out our voronoi object.

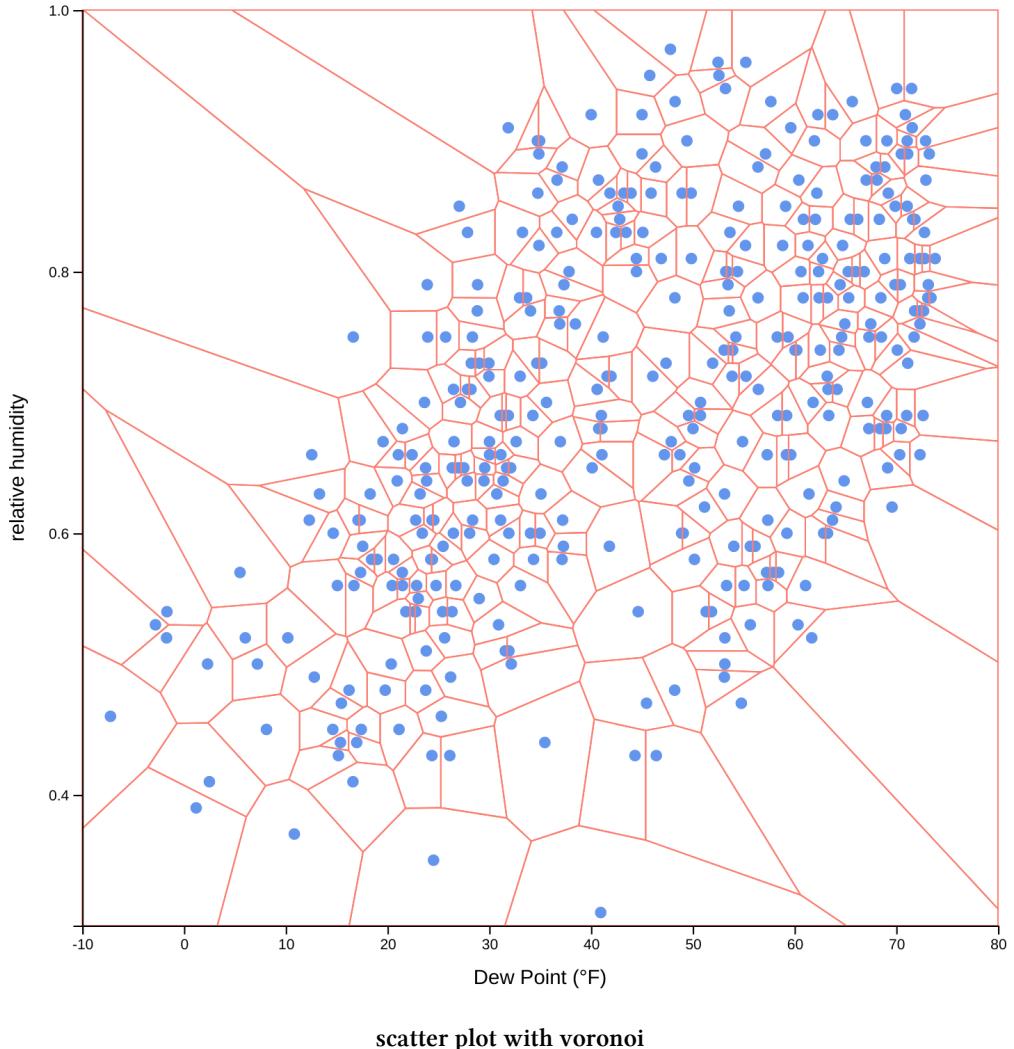
```
scatter.js:106
▼ {delaunay: y, circumcenters: Float64Array(1426), vectors: Float64Array(1
  460), xmax: 960, xmin: 0, ...} 1
  ► circumcenters: Float64Array(1426) [397.0274000000031, 433.3169130370363...
  ► delaunay: y {points: Float64Array(730), halfedges: Int32Array(2139), hu...
  ► vectors: Float64Array(1460) [0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, 0, -194.8...
    xmax: 960
    xmin: 0
    ymax: 500
    ymin: 0
  ► __proto__: Object
```

voronoi paths

Let's specify the size of our diagram by setting our voronoi's .xmax and .ymax values (before we draw our <path>s).

```
const voronoi = delaunay.voronoi()
voronoi.xmax = dimensions.boundedWidth
voronoi.ymax = dimensions.boundedHeight
```

Voila! Now our diagram is the correct size.



What we want is to capture hover events for our paths instead of an individual dot. This will be much easier to interact with because of the contiguous, large hover targets.

Let's remove that last line where we set the `stroke(.attr("stroke", "salmon"))` so our Voronoi cells are invisible. Next, we'll update our interactions, starting by moving our `mouseenter` and `mouseleave` events from the dots to our Voronoi paths.

Note that the mouse events on our dots won't be triggered anymore, since they're covered by our voronoi paths.

```
bounds.selectAll(".voronoi")
// ...
.on("mouseenter", onMouseEnter)
.on("mouseleave", onMouseLeave)
```

When we refresh our webpage, notice how much easier it is to target a specific dot!

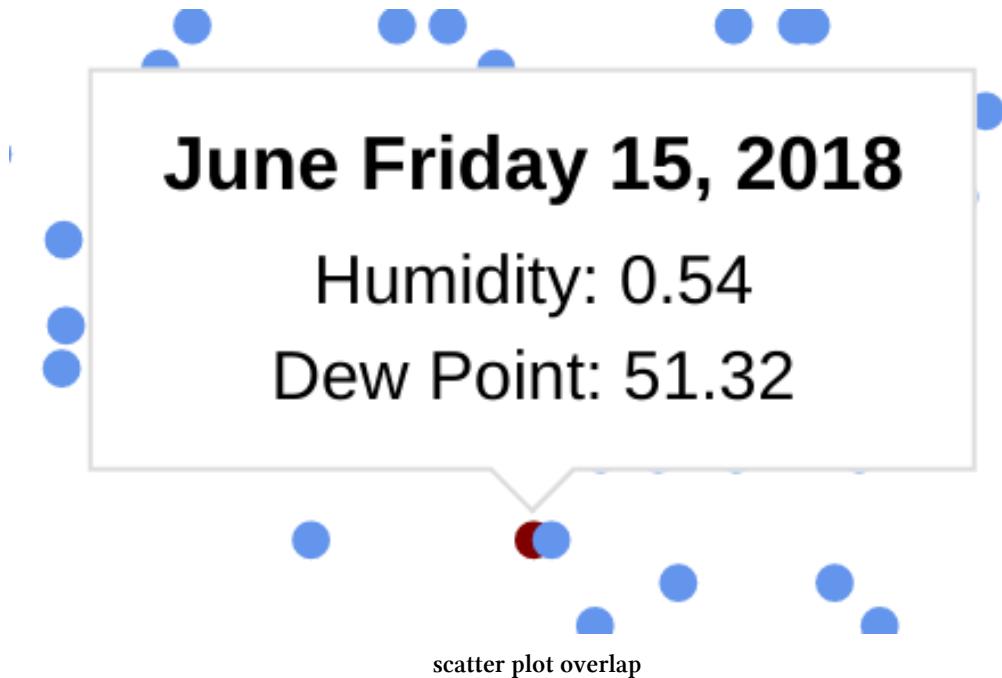
Changing the hovered dot's color

Now that we don't need to directly hover over a dot, it can be a bit unclear which dot we're getting data about. Let's make our dot change color and grow on hover.

The naive approach would involve selecting the corresponding `circle` and changing its fill. Note that d3 selection objects have a `.filter()` method that mimics a native Array's.

```
function onMouseEnter(datum) {
  bounds.selectAll("circle")
    .filter(d => d == datum)
    .style("fill", "maroon")
```

However, we'll run into an issue here. Remember that SVG elements' z-index is determined by their position in the DOM. We can't change our dots' order easily on hover, so any dot drawn after our hovered dot will obscure it.



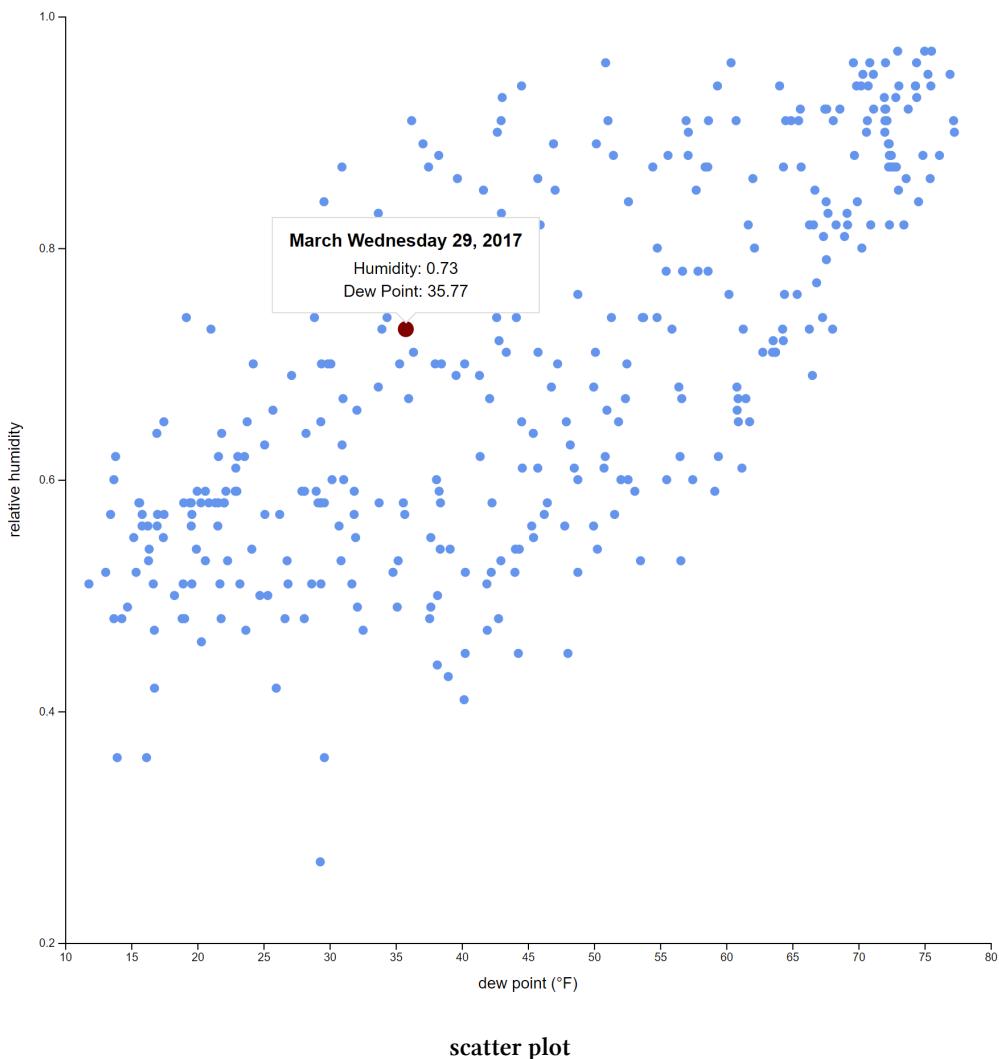
Instead, we'll draw a completely new dot which will appear on top.

```
function onMouseEnter(datum) {  
  const dayDot = bounds.append("circle")  
    .attr("class", "tooltipDot")  
    .attr("cx", xScale(xAccessor(datum)))  
    .attr("cy", yScale(yAccessor(datum)))  
    .attr("r", 7)  
    .style("fill", "maroon")  
    .style("pointer-events", "none")  
}
```

Let's remember to remove this new dot on mouse leave.

```
function onMouseLeave() {  
  d3.selectAll(".tooltipDot")  
    .remove()  
}
```

Now when we trigger a tooltip, we can see our hovered dot clearly!

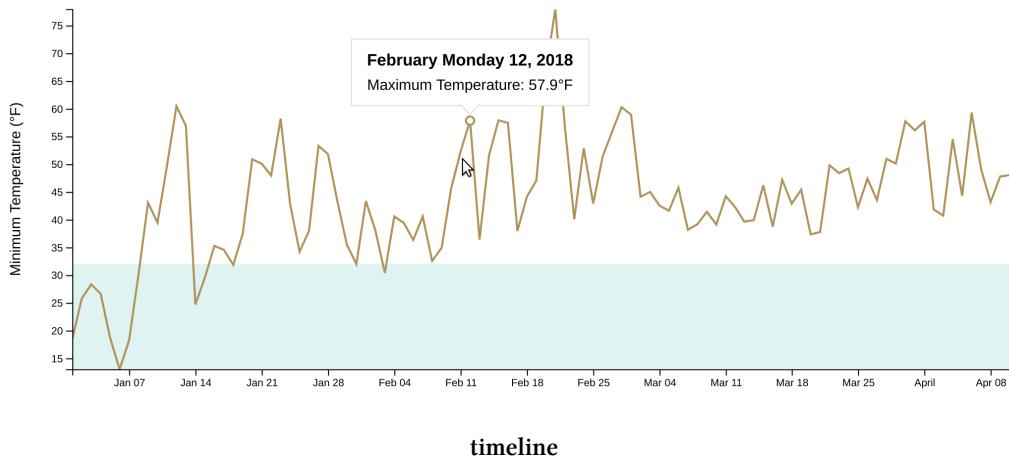


Making a tooltip for our scatter plot was trickier than expected, but we saw how important encouraging interaction can be. When our hover targets were small, it felt like work to get more information about a specific point. But now that we're using voronoi cells, interacting with our chart is almost fun!

Line chart

Let's go through one last example for adding tooltips. So far, we've added tooltips to individual elements (bars, circles, and paths). Adding a tooltip to a timeline is a bit different. Let's dig in by navigating to `/code/05-interactions/4-line/draft/` in the browser and `/code/05-interactions/4-line/draft/line.js` in your text editor.

In this section, we're aiming to add a tooltip to our line chart like this:



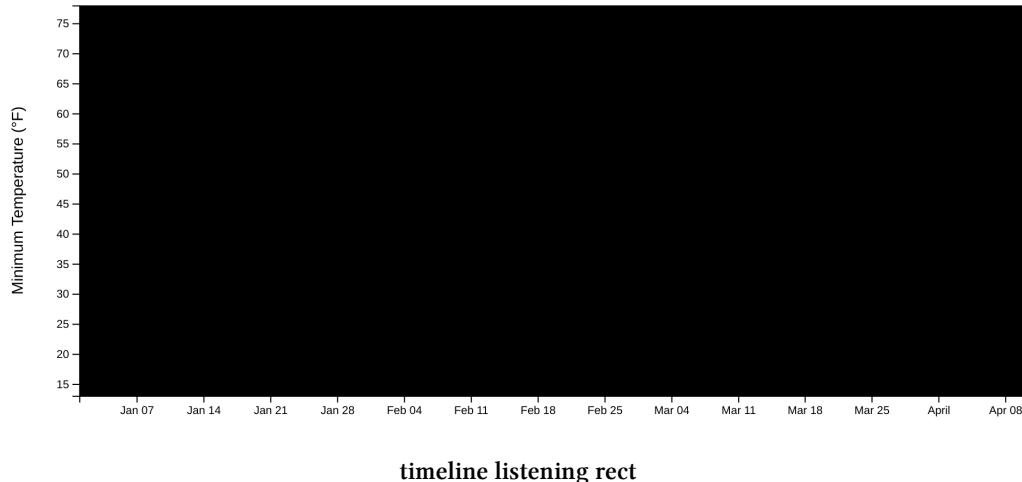
Instead of catching hover events for individual elements, we want to display a tooltip whenever a user is hovering anywhere **on the chart**. Therefore, we'll want an element that spans our entire **bounds**.

To start coding up our **Set up interactions** step, let's create a `<rect>` that covers our **bounds** and add our mouse event listeners to it. This time we'll want to listen for `mousemove` events instead of `mouseenter` events because we'll want to update the tooltip's position when a reader moves their mouse around the chart.

Note that we don't need to define our `<rect>`'s `x` or `y` attributes because they both default to `0`.

```
const listeningRect = bounds.append("rect")
  .attr("class", "listening-rect")
  .attr("width", dimensions.boundedWidth)
  .attr("height", dimensions.boundedHeight)
  .on("mousemove", onMouseMove)
  .on("mouseleave", onMouseLeave)
```

Perfect! We can see that our `listeningRect`, defaulted to a black fill, covers our entire bounds.



Let's add a rule to `styles.css` so we can see our chart again.

```
.listening-rect {
  fill: transparent;
}
```

Great! Now we can set up our `tooltip` variable and `onMouseMove` and `onMouseLeave()` functions (back in our `line.js` file).

```
const tooltip = d3.select("#tooltip")
function onMouseMove() {
}

function onMouseLeave() {
}
```

Let's start fleshing out `onMouseMove` — how will we know the location on our line that we are hovering over? The passed parameters we used previously (`datum`, `index`, and `nodes`) won't be helpful here, and `this` will just point us at the listener rect element.

When an event listener is invoked, the `d3-selection` library sets a global `d3.event`. `d3.event` will refer to the currently triggered event and will be reset when the event listener is done. During the event listener handler, we also get access to a `d3.mouse()` function which will return the `x`, `y` coordinates of the mouse event, relative to a specified container.

Let's see what that would look like in action and pass our listener container to `d3.mouse()`.

```
function onMouseMove() {
  const mousePosition = d3.mouse(this)
  console.log(mousePosition)
```

Now we can see our mouse position as an `[x,y]` array when we move our mouse around the chart.

▶ (2) [220.99826049804688, 103.00520324707031]

mouse coordinates

Test it out — what do the numbers look like when you hover over the top left of the chart? What about the bottom right?

There was a bug that will be fixed in the next version of Firefox (68)^a that fails to properly offset the `d3.mouse()`'s reported mouse coordinates, disregarding any `CSS transform: translate()` properties on ancestor elements within an `<svg>` element.

Until most people have updated their Firefox to more recent versions, make sure that you set the `transform` HTML attribute instead of setting a CSS `transform` property if possible (for example, when shifting your `bounds`), if you intend on grabbing the mouse position.

^ahttps://bugzilla.mozilla.org/show_bug.cgi?id=972041

Great, but in order to show the tooltip next to an actual data point, we need to know which point we're closest to. First, we'll need to figure out what date we're hovering over — how do we convert an x position into a date? So far, we've only used our scales to convert from the data space (in this case, JavaScript date objects) to the pixel space.

Thankfully, d3 scales make this very simple! We can use the same `xScale()` we've used previously, which has an `.invert()` method. `.invert()` will convert our units backwards, from the `range` to the `domain`.

Let's pass the x position of our mouse (`mousePosition[0]`) to the `.invert()` method of our `xScale()`.

```
const mousePosition = d3.mouse(this)
const hoveredDate = xScale.invert(mousePosition[0])
```

Okay great, now know what date we're hovering over — let's figure out how to find the closest data point.

d3.scan()

If you ever need to know where a variable will fit in a sorted list, `d3.scan()`³⁶ can help you out. `d3.scan()` requires two parameters:

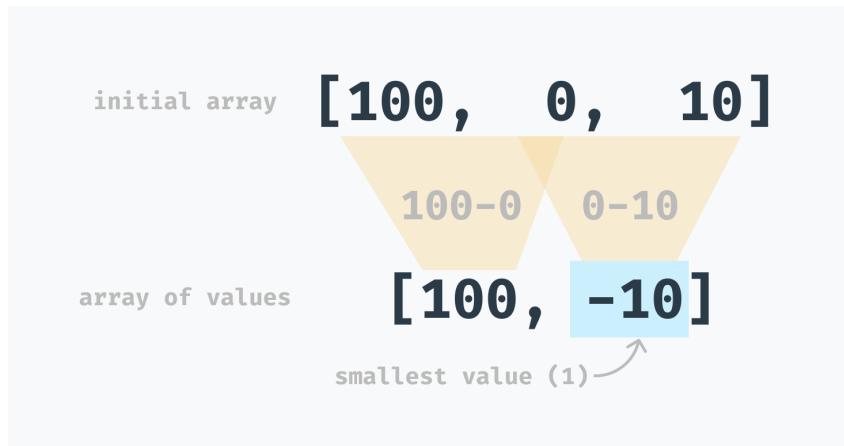
³⁶<https://github.com/d3/d3-array#scan>

1. an array (in this case, our dataset), and
2. an optional comparator function.

The comparator function will take two adjacent items in the passed array and return a numerical value. `d3.scan()` will take those returned values and return the *index* of the *smallest value*.

Let's look at a few examples to get that description to *click*:

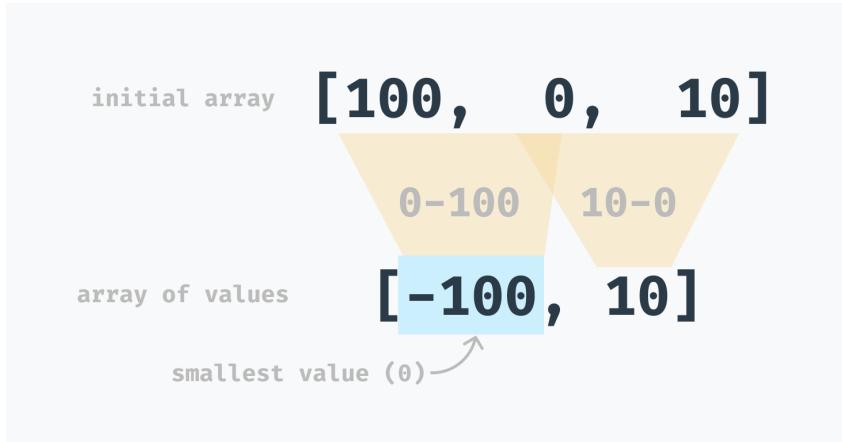
`d3.scan([100, 0, 10], (a,b) => a - b)` would create an array of values that looks like [100, -10].



`d3.scan()` example

This expression would then return 1 because the second item in the array of values is the smallest (remember, the second item is referred to as 1 when we're looking at zero-indexed indices).

`d3.scan([100, 0, 10], (a,b) => b - a)` would create the array `[-100, 10]`



`d3.scan()` example

This expression would then return `0`, because the first item of the array of values is the smallest.

Let's try it out — we'll first create a function to find the distance between the hovered point and a datapoint. We don't care if the point is before or after the hovered date, so we'll use `Math.abs()` to convert that distance to an absolute distance.

```
const getDistanceFromHoveredDate = d => Math.abs(  
  xAccessor(d) - hoveredDate  
)
```

Then we can use that function to compare the two data points in our `d3.scan()` comparator function. This will create an array of distances from the hovered point, and we'll get the index of the closest data point to our hovered date.

```
const closestIndex = d3.scan(dataset, (a, b) => (  
  getDistanceFromHoveredDate(a) - getDistanceFromHoveredDate(b)  
)
```

Next, we need to grab the data point at that index.

```
const closestDataPoint = dataset[closestIndex]
```

Let's `console.table(closestDataPoint)` to make sure we're grabbing the right value.

(index)	Value
apparentTemperatureMinTi...	1483380000
precipType	"rain"
temperatureMin	37.22
summary	"Clear throughout the da...
dewPoint	35.15
apparentTemperatureMax	40.43
temperatureMax	45.34
temperatureMaxTime	1483434000
windBearing	223
moonPhase	0.16
sunsetTime	1483433080
pressure	1023.9
uvIndexTime	1483423200
apparentTemperatureMin	29.68
date	"2017-01-03"
icon	"clear-day"
apparentTemperatureMaxTi...	1483423200
humidity	0.78
windSpeed	7.63
time	1483372800
uvIndex	1
sunriseTime	1483398430
temperatureMinTime	1483380000

► Object

console table example

When we move our mouse to the left of our chart, we should see dates close to the beginning of our dataset, which increase as we move right.

Perfect! Now let's grab the closest x and y values using our accessor functions —

these will come in handy when we're updating our tooltip.

```
const closestXValue = xAccessor(closestDataPoint)
const closestYValue = yAccessor(closestDataPoint)
```

We can use our `closestXValue` to set the date in our tooltip. Let's also format it nicely using `d3.timeFormat()` with the same specifier string we used for our scatter plot.

```
const formatDate = d3.timeFormat("%B %A %-d, %Y")
tooltip.select("#date")
  .text(formatDate(closestXValue))
```

Next up, we can set the temperature value in our tooltip — this time our formatter string will also add a $^{\circ}$ F suffix to clarify.

```
const formatTemperature = d => `${d3.format(".1f")(d)} $^{\circ}$ F`
tooltip.select("#temperature")
  .text(formatTemperature(closestYValue))
```

Lastly, we'll want to grab the x and y position of our closest point, shift our tooltip, and hide/show our tooltip appropriately. This should look like what we've done in the past two sections.

```
const x = xScale(closestXValue)
  + dimensions.margin.left
const y = yScale(closestYValue)
  + dimensions.margin.top

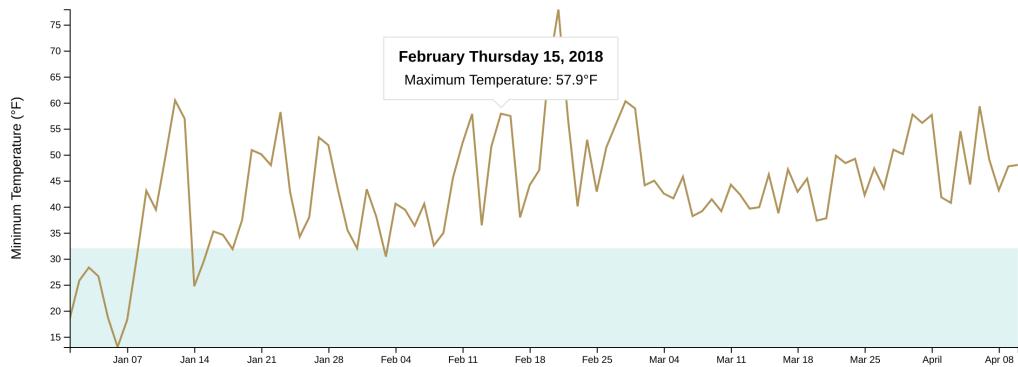
tooltip.style("transform", `translate(
  + calc( -50% + ${x}px) ,
  + calc(-100% + ${y}px)
  + )`)

tooltip.style("opacity", 1)
```

```
}
```

```
function onMouseLeave() {  
  tooltip.style("opacity", 0)  
}
```

Wonderful! When we refresh our webpage, we can see a tooltip that will match the horizontal position of our cursor, while sitting just above our line.



timeline with tooltip

Extra credit

You may notice an issue that we had before with our scatter plot — it's not immediately clear what point we're hovering over. Let's solve this by positioning a `<circle>` over the spot we're hovering — this should make the interaction clearer and the dataset more tangible.

First, we need to create our circle element — let's draw it right after we create our `tooltip` variable. We'll hide it with an `opacity` of `0` to start.

```
const tooltip = d3.select("#tooltip")
const tooltipCircle = bounds.append("circle")
  .attr("r", 4)
  .attr("stroke", "#af9358")
  .attr("fill", "white")
  .attr("stroke-width", 2)
  .style("opacity", 0)
```

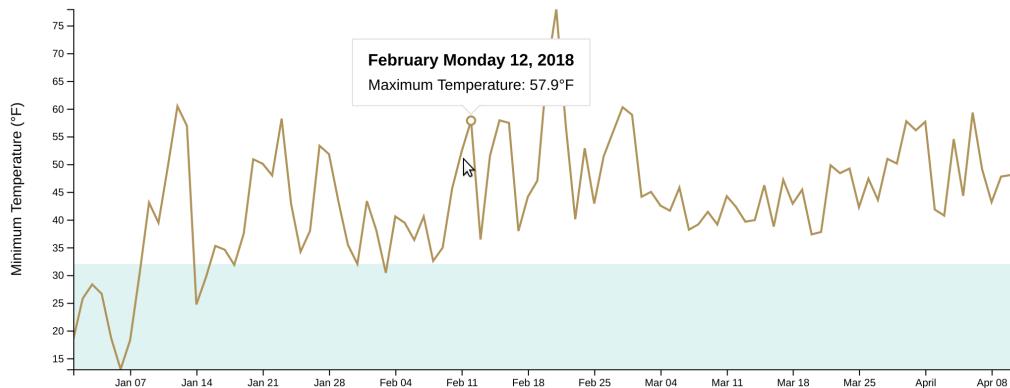
Now, right after we position our tooltip in `onMouseEnter()`, we can also position our `tooltipCircle` and give it an opacity of 1.

```
tooltipCircle
  .attr("cx", xScale(closestXValue))
  .attr("cy", yScale(closestYValue))
  .style("opacity", 1)
}
```

Lastly, we'll hide it in `onMouseLeave()` after we hide our tooltip.

```
function onMouseLeave() {
  tooltip.style("opacity", 0)
  tooltipCircle.style("opacity", 0)
}
```

Voila! Now we should see a circle under our tooltip, right over the “hovered” point.



finished timeline

Give it a spin and feel out the difference. Putting yourself in the user's shoes, you can see how highlighting the hovered data point makes the data feel more tangible.

Making a map

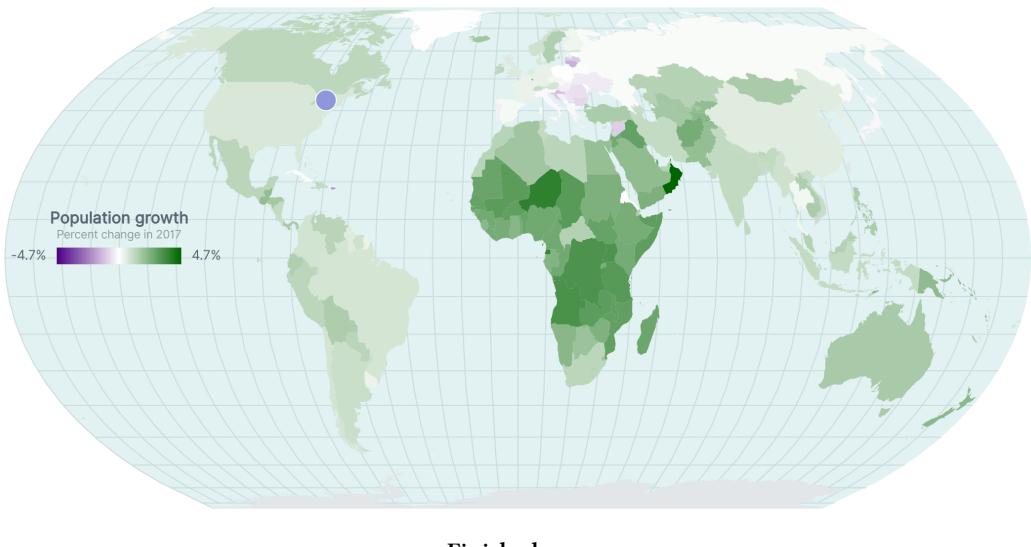
Let's take a step back from our weather data for a chapter and replace it with a dataset of population growth per country. We could visualize our country data as a bar chart, which would be great for finding the countries with the highest population growth. Instead, let's visualize this data using a **choropleth map**, which is a map with shaded areas representing a metric's values.

Note that a map isn't always the best way to visualize geographic data — that depends on the goal of the visualization (and the geographic literacy of your audience). One of the main benefit of maps is that they mimic our knowledge of the world — name a country and the first thing that comes to mind is where it's located in the world. This knowledge helps to navigate the visualization — there are almost 200 countries, but we can almost instantaneously find our own.

Maps are also uniquely good at answering several questions, such as:

- Do neighboring countries tend to have the same amount of population growth?
- Are there patterns across continents?
- Do geographically larger countries tend to have more or less population growth?

In this chapter, we'll build the following map:



Finished map

Digging in

The first step to making a map is to decide what it will be composed of. Will it have streets, mountains, cities, or lakes? We want to visualize data at the country level, so we need to make a map that consists of every country. Hmm, how do we find the shape and location of each country?

To create our map, we'll use the `d3-geo`³⁷ module which accepts GeoJSON data.

What is GeoJSON?

GeoJSON is a format used to represent geographic structures (a geometry, a feature, or a collection of features). A GeoJSON object can contain features of the following types: `Point`, `MultiPoint`, `LineString`, `MultiLineString`, `Polygon`, `MultiPolygon`, `GeometryCollection`, `Feature`, or `FeatureCollection`. If you're curious, check out the spec³⁸ for more information.

³⁷<https://github.com/d3/d3-geo>

³⁸<https://tools.ietf.org/html/rfc7946>

There are many sources of GeoJSON files. We'll use **Natural Earth**³⁹, which is a large collection of public domain map data of various features, locations, and granularities. We'll use the **Admin 0 - Countries** dataset, downloadable [here](#)⁴⁰.

Feel free to either follow along and generate the GeoJSON file yourself or read through this section as an observer. Either way, you'll get an idea of how to create your own GeoJSON file for your custom maps. This resource will still be here when you want to dive deeper.

When we download the countries from **Natural Earth**, we'll get a zip file of various formats.



Natural Earth zip files

We're interested in the **shapefile**: `ne_50m_admin_0_countries.shp`. A **shapefile** is another format for representing geographic data. Let's extract it into our `/code/06-making-a-map/` folder to turn it into a GeoJSON file.

Thankfully, there are many tools for converting **shapefiles** into GeoJSON objects. We'll use the **Geographic Data Abstraction Library (GDAL)**⁴¹ - check out [this page](#)⁴² to download it, or if you're on a Mac, run (`brew install gdal`).

Once we have GDAL installed, we can convert our `ne_50m_admin_0_countries.shp` **shapefile** into a JSON file containing a **GeoJSON** object. The following command will do that and throw the output into a file called `world-geojson2.json`. (The

³⁹<https://www.naturalearthdata.com/>

⁴⁰<https://www.naturalearthdata.com/downloads/50m-cultural-vectors/50m-admin-0-countries-2/>

⁴¹<https://www.gdal.org/>

⁴²<http://trac.osgeo.org/gdal/wiki/DownloadingGdalBinaries>

`world-geojson.json` file already exists in our `/code/06-making-a-map/` folder, allowing us to skip this step if preferred).

```
ogr2ogr -f GeoJSON ./world-geojson2.json ./ne_50m_admin_0_countries.shp
```

The `-f` flag specifies that we want the command to output GeoJSON data.

Great! Now that we have our GeoJSON data, we can import it into our map drawing file. Let's start our server (`live-server`) and open up the `/code/06-making-a-map/draft/draw-map.js` file.

The completed code is available in the `/code/06-making-a-map/completed/` folder if you need to peek at any point.

Access data

To start, we'll import our country shapes from our new JSON file.

`code/06-making-a-map/completed/draw-map.js`

```
5 const countryShapes = await d3.json("./../world-geojson.json")
```

Let's log our `countryShapes` to the console to look at the structure:

```
console.log(countryShapes)
```

```
draw-map.js:8
▼ {type: "FeatureCollection", name: "ne_50m_admin_0_countries", crs: {...}, features: Array(241)} ⓘ
  ► crs: {type: "name", properties: {...}}
  ► features: (241) [{}]
  ... name: "ne_50m_admin_0_countries"
  type: "FeatureCollection"
  ► __proto__: Object
```

GeoJSON data

Now we can see the structure of our GeoJSON data. It has four keys: `crs`, `features`, `name`, and `type`. All of these are metadata describing the object (eg. we're looking

at a `FeatureCollection` named "ne_50m_admin_0_countries"), except for the `features` array.

Let's expand the `features` array and look inside.

```
▼ {type: "FeatureCollection", name: "ne_50m_admin_0_countries", crs: {...}, features: Array(241)} ⓘ
  ► crs: {type: "name", properties: {...}}
  ▼ features: Array(241)
    ▼ [0 ... 99]
      ► 0: {type: "Feature", properties: {...}, geometry: {...}}
      ▼ 1:
        ► geometry: {type: "Polygon", coordinates: Array(1)}
        ► properties: {featurecla: "Admin-0 country", scalerank: 1, LABELRANK: 3, SOVEREIGNT: "Zambia", S...
          type: "Feature"
        ► __proto__: Object
      ► 2: {type: "Feature", properties: {...}, geometry: {...}}
      ► 3: {type: "Feature", properties: {...}, geometry: {...}}
      ► 4: {type: "Feature", properties: {...}, geometry: {...}}
      ► 5: {type: "Feature", properties: {...}, geometry: {...}}
      ► 6: {type: "Feature", properties: {...}, geometry: {...}}
      ▷ <...>
```

GeoJSON data, features

Each feature has a `geometry` object and a `properties` object. The `properties` object contains information about the feature — we can see that each of these features represents one country, based on the information in here.

```
▼ {type: "FeatureCollection", name: "ne_50m_admin_0_countries", crs: {...}, features: Array(241)} ⓘ
  ► crs: {type: "name", properties: {...}}
  ▼ features: Array(241)
    ▼ [0 ... 99]
      ► 0: {type: "Feature", properties: {...}, geometry: {...}}
      ▼ 1:
        ► geometry: {type: "Polygon", coordinates: Array(1)}
        ▼ properties:
          ABBREV: "Zambia"
          ABBREV_LEN: 6
          ADM0_A3: "ZMB"
          ADM0_A3_IS: "ZMB"
          ADM0_A3_UN: -99
          ADM0_A3_US: "ZMB"
          ADM0_A3_WB: -99
          ADM0_DIF: 0
          ADMIN: "Zambia"
          BRK_A3: "ZMB"
          BRK_DIFF: 0
          BRK_GROUP: null
          BRK_NAME: "Zambia"
          CONTINENT: "Africa"
          ECONOMY: "7. Least developed region"
          FIPS_10_: "ZA"
          FORMAL_EN: "Republic of Zambia"
          FORMAL_FR: null
```

GeoJSON data, geometry

For example, this feature has a NAME of "Zambia". If we dig into the geometry of this feature, we'll find an array of [latitude, longitude] coordinates.

```
▼ {type: "FeatureCollection", name: "ne_50m_admin_0_countries", crs: {...}, features: Array(241)} ⓘ
  ► crs: {type: "name", properties: {...}}
  ▼ features: Array(241)
    ▼ [0 ... 99]
      ► 0: {type: "Feature", properties: {...}, geometry: {...}}
      ▼ 1:
        ▼ geometry:
          ▼ coordinates: Array(1)
            ▼ 0: Array(377)
              ▼ [0 ... 99]
                ► 0: (2) [30.39609375, -15.64306640625]
                ► 1: (2) [30.25068359375001, -15.643457031250009]
                ► 2: (2) [29.994921875000017, -15.64404296875]
                ► 3: (2) [29.729589843750006, -15.644628906250006]
                ► 4: (2) [29.4873046875, -15.69677734375]
                ► 5: (2) [29.287890625000017, -15.776464843750006]
                ► 6: (2) [29.050585937500017, -15.901171875]
                ► 7: (2) [28.973046875000023, -15.950097656250009]
                ► 8: (2) [28.9130859375, -15.98779296875]
```

GeoJSON data, geometry

Latitude and **longitude** are part of a common geographic coordinate system:

latitude represents the North-South position of a point on Earth as the angle above (up to 90°) or below (down to -90°) the equator (0°).

longitude represents the East-West position of a point on Earth as an angle between -180° and 180°.

Fun fact: historically, geographers used a nation's capital as the 0° reference for **longitude**. In 1884, the International Meridian Conference decided on the universal **longitude** that we use today.

Next, we want to create our accessor functions. We'll need to access our country ID to find the metric value in our population growth dataset (which we'll look at later in this chapter). We'll also want our hovered country's name to display in a tooltip.

We can look in a sample `feature` to find the relevant keys.

code/06-making-a-map/completed/draw-map.js

```
8  const countryNameAccessor = d => d.properties["NAME"]  
9  const countryIdAccessor = d => d.properties["ADM0_A3_IS"]
```

Our dataset

Let's dig in to the code! We'll be working in the `/code/06-making-a-map/draft/` folder, starting with the `draw-map.js` file.

Let's grab our dataset that shows population growth — it's located in the `/code/06-making-a-map/data_bank_data.csv` file.

This data was pulled from [The World Bank](#)^a— feel free to download a more recent dataset if you'd like. The population growth metric is the percent change of total population in 2017.

^a<https://databank.worldbank.org/data/source/world-development-indicators#>

Note that this is a CSV (comma-separated values) instead of a JSON file. No worries! This is just a different format of storing information — it ends up being much smaller than a JSON file because it only mentions each data point's properties once, similar to a table's header. Open the file up and take a peek!

To load data from a CSV file, we can make a simple tweak. Instead of using `d3.json()`, we can use `d3.csv()`.

code/06-making-a-map/completed/draw-map.js

```
6  const dataset = await d3.csv("./../data_bank_data.csv")
```

Put this line of code on line 4, right after we load `countryShapes` to keep the file loading in one place.

Easy peasy! Let's log the result to the console and take a look (`console.log(dataset)`).

```
draw-map.js:5
(1061) [{}]
  ▼ [0 ... 99]
    ▼ 0:
      2017 [YR2017]: "19000000"
      Country Code: "AFG"
      Country Name: "Afghanistan"
      Series Code: "ST.INT.RCPT.CD"
      Series Name: "International tourism, receipts (current US$)"
      ▶ __proto__: Object
    ▶ 1: {Country Name: "Afghanistan", Country Code: "AFG", Series Name: "Net migration", Series Code: "SM..."}
    ▶ 2: {Country Name: "Afghanistan", Country Code: "AFG", Series Name: "Population growth (annual %)", ...}
    ▶ 3: {Country Name: "Afghanistan", Country Code: "AFG", Series Name: "Population density (people per ..."}
    ▶ 4: {Country Name: "Albania", Country Code: "ALB", Series Name: "International tourism, receipts (cu..."}
    ▶ 5: {Country Name: "Albania", Country Code: "ALB", Series Name: "Net migration", Series Code: "SM.PO..."}
    ▶ 6: {Country Name: "Albania", Country Code: "ALB", Series Name: "Population growth (annual %)", Seri...}
    ▶ 7: {Country Name: "Albania", Country Code: "ALB", Series Name: "Population density (people per sq ..."
      dataset
```

This data structure is a bit messy — a great chance to practice with a more real-world scenario! We can see that our `dataset` is an array which lists each country multiple times, each time with a different metric, named under the `Series Name` key.

We're only interested in the "Population growth (annual %)" metric. Let's define that in a variable name.

`code/06-making-a-map/completed/draw-map.js`

```
10  const metric = "Population growth (annual %)"
```

Later on, once you're more comfortable, play around with changing this metric value and visualize the other metrics.

We'll want an easy way to look up the population growth number using a country name. We could find a matching item in our `dataset` array, but that could be expensive. Instead, let's create an object with **country ids** as **keys** and the population growth amount as **values**:

```
{  
  AFG: 2.49078956147291,  
  ALB: -0.0919722937442495,  
  // ...  
}
```

To start, we'll initialize an empty object:

`code/06-making-a-map/completed/draw-map.js`

```
14  let metricDataByCountry = {}
```

Then we'll go over each item in our `dataset` array. If the item's "Series Name" doesn't match our `metric`, we won't do anything. If it does match, we'll add a new **value** to our `metricDataByCountry` object: the **key** is the item's "Country Code" and the **value** is the item's "2017 [YR2017]" number.

Note that these values are stored as **Strings** — we'll convert the **value** to a number by prepending `+`, and default to `0` if the **value** doesn't exist.

`code/06-making-a-map/completed/draw-map.js`

```
15  dataset.forEach(d => {  
16    if (d["Series Name"] != metric) return  
17    metricDataByCountry[d["Country Code"]] = +d["2017 [YR2017]"] || 0  
18  })
```

That's a messy operation, isn't it? This is part of the reason to deal with the data at the top of our chart-making code. We'll polish our dataset into a cleaner format, separating handling our data idiosyncrasies from actually drawing the chart.

Create chart dimensions

Let's move on! Next up, we'll define our chart dimensions. We know that we want our chart to be 90% as wide as our window and we want a small margin on every side of our map.

Let's hold off on setting the height for now.

code/06-making-a-map/completed/draw-map.js

```
22 let dimensions = {  
23   width: window.innerWidth * 0.9,  
24   margin: {  
25     top: 10,  
26     right: 10,  
27     bottom: 10,  
28     left: 10,  
29   },  
30 }  
31 dimensions.boundedWidth = dimensions.width  
32   - dimensions.margin.left  
33   - dimensions.margin.right
```

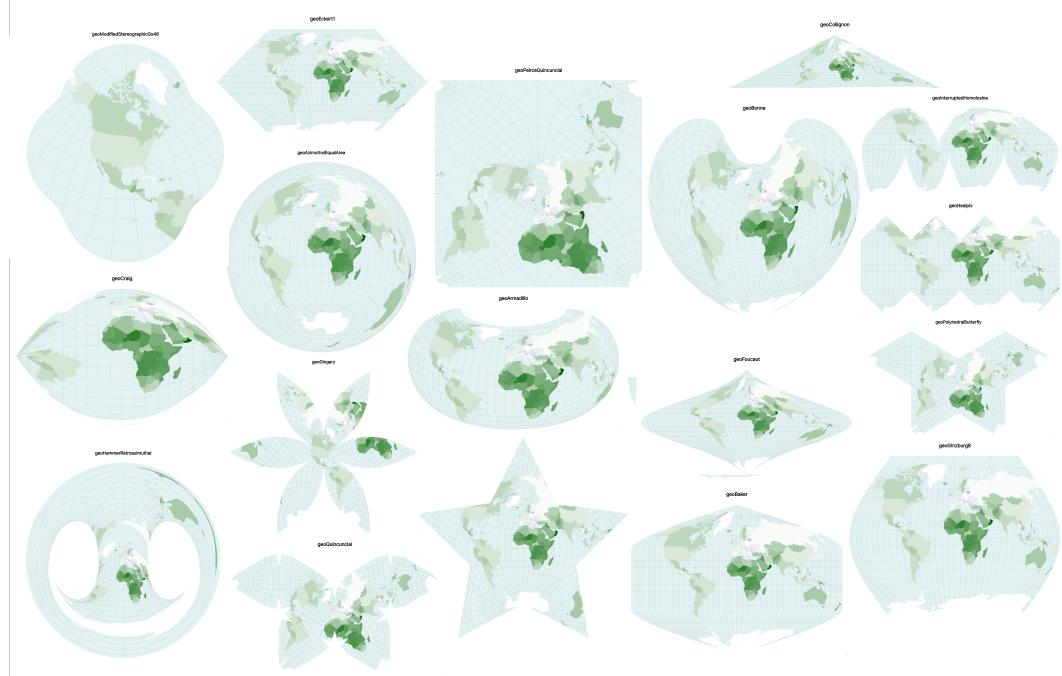
How tall does our map need to be? That will depend on our map's **projection**.

What is a projection?

Despite what flat-earthers might say, the Earth is a sphere. When representing it on a 2D screen, we'll need to create some rules — these rules are the **projection**.

Imagine peeling an orange and turning the skin into a flat sheet — even with slicing it in various places, it's impossible to do perfectly. **Projections** will use a combination of distortion (stretching parts of the map) and slicing to approximate the Earth's actual shape.

There are *many* projections out there. **d3-geo**⁴³ has 15 projections built in, but many more in a d3 module that's not included in the core build: **d3-geo-projection**⁴⁴. Play around with the different options at [http://localhost:8080/06-making-a-map/-completed-projections](http://localhost:8080/06-making-a-map/completed-projections)⁴⁵.



projections, galore!

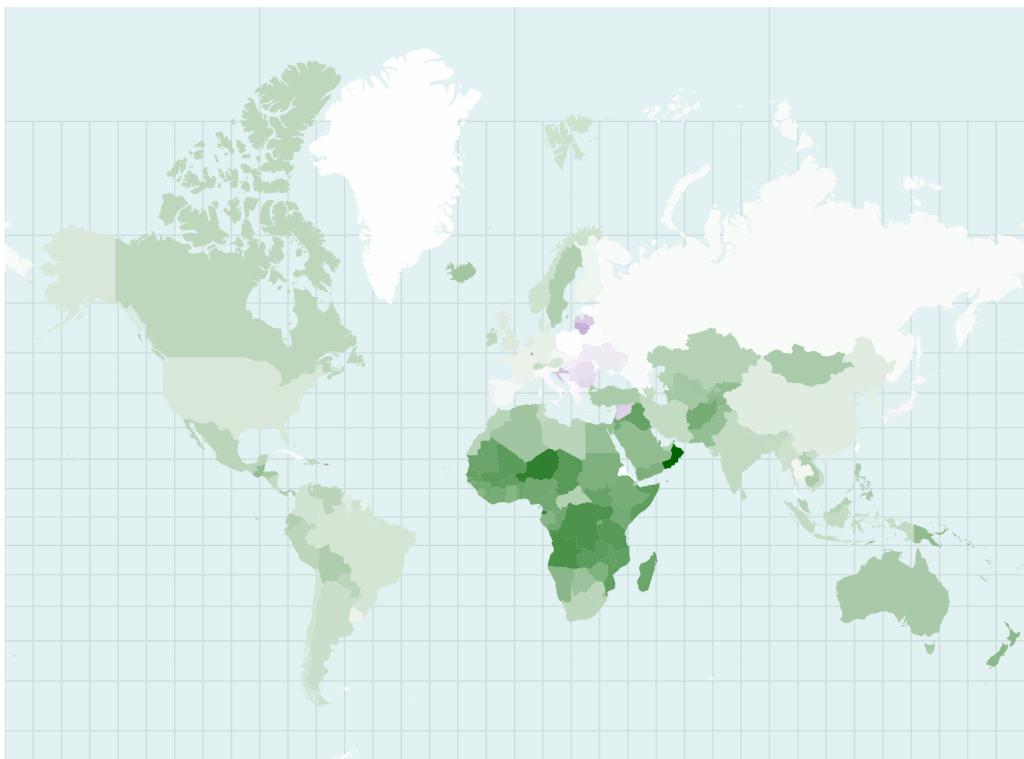
Which projection should I use?

The projection you might be most used to is the **Mercator** projection.

⁴³<https://github.com/d3/d3-geo>

⁴⁴<https://github.com/d3/d3-geo-projection/>

⁴⁵<http://localhost:8080/06-making-a-map/completed-projections>

geoMercator**Mercator projection**

Mercator has been around for a long time — it was created in 1569, before Antarctica had even been discovered! Its parallel lines preserve true angles, which made it easy for sailors to use for navigation, but it sacrifices vertical distortion for horizontal distortion.

You can see this distortion in the **graticule** (grid lines on the map) — see how tall the “squares” get near the top of the map. Also note how the country sizes get skewed as they approach the poles — Greenland is depicted as the same as Africa!

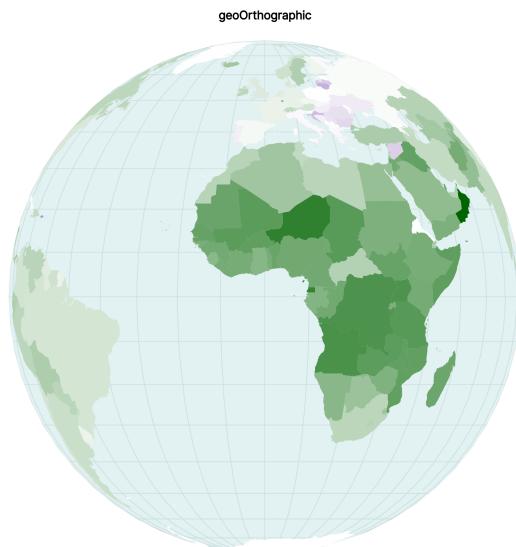
Another option is the **Winkel-Tripel** projection, which attempts to balance the three types of distortion: area, direction, and distance.



Winkel Tripel projection

Land near the North and South poles is still enlarged, but the country shapes and sizes are more accurate.

If we want to show a globe, we can use the `d3.geoOrthographic()` projection.



Orthographic projection

Note that the type of projection matters more with maps that cover more area. The more we're zoomed in, the less distortion we'll need to flatten our round Earth shape. For example, a close-up of a city will look basically the same in two different projections.

There really isn't a "right" map projection to use — they all have to distort *something*, mapping a 3D shape into 2D. As a general rule of thumb, a variant of the **Mercator**, the **Transverse Mercator** (`d3.geoTransverseMercator()`) is a good bet for showing maps that cover one country or smaller. The **Winkel Tripel** (`d3.geoWinkel3()`) or **Equal Earth** (`d3.geoEqualEarth()`) are good bets for maps covering larger areas, such as the whole world. But this is really the tip of the iceberg - if you're interested, read more about [how projections work⁴⁶](#) and [specific projections⁴⁷](#).

Finishing creating our chart dimensions

How does this information help us decide our chart height? We're going to plot the whole Earth to cover the full bounded width, then measure the height.

⁴⁶https://en.wikipedia.org/wiki/Map_projection

⁴⁷https://en.wikipedia.org/wiki/List_of_map_projections

If you remember from before, a GeoJSON object can contain features of the following types: Point, MultiPoint, LineString, MultiLineString, Polygon, MultiPolygon, GeometryCollection, Feature, or FeatureCollection. But none of these cover the whole Earth! Worry not, [d3-geo⁴⁸](#) adds support for a type of "Sphere", which will cover the whole globe.

Let's define our globe using this mock-GeoJSON format.

code/06-making-a-map/completed/draw-map.js

35 `const sphere = ({type: "Sphere"})`

Each of the **projections** mentioned in the previous section (and more) are implemented in either **d3-geo** or **d3-geo-projection**. We can use these **projection functions** to convert from [longitude, latitude] coordinates to [x, y] pixel coordinates. Essentially, a **projection function** is our scale in the geographic world.

Let's create our **projection function**. We'll use `d3.geoEqualEarth()` — feel free to play around with other options once we're finished drawing our map.

code/06-making-a-map/completed/draw-map.js

36 `const projection = d3.geoEqualEarth()`

Each **projection function** has its own default size (think: **range**). But we want our projection to be the same width as our **bounds**.

To update our projection's width, we can use its `.fitWidth()` method, which takes two parameters:

1. the width in pixels
2. a GeoJSON object

When we call this method, our **projection** will update its size so that the GeoJSON object (2) we pass will be the specified width (1).

⁴⁸https://github.com/d3/d3-geo#_path

code/06-making-a-map/completed/draw-map.js

```
36 const projection = d3.geoEqualEarth()
37   .fitWidth(dimensions.boundedWidth, sphere)
```

Okay great, but how do we get the **height** of our **sphere**?

d3.geoPath()⁴⁹ is similar to the line generator (d3.line()) we used for our timeline in Chapter 1. When we pass it our projection, it will create a generator function that will help us create our geographical shapes.

code/06-making-a-map/completed/draw-map.js

```
39 const pathGenerator = d3.geoPath(projection)
```

Let's test it out, what happens when we pass it our **sphere**:

`console.log(pathGenerator(sphere))`

```
M137.6850612805441,2.842170943040401e-14L337.85,2.842170943040401e-draw-map.js:38
14L538.0149387194559,2.842170943040401e-
14L573.2969871814703,17.046318574021143L601.7388721244954,35.11582806465225L627.4033152905058,57.
060601880579455L648,1636120831404,81,74126598217812L656.4778194202438,94.83731333047876L663.34592
97549905,108.31468205596332L668.7284340918884,122.09019037238977L672.5945365846244,136.0854555836
9653L674.922602578265,150.22546379143057L675.7,164.4373301346346L674.922602578265,178.64919647783
864L672.5945365846244,192.78920468557268L668.7284340918884,206.78446989687944L663.3459297549905,2
20.5599782133059L656.4778194202438,234.03734693879045L648.1636120831404,247.1333942870911L627.403
3152905058,271.8140583886898L601.7388721244954,293.75883220461697L573.2969871814703,311.828341695
24807L538.0149387194559,328.87466026926916L337.85,328.87466026926916L137.6850612805441,328.874660
26926916L102.40301281852976,311.82834169524807L73.96112787550464,293.75883220461697L48.2966847094
94286,271.8140583886898L27.536387916859724,247.1333942870911L19.222180579756184,234.0373469387904
5L12.35407024500961,220.5599782133059L6.9715659081117,206.78446989687944L3.1054634153755956,192.7
8920468557268L0.7773974217350315,178.64919647783864L5.684341886080802e-
14,164.4373301346346L0.7773974217350315,150.22546379143057L3.1054634153755956,136.08545558369653L
6.9715659081117,122.09019037238977L12.35407024500961,108.31468205596332L19.222180579756184,94.837
31333047876L27.536387916859724,81.74126598217812L48.296684709494286,57.060601880579455L73.9611278
7550464,35.11582806465225L102.40301281852976,17.046318574021143L137.6850612805441,2.8421709430404
01e-14Z
```

pathGenerator result

That looks familiar! Perfect, we got back a `<path> d` string. But how do we find out how **tall** that path would be?

Thankfully, our `pathGenerator()` has a `.bounds()` method that will return an array of `[x, y]` coordinates describing a bounding box of a specified GeoJSON object. Let's test that by logging the bounding box for our **sphere**:

⁴⁹<https://github.com/d3/d3-geo/blob/master/README.md#geoPath>

```
console.log(pathGenerator.bounds(sphere))

▼ (2) [Array(2), Array(2)] ⓘ
▶ 0: (2) [0, 0]
▶ 1: (2) [831.4, 404.656493337088]

sphere bounds
```

Great! When drawn on our map, the whole Earth will span from 0px to 831.4px horizontally and from 0px to 404.7px vertically. We can use [ES6 array destructuring⁵⁰](#) to grab the y1 value.

[code/06-making-a-map/completed/draw-map.js](#)

```
39 const pathGenerator = d3.geoPath(projection)
40 const [[x0, y0], [x1, y1]] = pathGenerator.bounds(sphere)
```

We want the entire Earth to fit within our **bounds**, so we'll want to set our `boundedHeight` to *just* cover our `sphere`.

[code/06-making-a-map/completed/draw-map.js](#)

```
42 dimensions.boundedHeight = y1
43 dimensions.height = dimensions.boundedHeight
44   + dimensions.margin.top
45   + dimensions.margin.bottom
```

Draw canvas

Now we're back in familiar territory! Let's create our **wrapper** and **bounds** elements. See if you can do this step from memory.

First, we want to add a new `<svg>` element (our **wrapper**) to the existing `<div>` with an `id` of "wrapper". Then we want to add a `<g>` element (our **bounds**) and shift it to respect our `top` and `left` margins.

⁵⁰https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Operators/Destructuring_assignment#Array_destructuring

code/06-making-a-map/completed/draw-map.js

```
49 const wrapper = d3.select("#wrapper")
  .append("svg")
    .attr("width", dimensions.width)
    .attr("height", dimensions.height)
53
54 const bounds = wrapper.append("g")
  .style("transform", `translate(${{
55   dimensions.margin.left
56     }px, ${{
57       dimensions.margin.top
58     }px})`)
```

Create scales

Next up, we need to create our scales. Typically, we would create x and y scales, but those are covered by our **projection**. We'll only need a scale to help us turn our **metric values** (population growth amounts) into **color values**.

Remember our object of **country id** and **population growth values**? We can grab all of the population growth values by using `Object.values()`, which returns an object's values in an array. More info [here⁵¹](#), for the curious.

code/06-making-a-map/completed/draw-map.js

```
63 const metricValues = Object.values(metricDataByCountry)
```

Then we can extract the smallest and largest values by using `d3.extent()`.

⁵¹https://developer.mozilla.org/en-US/docs/Web/JavaScript/Reference/Global_Objects/Object/values

code/06-making-a-map/completed/draw-map.js

64 **const** metricValueExtent = d3.extent(metricValues)

Let's take a look at the range we're dealing with with `console.log(metricValueExtent)`

► (2) [-2.05660023263167, 4.669194641437]

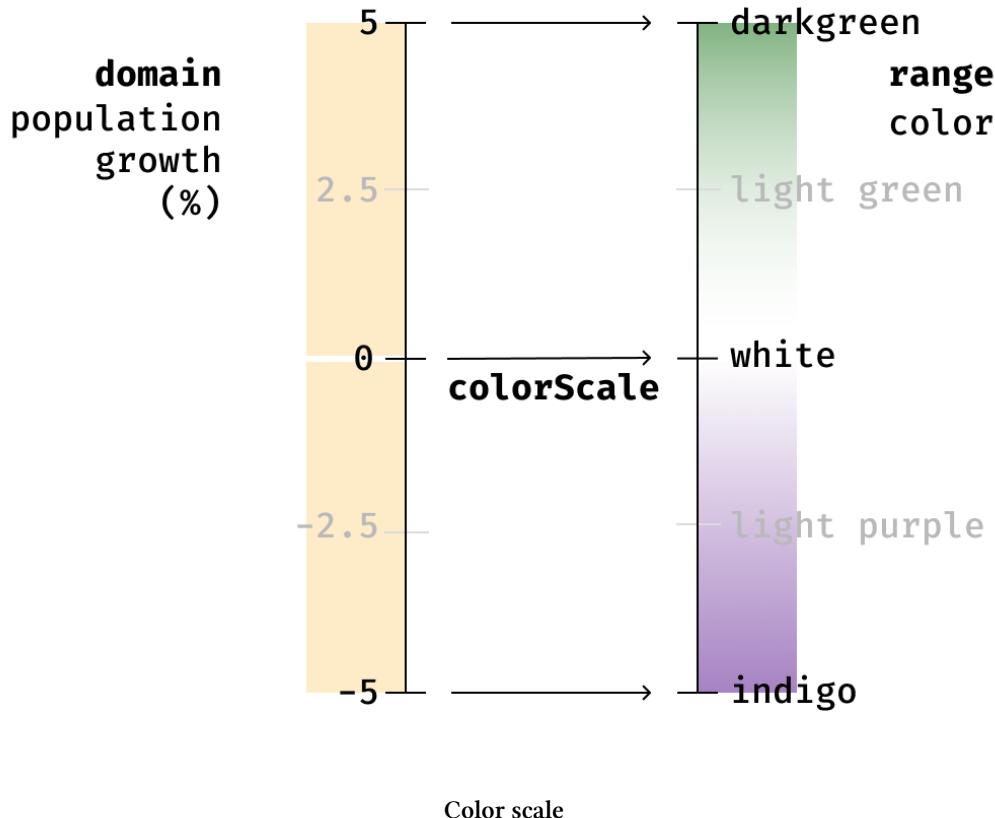
Metric values extent

Aha! Our extent starts below zero, meaning that some countries had a *negative* population growth in 2017. Let's represent population declines on a **red** color scale and population growths on a **green** color scale.

This kind of color scale is called a **diverging** color scale, which we'll cover in more detail in **Chapter 7**.

So far, we've only create scales that have one *minimum* and one *maximum* value for both their **domain** and **range**. But we can create *piecewise* scales, which are basically multiple scales in one. If we use a **domain** and **range** with more than two values, d3 will create a scale with more than two "anchors".

In this case, we want to create a scale with a "middle" population growth of 0% that converts to a middle color of **white**. Population growth amounts above 0 should be converted with a color scale from **white** (0) to **green** (5) and population growth amounts below 0 should be converted with a color scale from **red** (-2) to **white** (0).



Creating a *piecewise* scale like this is easier than it might seem: just add a middle value to both the **domain** and the **range**.

[code/06-making-a-map/completed/draw-map.js](#)

```

65  const maxChange = d3.max([-metricValueExtent[0], metricValueExtent[1]]\)
66  ])
67  const colorScale = d3.scaleLinear()
68  .domain([-maxChange, 0, maxChange])
69  .range(["indigo", "white", "darkgreen"])

```

We have a choice here:

1. create a scale which scales evenly on both sides (eg. `.domain([-5, 0, 5])`)

This approach will let the viewer see if countries are shrinking at the same pace as countries that are growing, since a light red (-2.5%) will correlate to a light green (2.5%).

2. create a scale which scales unevenly from white to red and white to green (eg. `.domain([-2, 0, 5])`)

This approach will maximize the color scale, making distinctions between countries with different growth rates clear. However, it can be confusing when a light red color (-0.1%) doesn't match the scale of a light green color (2.5%).

We'll go with approach number 1 here, to keep things simple, but play around with the two approaches yourself to get a feel for the differences! How might the legend (coming up) look different with approach number 2?

Why did we choose **indigo** and **darkgreen** as our colors? A more semantic mapping to “negative” and “positive” might be **red** and **green**, but the most common form of color blindness is red-green color blindness, and we want our readers to be able to see differences in our color scale. We'll talk more about this in **Chapter 7**.

Draw data

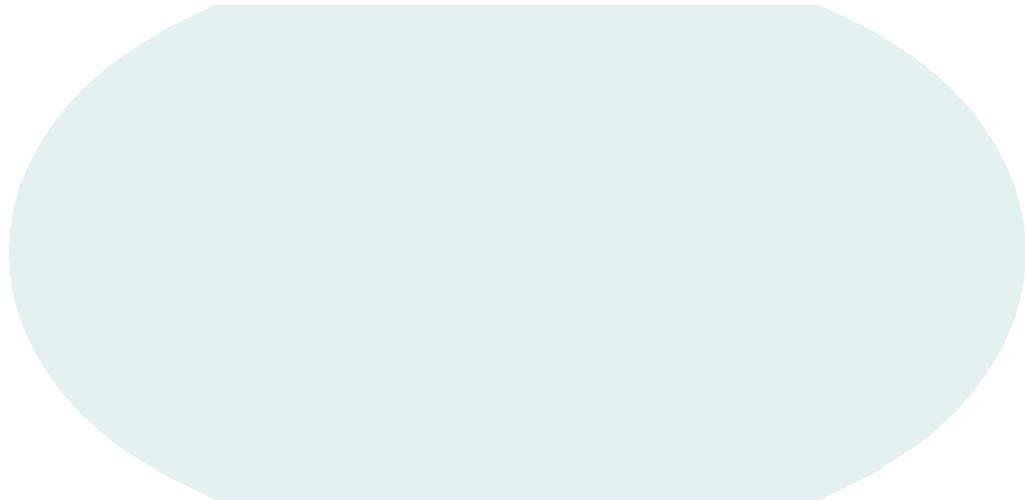
Here comes the action! To start, we'll draw our Earth outline. It makes sense to draw this first, since SVG elements layer based on their order in the DOM, and we'll want all other elements to cover this outline.

Remember that our **pathGenerator** is aware of the **projection** we're using and will act as a **scale**, converting a GeoJSON object into a `<path> d` string.

code/06-making-a-map/completed/draw-map.js

```
72 const earth = bounds.append("path")
73   .attr("class", "earth")
74   .attr("d", pathGenerator(sphere))
```

Great! Now we can see the outline of our earth.



Earth outline

Note that we're styling our elements in the `/code/06-making-a-map/styles.css` file, using their class names. To see the elements without the styling, remove the `class` attribute.

Next, let's display a **graticule** on our map. A **graticule** is a grid of latitudinal and longitudinal lines — essentially tick marks for a map. These are helpful for aligning map elements that are far apart, and also for conveying how the projection is distorting the globe.

Thankfully, we can easily create a GeoJSON graticule — `d3.geoGraticule10()` will generate a classic graticule with lines every 10 degrees.

code/06-making-a-map/completed/draw-map.js

```
76 const graticuleJson = d3.geoGraticule10()
```

There are many ways to configure this GeoJSON graticule, check out the options [in the d3-geo docs^a](#).

^a<https://github.com/d3/d3-geo#geoGraticule>

If we log this graticule object to the console (`console.log(graticuleJson)`), we can see that it's a GeoJSON object of type "MultiLineString".

```
▼ {type: "MultiLineString", coordinates: Array(53)} ⓘ
  ► coordinates: (53) [Array(3), Array(3), Array(3), Array(3), Array(145), Array(3), Ar...
    type: "MultiLineString"
  ► __proto__: Object
```

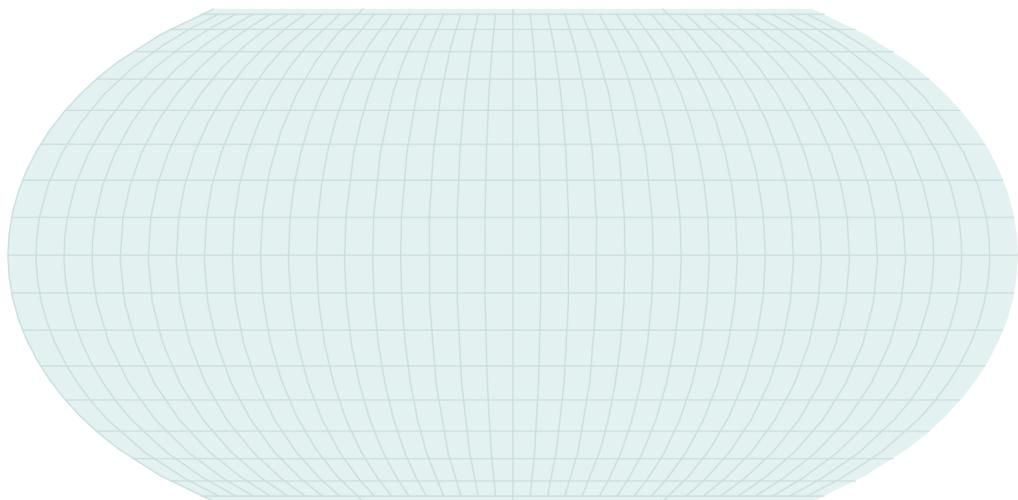
graticule json

Thankfully, our `pathGenerator()` knows how to handle any GeoJSON type — let's use it to draw our graticules.

code/06-making-a-map/completed/draw-map.js

```
77 const graticule = bounds.append("path")
  .attr("class", "graticule")
  .attr("d", pathGenerator(graticuleJson))
```

Wonderful!



globe with graticule

Finally, let's draw our countries. We'll use the **data join** method we used in Chapter 2. First, we'll select all of the elements with a class of `.country`. Remember: even though these elements don't yet exist, this is priming our **d3 selection object** to bind data to similar elements.

code/06-making-a-map/completed/draw-map.js

```
81 const countries = bounds.selectAll(".country")
  .data(countryShapes.features)
```

Next, we'll bind our data to our selection object. Remember what our `countryShapes` GeoJSON object looks like?

```
▼ {type: "FeatureCollection", name: "ne_50m_admin_0_countries", crs: {...}, features: Array(241)} ⓘ
  ► crs: {type: "name", properties: {...}}
  ▼ features: Array(241)
    ▼ [0 ... 99]
      ► 0: {type: "Feature", properties: {...}, geometry: {...}}
      ▼ 1:
        ► geometry: {type: "Polygon", coordinates: Array(1)}
        ► properties: {featurecla: "Admin-0 country", scalerank: 1, LABELRANK: 3, SOVEREIGNT: "Zambia", S...
          type: "Feature"
        ► __proto__: Object
      ► 2: {type: "Feature", properties: {...}, geometry: {...}}
      ► 3: {type: "Feature", properties: {...}, geometry: {...}}
      ► 4: {type: "Feature", properties: {...}, geometry: {...}}
      ► 5: {type: "Feature", properties: {...}, geometry: {...}}
      ► 6: {type: "Feature", properties: {...}, geometry: {...}}
      ▷ ...
```

GeoJSON data, features

Because d3 will create **one element per item in the dataset we pass to `.data()`**, we'll want to use the list of `features` instead of the whole object.

code/06-making-a-map/completed/draw-map.js

```
81 const countries = bounds.selectAll(".country")
  .data(countryShapes.features)
  .enter().append("path")
```

Next, we'll use `.enter().append("path")` to tell d3 to create one new `<path>` for each item in our list of countries.

For a reminder of how `.enter().append()` works, refer back to **Chapter 3**.

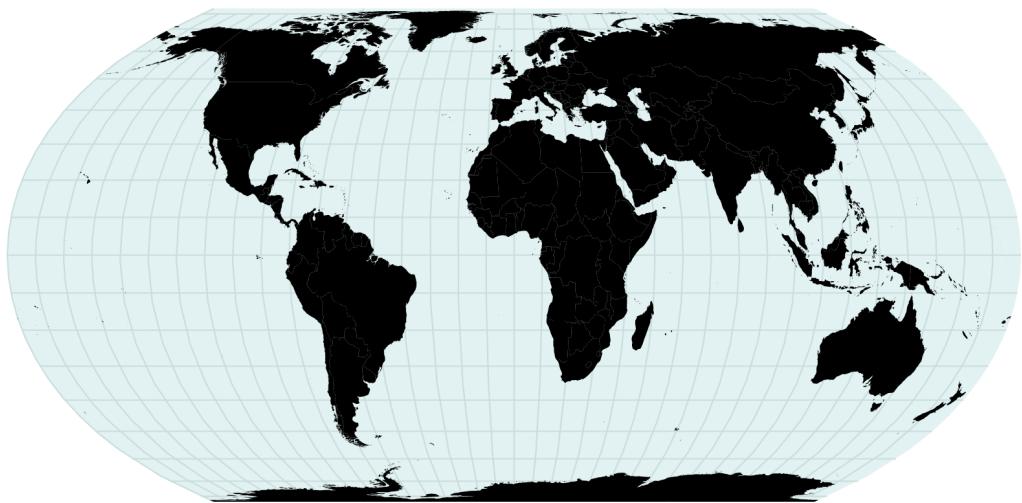
We'll give each of these `<path>`s a class of "country" and then pass it straight to our `pathGenerator()` to get a `d` string, generating the shape of each country.

code/06-making-a-map/completed/draw-map.js

```
81 const countries = bounds.selectAll(".country")
82   .data(countryShapes.features)
83   .enter().append("path")
84     .attr("class", "country")
85     .attr("d", pathGenerator)
86     .attr("fill", d => {
```

Remember that `.attr("d", pathGenerator)` is the same as `.attr("d", d => pathGenerator(d))`.

If we look at our browser now, we'll see all of our countries!



globe with countries

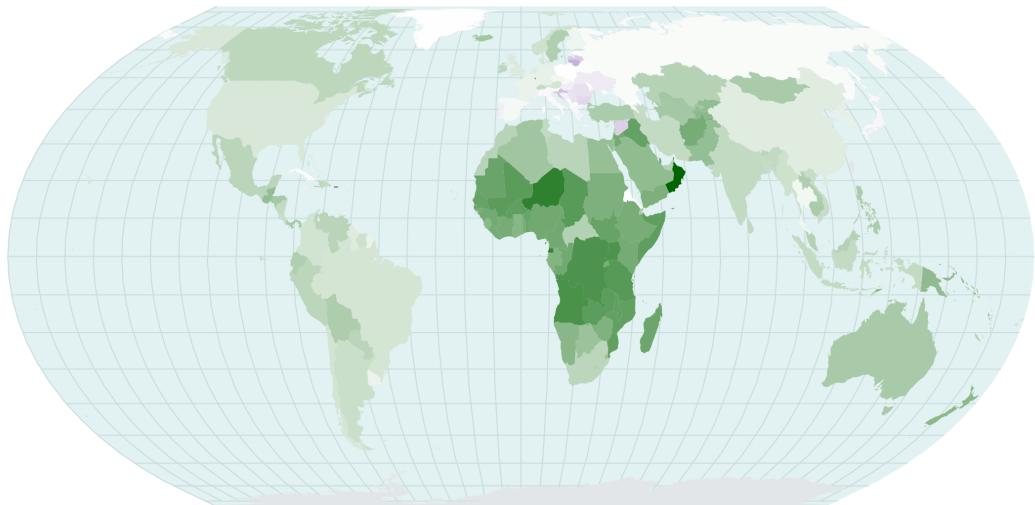
We have one last task — we want to color our countries based on their population growth amount. We'll need to set their fill by looking up the country's id in the `metricDataByCountry` object we created at the top of our file, then passing that id to our `colorScale()`.

Since some countries might be missing from our dataset, we'll want to indicate that on our map by coloring them white.

code/06-making-a-map/completed/draw-map.js

```
87 const metricValue = metricDataByCountry[countryIdAccessor(d)]
88 if (typeof metricValue == "undefined") return "#e2e6e9"
89 return colorScale(metricValue)
```

Perfect! Now we can see our map in full color!



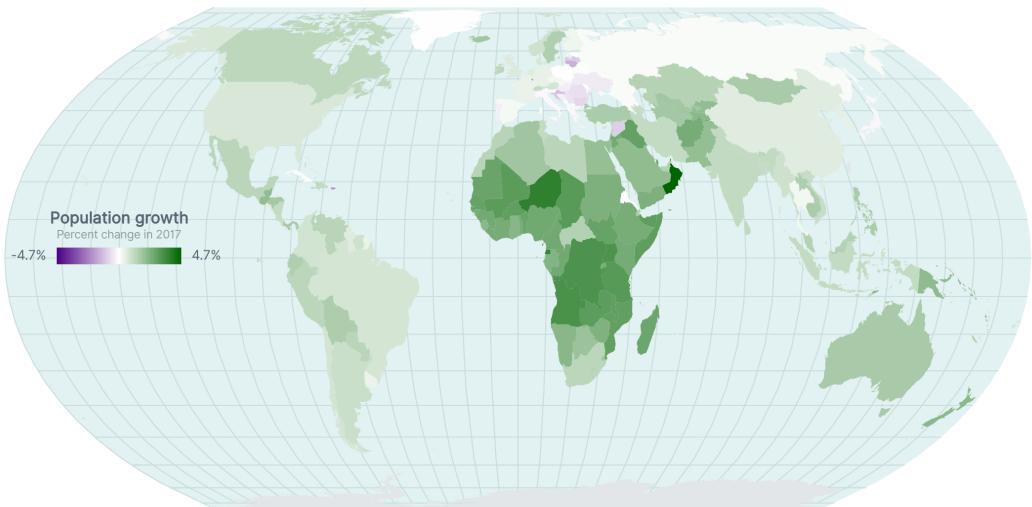
globe with colored countries

It's clear now that the highest growth countries are in Africa and the Middle East, and that a cluster of European countries shrunk in 2017. Try to find a few other insights that are easy to see now that we've visualized this data with a map.

Draw peripherals

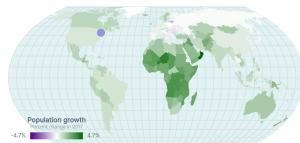
Drawing a legend

Data visualizations should be able to stand on their own, without accompanying text. Let's create a legend to explain our color scale.

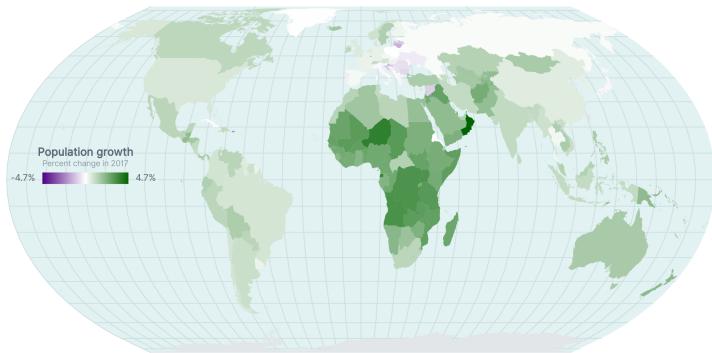


globe with legend

To start, we'll create a `<g>` element and position it on the left side of our map. To get a taste of making our map work at different screen sizes, we'll position it differently based on the screen size. On a skinnier screen, let's put it at the bottom of the map, and on larger screens, we can place it halfway down the map (to the left of the Americas).



globe with legend, small



globe with legend, large

code/06-making-a-map/completed/draw-map.js

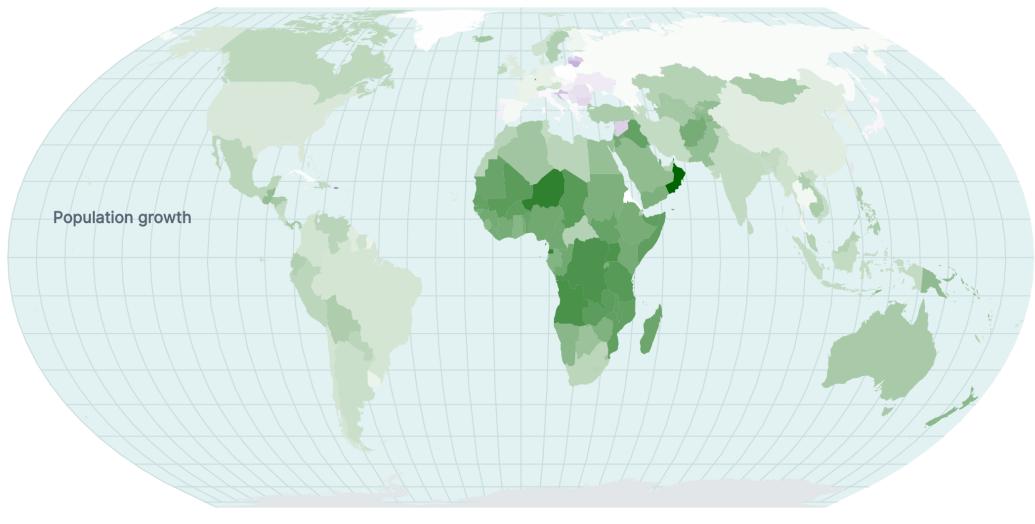
```
94 const legendGroup = wrapper.append("g")
95   .attr("transform", `translate(${ 
96     120
97   }, ${ 
98     dimensions.width < 800
99       ? dimensions.boundedHeight - 30
100      : dimensions.boundedHeight * 0.5
101    })`)
```

To keep things simple, let's align the top of our `<g>` element to the top of the gradient bar. When we add our legend title, we'll move it up 23 pixels to give the bar some space.

code/06-making-a-map/completed/draw-map.js

```
103 const legendTitle = legendGroup.append("text")
104   .attr("y", -23)
105   .attr("class", "legend-title")
106   .text("Population growth")
```

Great! Now we can see where our legend group is positioned.



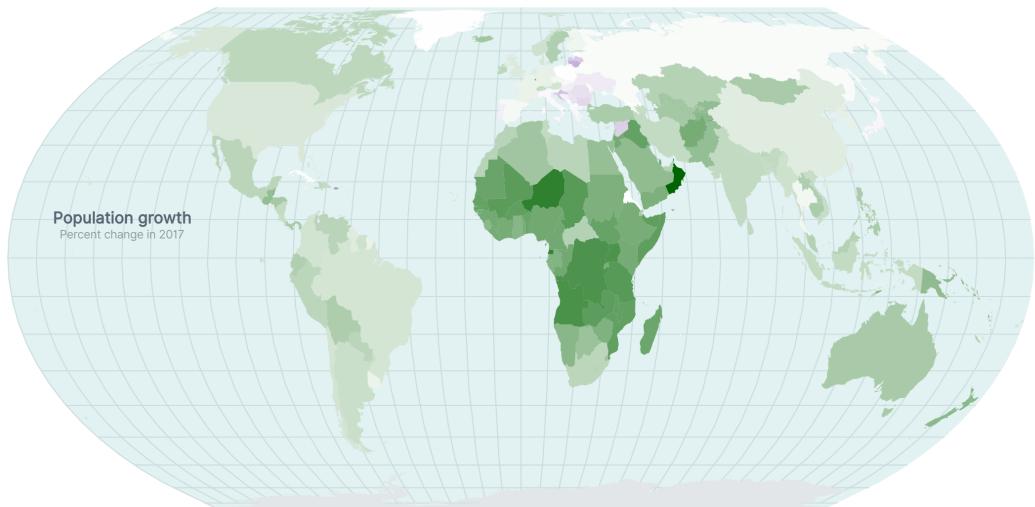
globe with legend title

Next, let's add a byline to give the reader more context.

code/06-making-a-map/completed/draw-map.js

```
108 const legendByline = legendGroup.append("text")
109   .attr("y", -9)
110   .attr("class", "legend-byline")
111   .text("Percent change in 2017")
```

Note that we're adding classes that match up with existing CSS styles — feel free to check out `styles.css` to investigate. We're using the CSS property `text-anchor: middle` to horizontally center our `<text>`.



globe with legend byline

Next up, we'll create the bar showing the colors in our scale.



legend gradient bar

Remember when we created a `<defs>` SVG element for our `<clipPath>` in Chapter 4? Elements created within `<defs>` won't be visible by themselves, but we can use them later. Let's create a `<defs>` element to store a gradient.

[code/06-making-a-map/completed/draw-map.js](#)

113

```
const defs = wrapper.append("defs")
```

Since we'll be creating a bar here and referencing it in another place, let's define a variable to hold the `id` of our gradient.

code/06-making-a-map/completed/draw-map.js

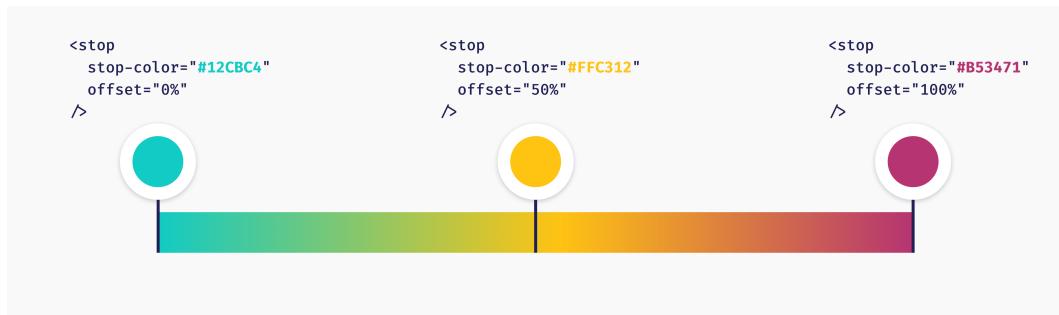
114 **const legendGradientId = "legend-gradient"**

Time to create our gradient! To make a gradient in SVG, we'll need to create a `<linearGradient>`⁵² SVG element. Within that, we'll create several `<stop>` SVG elements that will tell the gradient what colors to interpolate between, using `stop-color` and `offset` attributes.

For example, this HTML code:

```
<linearGradient>
  <stop stop-color="#12CBC4" offset="0%"></stop>
  <stop stop-color="#FFC312" offset="50%"></stop>
  <stop stop-color="#B53471" offset="100%"></stop>
</linearGradient>
```

will create this gradient:



Gradient example

The `offset`'s percentage value is *in proportion to the element using the `<linearGradient>`*.

Let's try it out!

⁵²<https://developer.mozilla.org/en-US/docs/Web/SVG/Element/linearGradient>

We'll set the `id` attribute to our `legendGradientId` so that we can reference our gradient later.

code/06-making-a-map/completed/draw-map.js

```
115 const gradient = defs.append("linearGradient")
116   .attr("id", legendGradientId)
```

To add our stops, we'll use `colorScale.range()` to get an array of the colors in our color scale. Then we'll use `.enter().append()` to create one stop per color. We'll use the color's index to calculate the percent offset (from 0% to 100%).

code/06-making-a-map/completed/draw-map.js

```
115 const gradient = defs.append("linearGradient")
116   .attr("id", legendGradientId)
117   .selectAll("stop")
118   .data(colorScale.range())
119   .enter().append("stop")
120   .attr("stop-color", d => d)
121   .attr("offset", (d, i) => `${
122     i * 100 / 2 // 2 is one less than our array's length
123   }%`)
```

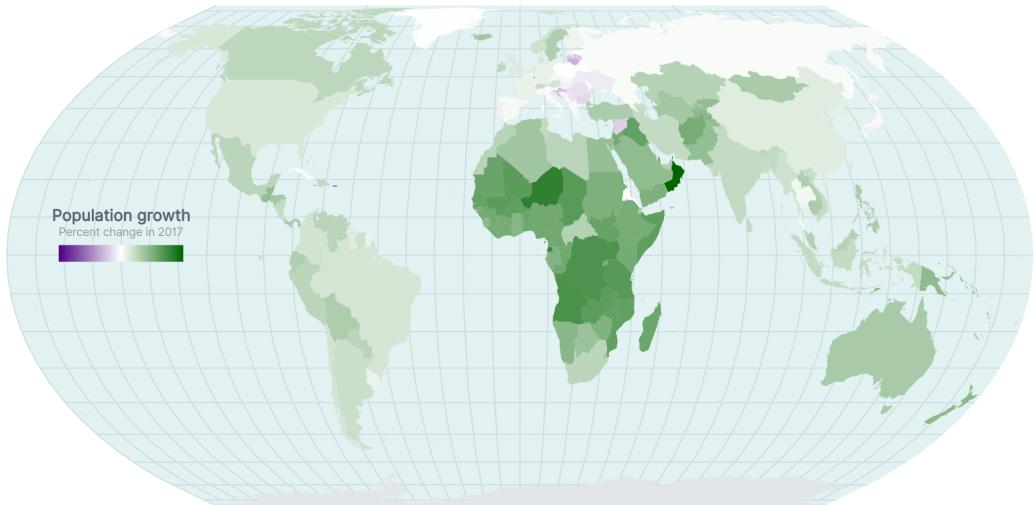
To use a `<linearGradient>`, all we need to do is set the `fill` or `stroke` of a SVG element to `url(#GRADIENT_ID)` (where `GRADIENT_ID` matches the `<linearGradient>`'s `id` attribute).

Let's create a `<rect>` that's centered horizontally and is filled with our gradient's id.

code/06-making-a-map/completed/draw-map.js

```
125 const legendWidth = 120
126 const legendHeight = 16
127 const legendGradient = legendGroup.append("rect")
128     .attr("x", -legendWidth / 2)
129     .attr("height", legendHeight)
130     .attr("width", legendWidth)
131     .style("fill", `url(#${legendGradientId})`)
```

Wonderful! Now we can see our color scale in our legend. Note how it highlights that most countries are *growing* and the countries that are shrinking are only shrinking a small amount.



globe with legend

Lastly, let's label our color scale with the minimum and maximum percent change in our scale. We'll start by labeling the right side, positioning our text 10 pixels to the right of our bar.

code/06-making-a-map/completed/draw-map.js

```
133 const legendValueRight = legendGroup.append("text")
134   .attr("class", "legend-value")
135   .attr("x", legendWidth / 2 + 10)
136   .attr("y", legendHeight / 2)
137   .text(` ${d3.format(".1f")(maxChange)}%`)
```

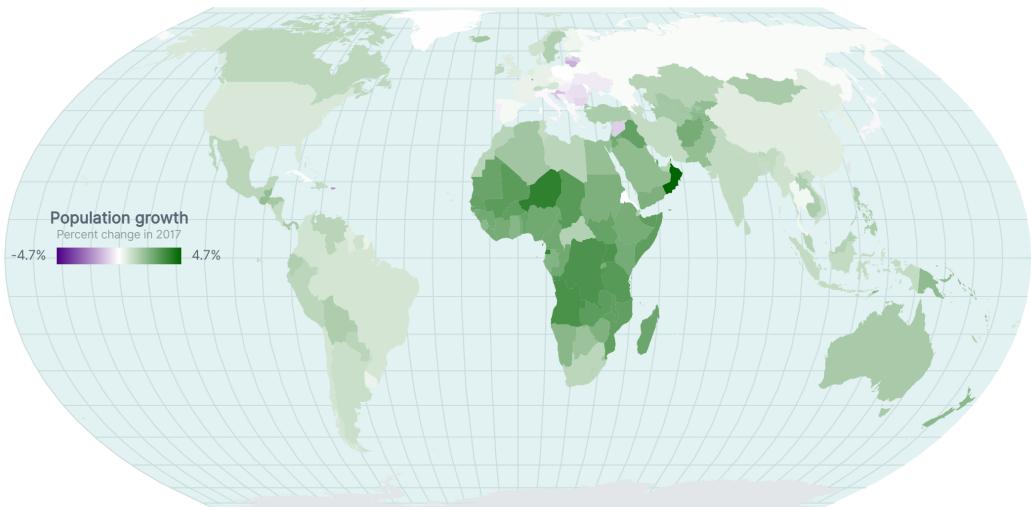
If you're wondering how our text is *vertically centered* with the gradient bar, check out the `styles.css` file. We're using the `dominant-baseline: middle` CSS property.

And now we'll label the left side — we can use the `text-anchor: end` CSS property to align the right side of our text to the left of our bar.

code/06-making-a-map/completed/draw-map.js

```
139 const legendValueLeft = legendGroup.append("text")
140   .attr("class", "legend-value")
141   .attr("x", -legendWidth / 2 - 10)
142   .attr("y", legendHeight / 2)
143   .text(` ${d3.format(".1f")(-maxChange)}%`)
144   .style("text-anchor", "end")
```

Great! Now readers will be able to tell what our colors mean, and also the largest decline and increase in percent of population.



globe with legend

We have a choice here on whether or not we want to use `.nice()` on our color scale to have friendlier min and max values. A color scale that spans -5% to 5% is easier to reason about, but there are two downsides in this case:

1. With a color scale, it's nice (no pun intended) to easily see the max change, which are labeled on our legend.
2. Since it's so hard to distinguish between colors, squishing the scale will make differences harder to see.

As with many aspects of data visualizations, there is no one right answer. But it's important to know the pros and cons and decide based on your goals.

Want more practice creating color scale legends? Adding a legend to the scatter plot we made in **Chapter 2** could help solidify the steps!

Marking my location

One of the fun parts of building in the browser is that we can interact with the user. Let's take advantage of this by plotting the reader's location on our map.

Modern browsers have a global `navigator` object, which provides information about the device accessing the website.

	Desktop						Mobile						
	Chrome	Edge	Firefox	Internet Explorer	Opera	Safari	Android webview	Chrome for Android	Edge Mobile	Firefox for Android	Opera for Android	Safari on iOS	Samsung Internet
Navigator	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes	Yes

navigator support, from developer.mozilla.org

Here's a good resource^a if you're curious about present support or available keys.

^a<https://developer.mozilla.org/en-US/docs/Web/API/Navigator>

`navigator.geolocation` has a method for grabbing the browser's location:

`.getCurrentPosition()`. Most modern browsers⁵³ have support for this method.

⁵³<https://caniuse.com/#feat=geolocation>

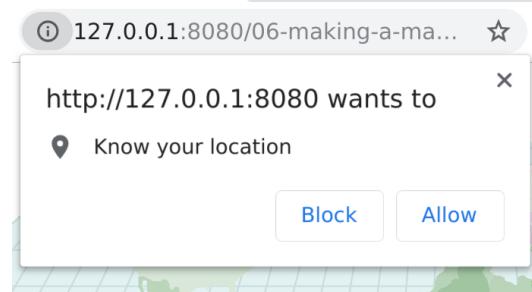


geolocation support, from caniuse.com

We can pass `.getCurrentPosition()` a callback function that has the user's location. Let's run this at the bottom of our code and log out the passed information.

```
navigator.geolocation.getCurrentPosition(myPosition => {
  console.log(myPosition)
})
```

The browser will ask to access our location the first time we load the page. The callback will only run if we click **Allow**.



Browser - dialog to receive location

We can see the structure of `myLocation` after the callback runs.

```
▼ Position {coords: Coordinates, timestamp: 1553996869267} ⓘ  
  ▼ coords: Coordinates  
    accuracy: 1168  
    altitude: null  
    altitudeAccuracy: null  
    heading: null  
    latitude: 43.1  
    longitude: -77.6  
    speed: null  
  ► __proto__: Coordinates  
  timestamp: 1553996869267  
  ► __proto__: Position  
  
myPosition
```

Great! We have access to both a `latitude` and a `longitude` — we know how to handle that! Let's use our `projection` to turn our location (in a `[latitude, longitude]` array), into a set of `[x, y]` coordinates.

code/06-making-a-map/completed/draw-map.js

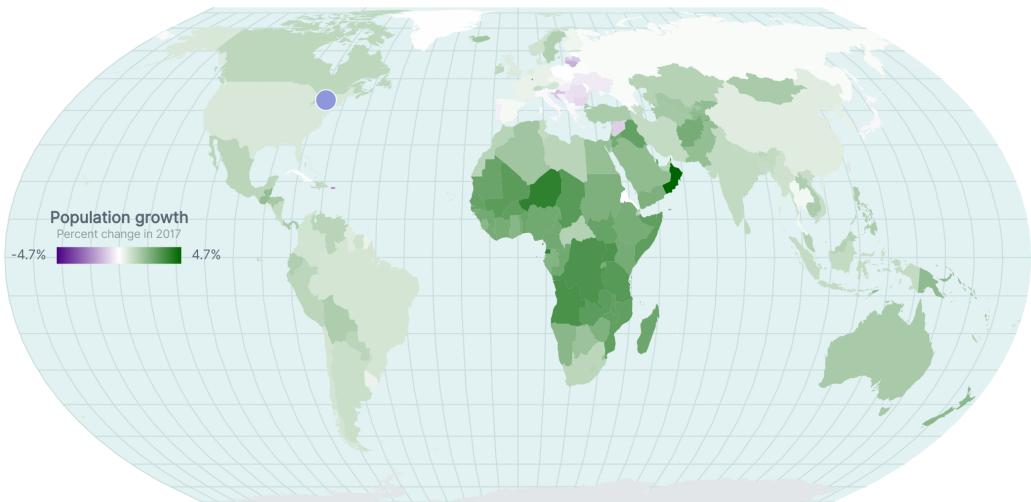
```
147 const [x, y] = projection([  
148   myPosition.coords.longitude,  
149   myPosition.coords.latitude  
150 ])
```

Next, we'll draw a `<circle>` with a class of "my-location" in that location. We'll also animate the circle's radius from 0 to 10, to draw the eye.

code/06-making-a-map/completed/draw-map.js

```
146 navigator.geolocation.getCurrentPosition(myPosition => {  
147   const [x, y] = projection([  
148     myPosition.coords.longitude,  
149     myPosition.coords.latitude  
150   ])  
151   const myLocation = bounds.append("circle")  
152     .attr("class", "my-location")  
153     .attr("cx", x)  
154     .attr("cy", y)  
155     .attr("r", 0)  
156     .transition().duration(500)  
157     .attr("r", 10)  
158 })
```

Nice! It's good to keep in mind that users relate to our data visualizations through their personal lens — seeing themselves in a visualization helps give it meaning.



map with my location

Set up interactions

Lastly, let's use what we learned in the last chapter to give the user more information when they hover over a country.

First, we want to initialize our **mouse events**, using the `.on()` method for our `countries` elements.

code/06-making-a-map/completed/draw-map.js

```
162 countries.on("mouseenter", onMouseEnter)
163   .on("mouseleave", onMouseLeave)
```

Next, we want to grab the tooltip element — this is already created in our `index.html` file. Having a static `tooltip` reference outside of our mouse event callbacks keeps us from having to look it up every time a country is hovered.

code/06-making-a-map/completed/draw-map.js

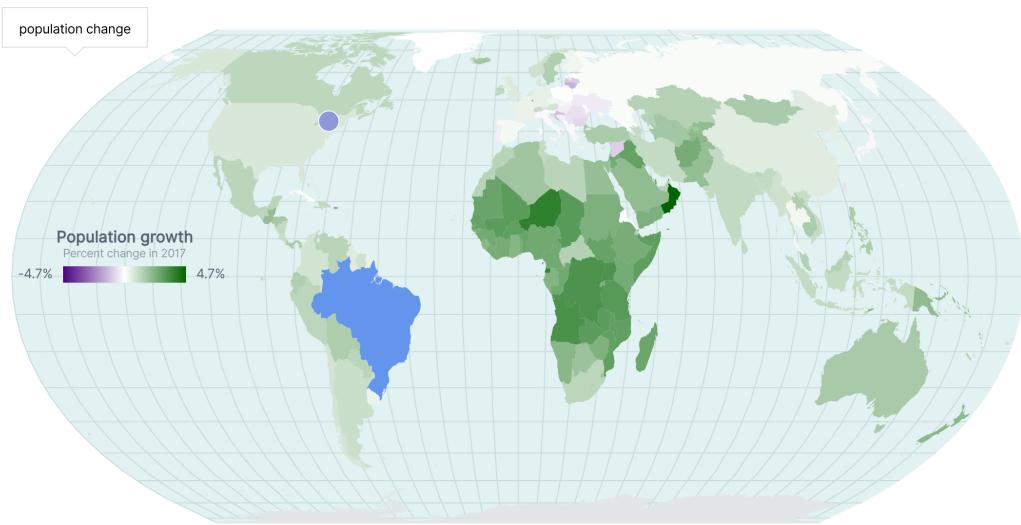
```
165 const tooltip = d3.select("#tooltip")
```

Next, let's initialize our `onMouseEnter()` and `onMouseLeave()` callback functions and have them toggle our tooltip's visibility.

```
function onMouseEnter(datum) {
  tooltip.style("opacity", 1)
}

function onMouseLeave() {
  tooltip.style("opacity", 0)
}
```

Great, now our tooltip shows up when we hover over a country.



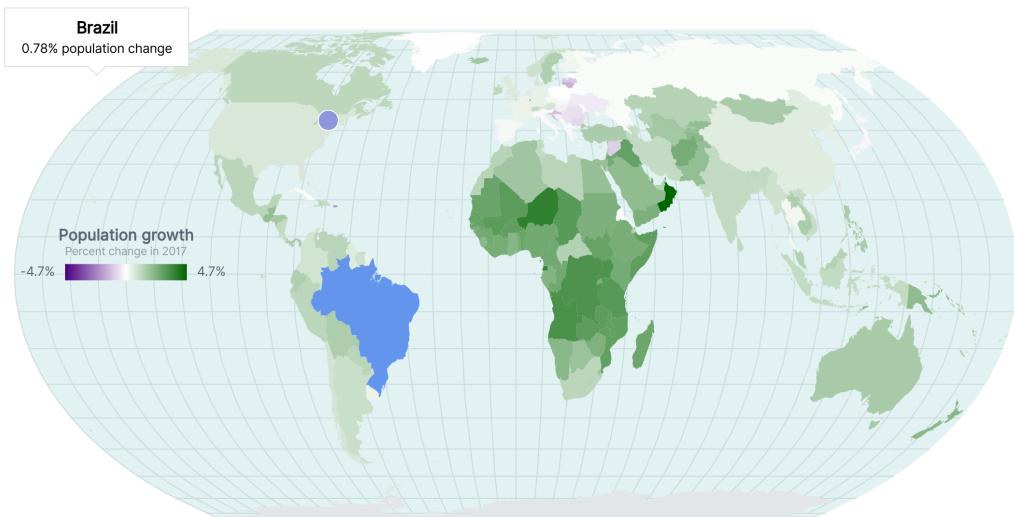
map on hover

Next, let's populate the text of our tooltip. Remember that the first parameter of our mouse event (that we've named `datum`) contains the data bound to the hovered element. We can use this to grab the country name and population growth value. We can check the `index.html` file to find the `ids` of the elements which will contain this text.

[code/06-making-a-map/completed/draw-map.js](#)

```
166 function onMouseEnter(datum) {  
167   tooltip.style("opacity", 1)  
168  
169   const metricValue = metricDataByCountry[countryIdAccessor(datum)]  
170  
171   tooltip.select("#country")  
172     .text(countryNameAccessor(datum))  
173  
174   tooltip.select("#value")  
175     .text(`\$${d3.format(",.2f")(metricValue || 0)}%`)
```

Perfect! Now our tooltip updates to give us information about the currently hovered country.



map on hover, with tooltip text

But how do we know where to position the tooltip? Our `pathGenerator` comes to the rescue again!

Remember how we used `pathGenerator.bounds(sphere)` to get the height of the entire globe? Our `pathGenerator` also has a `.centroid()` method that will give us the center of the passed GeoJSON object. Let's see what we get when we ask for the center of our hovered country's bound data:

```
console.log(pathGenerator.centroid(datum))
```

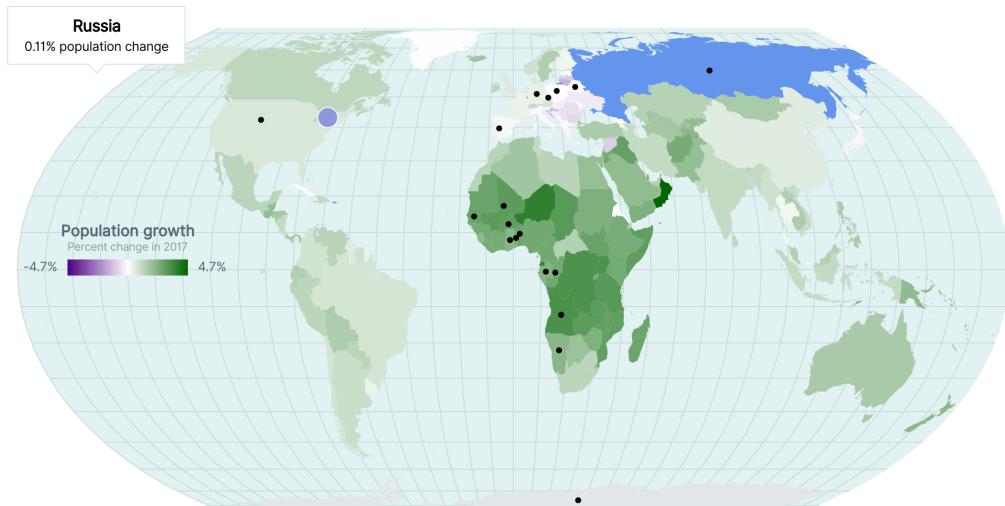
► (2) [362.60341787403684, 289.20854414569305]

map on hover, `.centroid()` response

Great! This looks like an `[x, y]` coordinate. Let's grab that `x` and `y` coordinate and position a new `<circle>` element to make sure this is the correct coordinate.

```
function onMouseEnter(datum) {  
  // ...  
  const [centerX, centerY] = pathGenerator.centroid(datum)  
  const hoveredCircle = bounds.append("circle")  
    .attr("cx", centerX)  
    .attr("cy", centerY)  
    .attr("r", 3)  
}
```

When we hover a country, we can see that a dot appears exactly in the center.



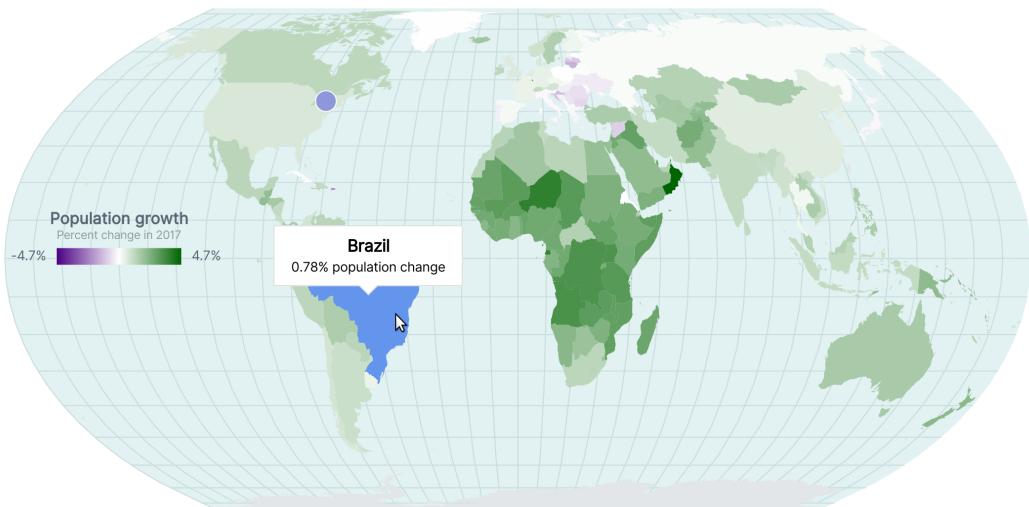
map on hover, dots at .centroid()

Let's take out that `hoveredCircle` code and instead position our tooltip. Remember that we want to shift it with our `top` and `left` margins, since our tooltip element lives outside of our `bounds`. Just like in the last chapter, we'll need to use `calc()` to accommodate the tooltip's own width and height.

code/06-making-a-map/completed/draw-map.js

```
177 const [centerX, centerY] = pathGenerator.centroid(datum)  
178  
179 const x = centerX + dimensions.margin.left  
180 const y = centerY + dimensions.margin.top  
181  
182 tooltip.style("transform", `translate(`  
183     + `calc( -50% + ${x}px),`  
184     + `calc(-100% + ${y}px)`  
185     + `)`)
```

Perfect! Now our map is easy to explore, and users can dig into the exact population growth numbers.



map on hover, with correctly positioned tooltip

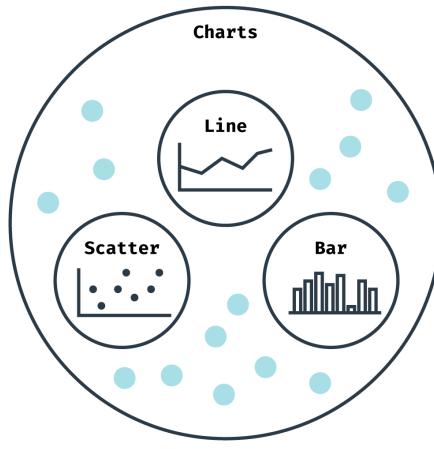
With this method of positioning the tooltip, it can be very hard to get more information about small countries. An alternative is to use the `voronoi` method we used for our scatter plot in the last chapter. This is implemented in the `/code/06-making-a-map/completed-voronoi/` folder. Feel free to check out the code and compare the two methods in the browser.

Wrapping up

Way to go! We learned a *ton* of new concepts in this chapter. Play around with downloading different GeoJSON files, changing the projection, and using different data to build on the skills we learned in here. Creating maps is a whole discipline, and this is just the tip of the iceberg!

Data Visualization Basics

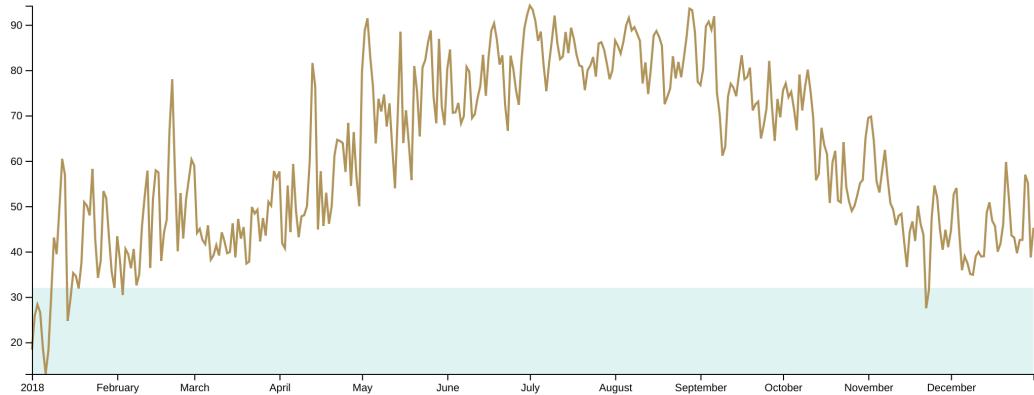
Now that we're comfortable with *how* to create a chart, we should zoom out a bit and talk about *what* chart to create. So far, we've created a line, scatter, and bar chart, but there are many more basic types to choose from, as well as charts that don't fall into a simple category.



Zooooming out

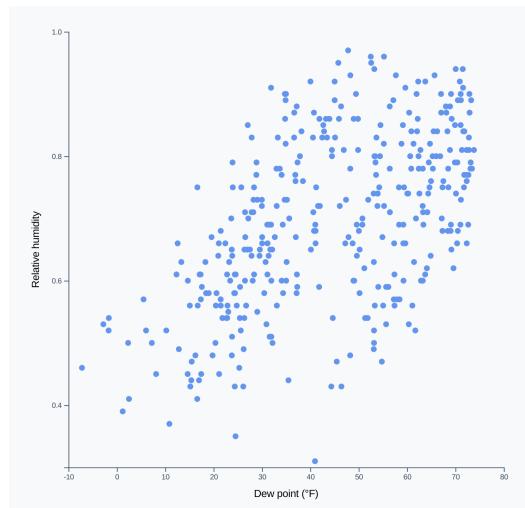
The format of our chart is the largest factor in what information our users take away — even given the same dataset! Let's look at the charts we've made with our weather data.

In **Chapter 1**, we charted a timeline of maximum temperatures. Looking at this chart, we could see how the temperature changed over time - how consistent was the weather day to day or season to season?



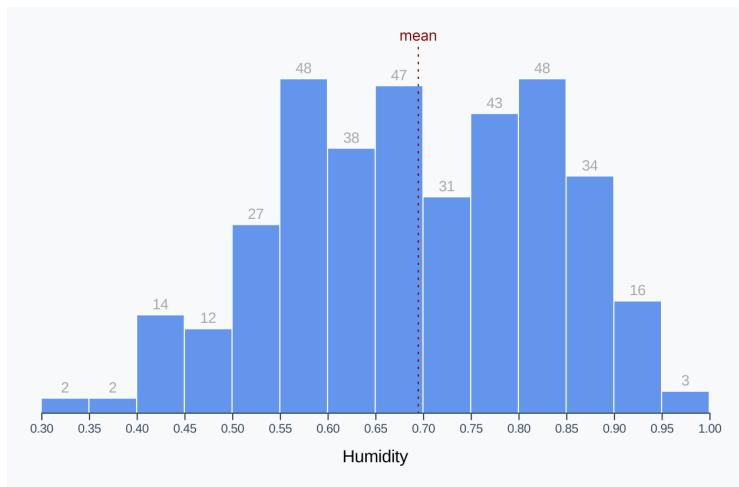
Finished line graph

In **Chapter 2**, we created a scatter plot with two metrics. Looking at this chart, we could see at how humidity and dew point are related, answering questions such as: does a high humidity also meant a high dew point?



Our Finished Scatterplot

In **Chapter 3**, we created a histogram of humidity values. Looking at this chart, we could see how much variety there was in a single metric — do most days stay around the same humidity level, or are they all very different?



Finished humidity histogram

Even with these three examples and a limited dataset, we can see how the type of chart will enable the user to answer very different questions. There are also many types of charts — thus, answering the question **What type of chart is best?** is both important and overwhelmingly open-ended. Don't worry, though — this chapter will equip you with the tools to make that decision, and quickly!

Let's start at the beginning: with our data.

Types of data

Given a dataset, the first task is to determine the structure of the available metrics. Let's look at the first item of our weather dataset.

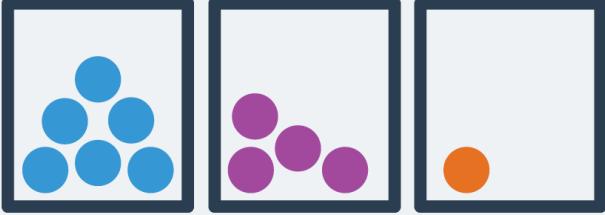
(index)	Value
time	1514782800
summary	"Clear throughout the day."
icon	"clear-day"
sunriseTime	1514809280
sunsetTime	1514842810
moonPhase	0.48
precipIntensity	0
precipIntensityMax	0
precipProbability	0
temperatureHigh	18.39
temperatureHighTime	1514836800
temperatureLow	12.23
temperatureLowTime	1514894400
apparentTemperatureHigh	17.29
apparentTemperatureHighTime	1514844600
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	1514844600
date	"2018-01-01"

Our dataset

There are many different values here, but two basic types: **strings** and **numbers**. These two types can roughly be split (respectively) into two basic types of data: **qualitative** and **quantitative**.

Qualitative data (our strings) does not have a numerical value, but it can be put into categories. For example, `precipType` can either have a value of “rain” or “snow”.

Qualitative



Usually a string

data type - qualitative

Quantitative data (our numbers) is numerical and can be measured objectively. For example, `temperatureMax` has values ranging from 10°F to 100°F.

Quantitative



Usually a number

data type - quantitative

Both of these types of data can be broken down even further.

Qualitative Data

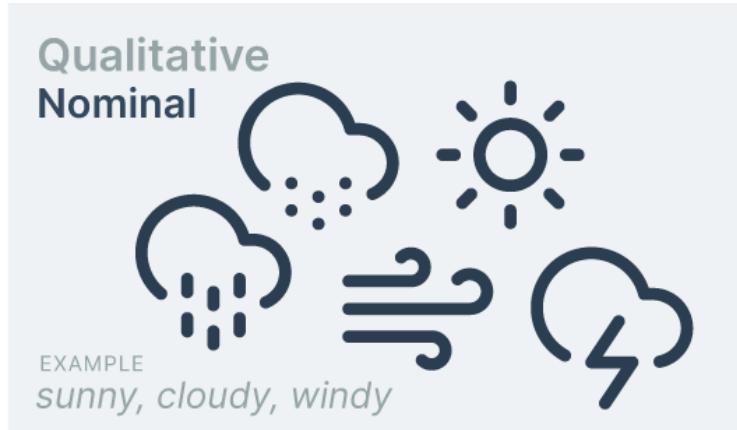
Binary data can be placed into only two categories.



data type - binary

For example, if our weather data had an **did rain** metric that was either **true** or **false**, that metric would be binary.

Nominal data can be placed in multiple categories that don't have a natural order.



data type - nominal

For example, our weather data has the metric **icon** with values such as **clear-day** and **wind** — these values can't be ordered.

Ordinal data can be placed in multiple categories with a natural order.

Qualitative Ordinal



EXAMPLE
not windy, very windy

data type - ordinal

For example, if our weather data instead represented **wind speed** values with **not windy**, **somewhat windy**, and **very windy**, that metric would be ordinal.

Quantitative Data

Discrete data has numerical values that can't be interpolated between, such as a metric that can only be represented by an integer (whole number).

Quantitative Discrete



EXAMPLE
2 tornados, 3 tornados

data type - discrete

A classic example is **number of kids** — a family can have 1 or 2 kids, but not 1.5 kids. With weather data, a good example would be number of tornados that happened.

Continuous data has numerical values that can be interpolated between. Usually a metric will fall under this category — for example, **max temperature** can be 50°F or 51°F or 50.5°F.



data type - continuous

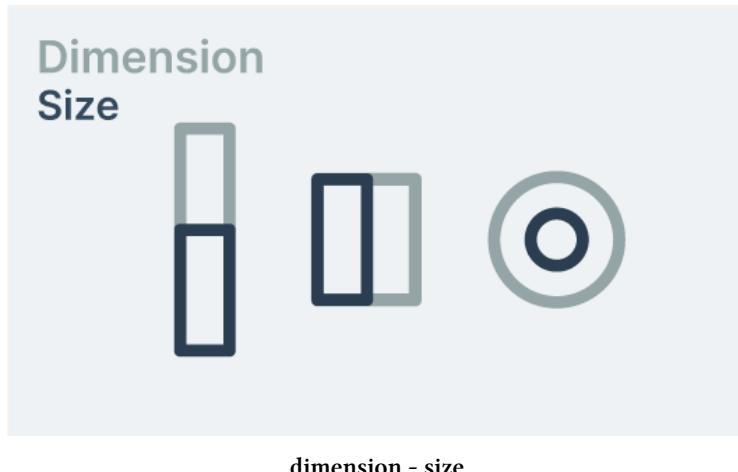
This categorization is just one way to group types of data. There are other ways, but this categorization is common and a handy place to start. When you are looking at a new dataset, familiarize yourself by categorizing each metric.

Ways to visualize a metric

Now that we understand the metrics that we're working with, we need to decide how to represent them visually. Keep in mind that we, as humans, are able to judge some dimensions more quickly or precisely than others.

Let's go through some possible ways we could represent **temperature**, as an example.

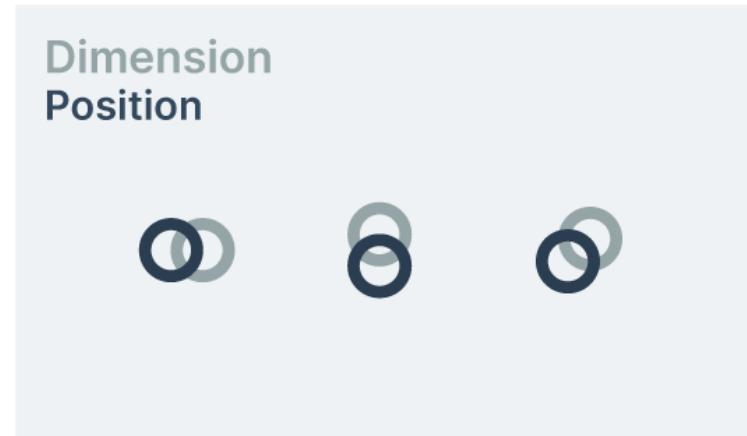
Size



We could represent **temperature** by sizing an element: larger for higher temperatures and smaller for lower temperatures. We could also scale just the height or the width — when making our histograms, we created taller bars for higher numbers.

Humans are very fast and precise at perceiving size differences (especially height or width) between objects, making it a good choice when the exact differences matter.

Position

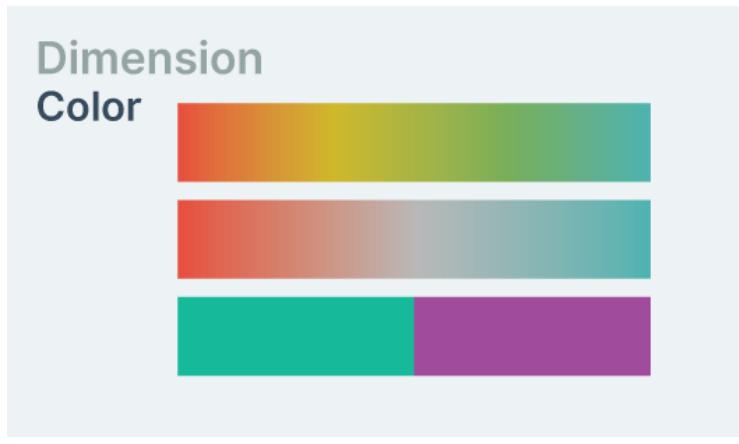


dimension - position

We could represent **temperature** by moving an element: horizontally, vertically, even diagonally. For example, we could make a scatter plot with lower temperatures to the left and higher temperatures to the right.

Like size, humans are very fast and precise at comparing horizontal and vertical positions, making it a great choice when precision matters.

Color



We could represent **temperature** by giving our elements a different color. For example, we could make a scatter plot with dots colored from blue (lower temperatures) to red (higher temperatures). Choosing colors can be overwhelming — don't worry, we have a section to help with just that coming up in this Chapter.

Humans are less adept at distinguishing two colors than distinguishing two sizes or two positions. However, we can more easily average colors together, making color a good choice for visualizations where the overall picture matters. If both the specific values and the overall picture are important, try showing the overall picture and progressively surfacing exact values with tooltips or a companion table.

There are other dimensions (for example, **orientation** or **pattern**), but these basic ones will get you thinking on the right path. These dimensions might remind you of a concept we've been using: **scales**. Scales have been helping us translate data values into a physical dimension: for example, **temperature** into **height of a bar in pixels**.

When you have a dataset that is chock full of metrics like our weather data, it often isn't ideal to visualize all of it in one chart. Focused charts are the most effective — sticking with 2-4 dimensions will help the user focus on what's important without being overwhelmed.

Putting it together

It can be tempting to jump right in and say “Oh I know, I’m going to make a bar chart!” But let’s investigate further: what is a bar chart made out of? A bar chart is made up of **bars** that vary in size (**height**) and are spread out horizontally (**position**).

In the next chapter, we’ll talk about common types of charts and how the **type of metrics** and **ways to visualize them** apply to each. But there is a whole world of chart types that haven’t yet been created! Now that you have this framework, you’ll be able to look at a dataset and brainstorm unique ways to visualize it.

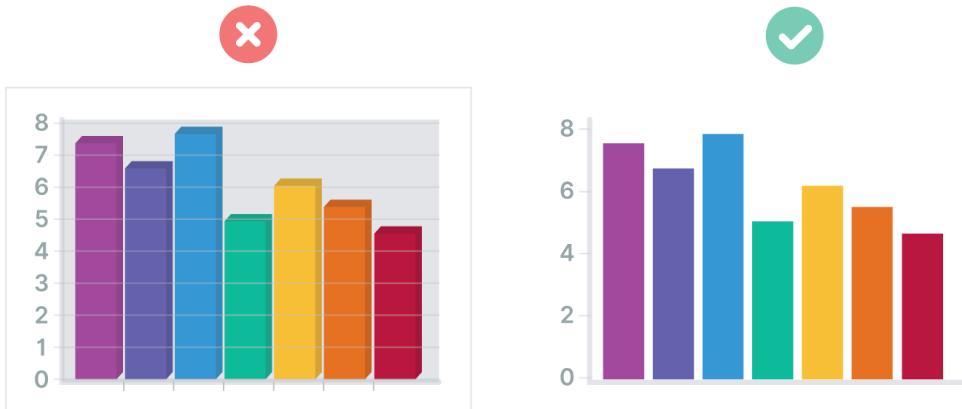
Chart design

Now that you understand these basics, you have the tools you need to make effective and intuitive charts. Once you’ve decided on the what and the how, here are some tips for keeping those charts easy to read.

Simplify, simplify, simplify

Many chart libraries will include default styles in order to satisfy as many use cases as possible. This can create charts with more elements than are necessary. Fortunately, we don’t need to rely on a library and can strip away as much as possible.

After you finish creating a chart, give it a critical look and ask yourself, “Is everything on here necessary?” If the answer is no, remove the cruft! No need to overwhelm your reader.



Simplified chart

Annotate in-place

It can be tempting to throw a legend next to a chart to clarify a color or size scale. While we should make sure to explain every part of our chart, ask yourself if you can label these elements directly. In-place annotations put the description as close as possible, which prevents forcing the reader to look back and forth between elements.

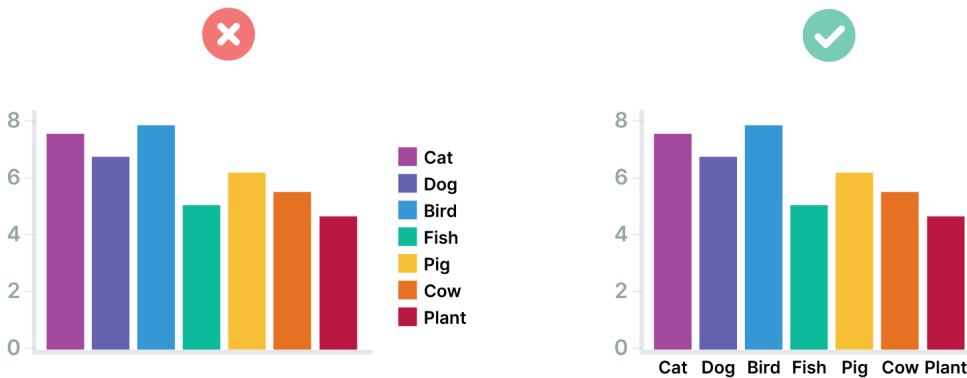
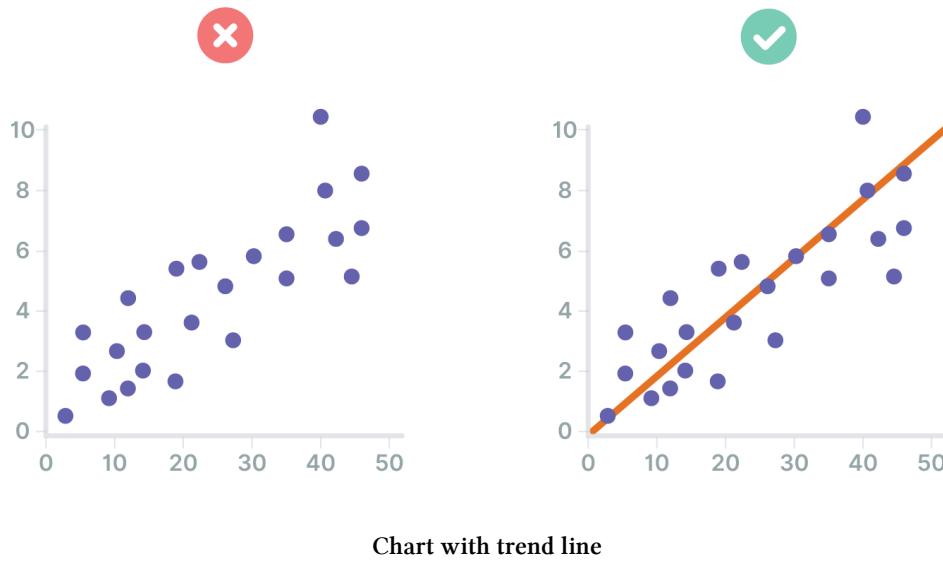


Chart with labels

Add enhancements, but not too many

As a corollary to the last two tips, check to see if you can enhance your chart to help create insights. For example, if the goal of your chart is to show trends, consider adding a trend or reference line to make any patterns more apparent.



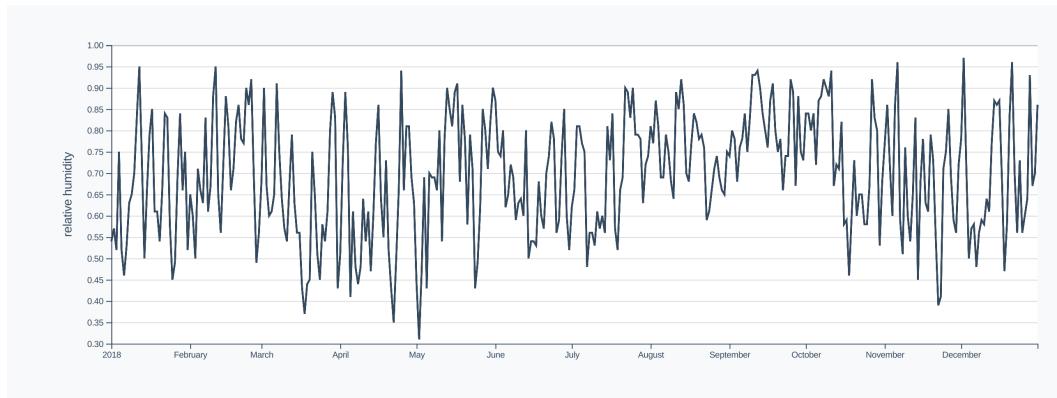
Example redesign

Let's redesign a simple chart to see these tips in practice. The goal of this chart is to examine how humidity changes, depending on the time of year.

We've decided to visualize this data using a timeline — open it up at <http://localhost:8080/07-data-visualization-basics/humidity-timeline/draft/>⁵⁴. We'll slowly work towards the finished version at <http://localhost:8080/07-data-visualization-basics/humidity-timeline/completed/>⁵⁵. Don't peek yet!

⁵⁴<http://localhost:8080/07-data-visualization-basics/humidity-timeline/draft/>

⁵⁵<http://localhost:8080/07-data-visualization-basics/humidity-timeline/completed/>



Humidity timeline

For practice, take a minute and write down the ways you would improve on this chart, with our goal in mind.

Goal: examine how humidity changes, depending on the time of year

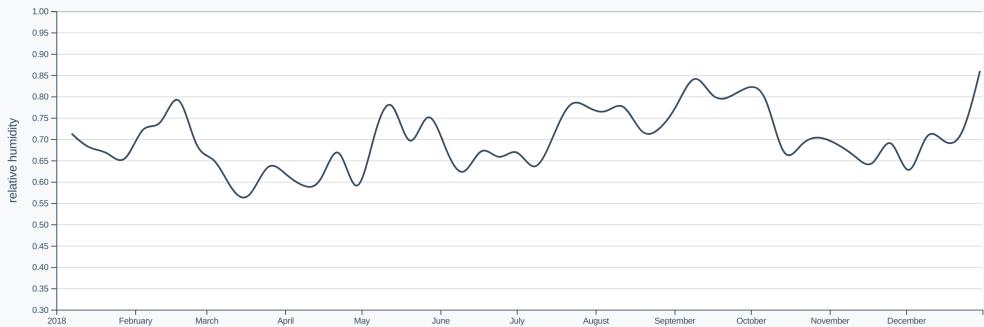
Ready to dive in? There is no “correct” way to improve on this chart, but we’ll do our best to make it easier to digest quickly.

The first problem we’ll tackle is the noisiness of the data. While the daily dips and spikes may help answer other questions, they only distract from our goal of investigating seasonal trends. Let’s downsample the data to one point per week and smooth our line using `d3.curveBasis()`.

If you’re following along, we have a `downsampleData()` function at the bottom of the file that we can pass our `dataset`, `xAccessor`, and `yAccessor` and receive a downsampled dataset with weekly values.

`d3.line()` has a method `.curve()` that can be passed a interpolation function. The default interpolation is linear: the line connects points directly. For a smoother line, check out the different interpolation options built in to d3 in the [d3-shape documentation^a](https://github.com/d3/d3-shape#curves). Here, we're adding the code `.curve(d3.curveBasis)` to diminish the importance of specific data points and give the reader an idea of the overall line shape.

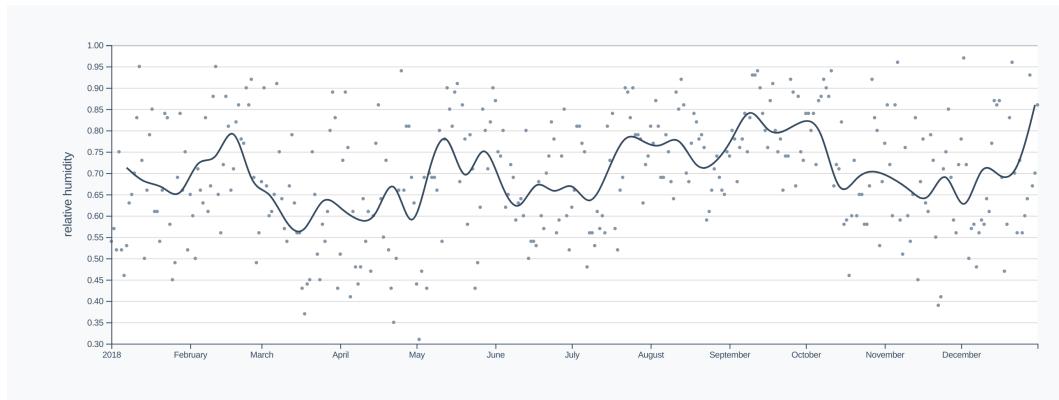
^a<https://github.com/d3/d3-shape#curves>



Humidity timeline, smoothed

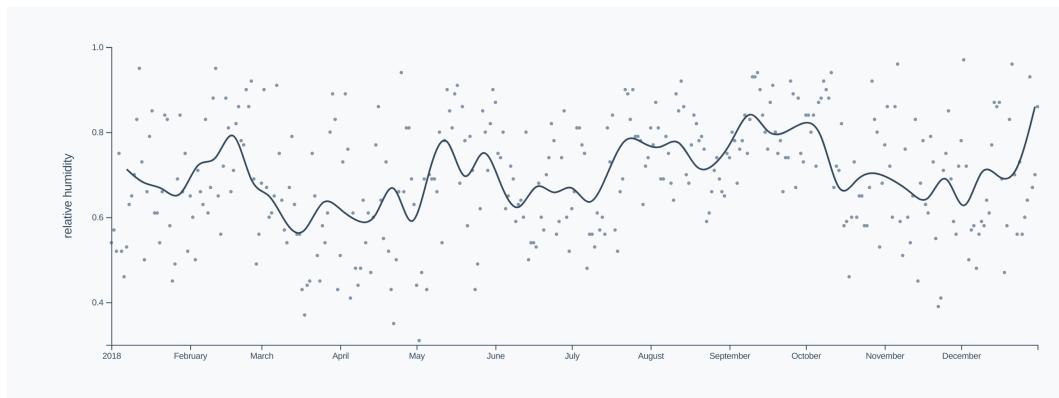
Let's add the original points back in, in the form of small circles. We don't want to lose the granularity of the original data, even when we're getting the basic *trend* with the downsampled line.

To prevent from drawing attention away from our trend line, we'll make these dots a lighter grey.



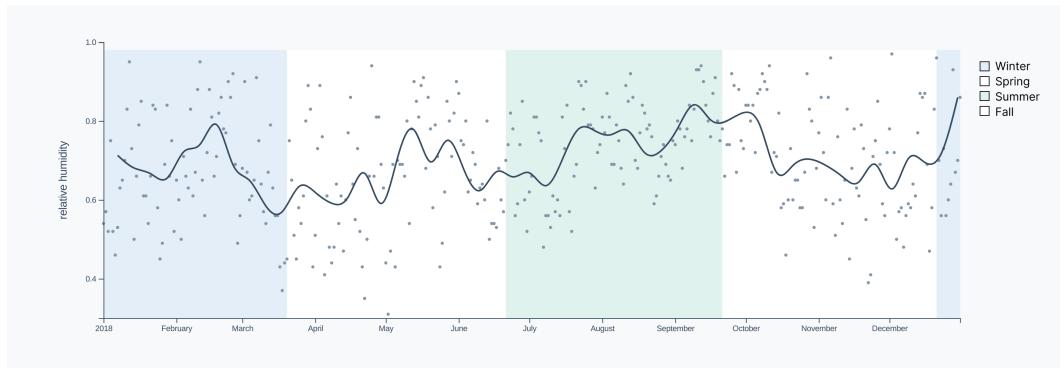
Humidity timeline, smoothed with dots

There's a lot going on right now! Let's simplify a bit and take out the grid marks and chart background. We'll also specify a smaller number of ticks (`.ticks(3)`) for our y axis — the extra ticks aren't giving any useful information since readers can easily extrapolate between values.



Humidity timeline, simpler

Revisiting our goal, we realize that we want to focus on trends based on the time of year. We're showing the months on our x axis, but we can do some work for the reader and highlight the different seasons. Let's block out each season with a `<rect>` underneath our main data elements.

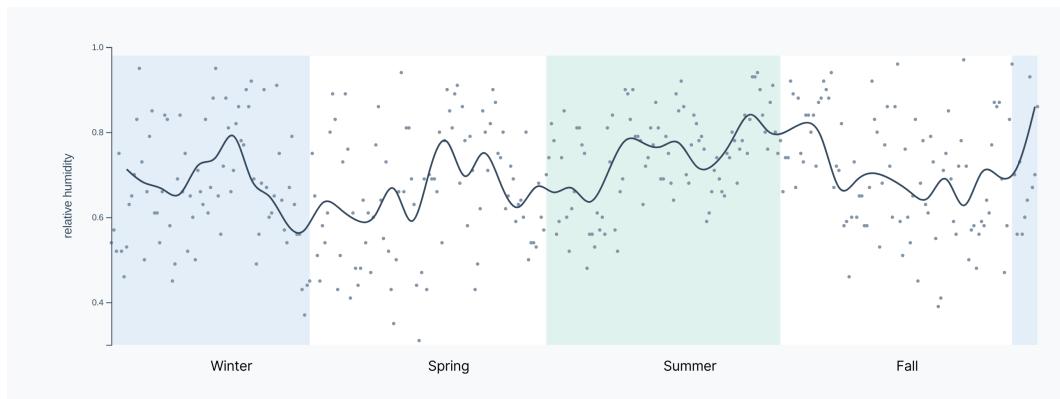


Humidity timeline, with seasons

While the legend is helpful, there are a few issues with it.

1. Both Spring and Fall are white because they are transitional seasons. This makes it unclear which is which on the chart.
2. As we discussed earlier in the chapter, legends make the reader look back and forth between the chart and the legend.

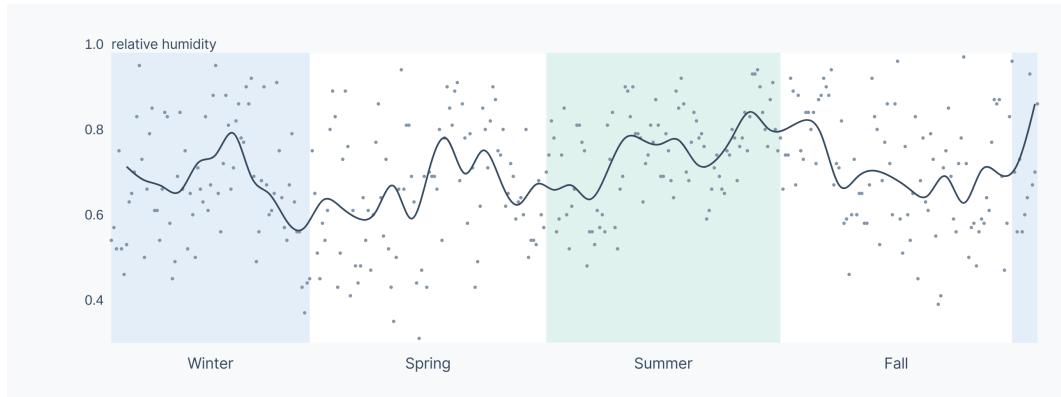
To fix these issues, let's label the seasons directly on the chart instead of having an x axis.



Humidity timeline, with season labels

Let's focus on the y axis. Rotated y axis labels are great — they're often the only way to fit the label to the side of our chart. However, the label is divorced from the values, and rotating text makes it harder to read.

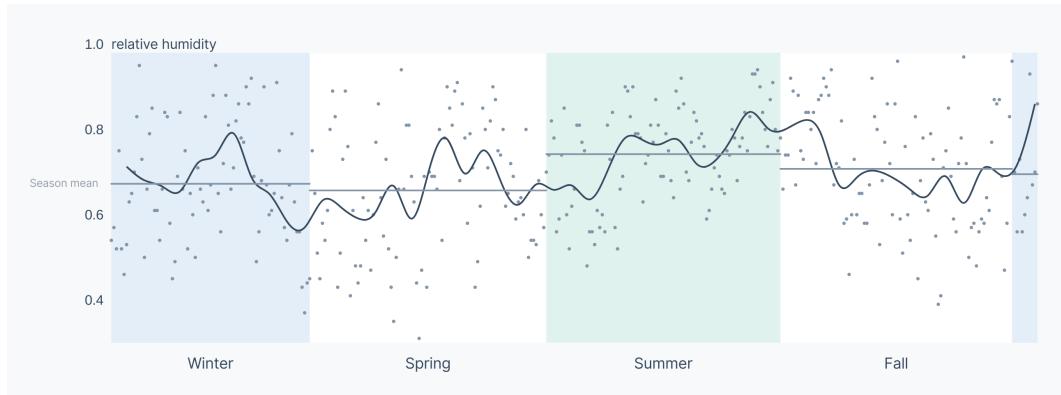
Instead, let's signify the units of our y axis by incorporating it into a phrase with our top y tick value. Human-readable labels can do some of the digesting work a reader has to do, hinting at how to interpret a number.



Humidity timeline, with inline y label

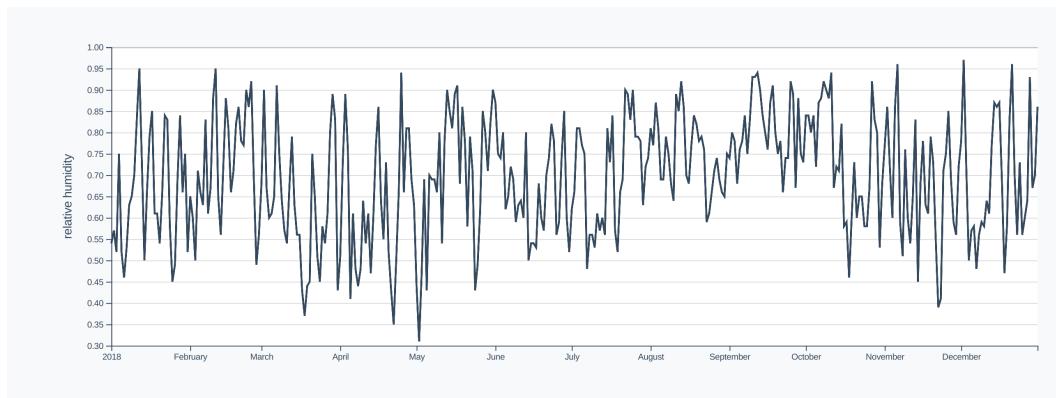
While we've made it easier to compare trends across seasons, it's not easy to conclude how the seasons compare in general. We *could* downsample our data and only display one point per season, but then we'd lose any insights from seasonal trends.

Instead, let's add seasonal means as lines, which should enhance the chart but not take away from the main picture.

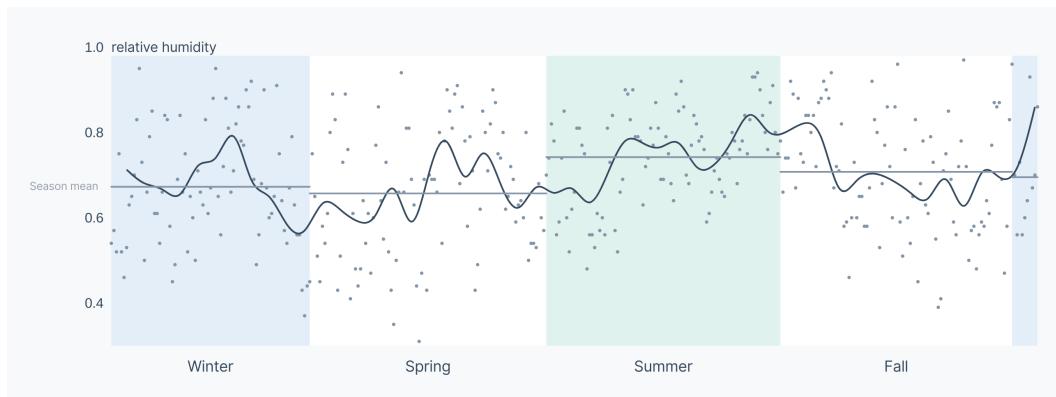


Humidity timeline, with means

Nice! Now we can easily see that the humidity is low in the Winter, but not as low as in the Spring. Let's look at two charts side-by-side.



Humidity timeline, start



Humidity timeline, finish

While both charts display the same data and can both be helpful, the final version is much more effective for our goal: to **examine how humidity changes, depending on the time of year.**

Colors

One of the hardest parts of creating charts is choosing colors. The wrong colors can make a chart unappealing, ineffective, and, worst of all, impossible to read. This section will give you a good framework for choosing colors, important facts about human perception, and simple tips.

Color scales

When choosing a color scale, you first want to identify its purpose. Going back to what we know about data types, there are three basic use cases:

1. Representing a category
2. Representing a continuous metric
3. Representing a diverging metric

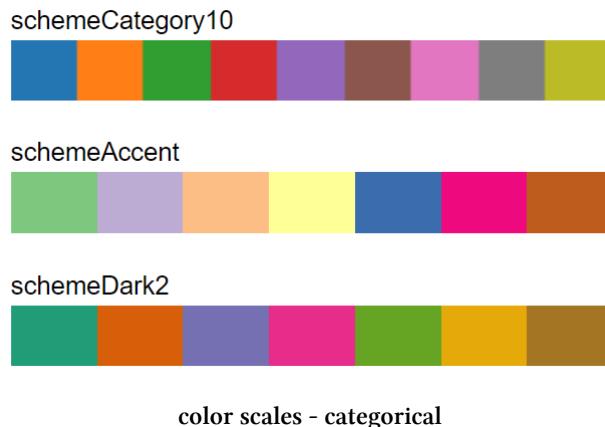
Let's dive deeper into each of these.

1. Representing a category

The first two data types we talked about (**binary** and **nominal**) will be best represented with a categorical color scheme.

Since our metric values don't have a natural order, we don't want to use a color scheme that has a natural order (like white to black).

d3 has built-in color schemes in its [**d3-scale-chromatic**](https://github.com/d3/d3-scale-chromatic)⁵⁶ library. We can see them listed under **Categorical** if we start our server (`live-server`) and navigate to <http://localhost:8080/07-data-visualization-basics/scales/>⁵⁷ in our browser.



⁵⁶<https://github.com/d3/d3-scale-chromatic>

⁵⁷<http://localhost:8080/07-data-visualization-basics/scales/>

These **categorical** color schemes have been carefully designed to have enough contrast between colors. Each of these schemes is an array of colors — for example, to use the first scale in this list, we would access each color at its index in `d3.schemeCategory10`.

For **categorical** color schemes, it's helpful for each color to have a different descriptive name. For example, there might be two colors that could be described as "blue", which can be confusing to talk about.

2. Representing a continuous metric

For metrics that are **continuous**, we'll want a way to interpolate in between color values. For example, we could represent **humidity** values with a color scale ranging from white to dark blue.

d3-scale-chromatic⁵⁸'s built-in continuous scales are visible under the **Sequential** section.

interpolateBlues



interpolateGreens



interpolateGreys



color scales - sequential

These are color scales instead of color schemes — `d3.interpolateBlues()` is a function instead of an array of colors. To get a color, we can give `d3.interpolateBlues()` a decimal in between `0` and `1` — `0` would return the leftmost color (a light gray)

⁵⁸<https://github.com/d3/d3-scale-chromatic>

and 1 would give us the dark blue on the right. To put it in familiar terms, `d3.interpolateBlues()` is a scale with a domain of $[0, 1]$ and a range of [light gray, dark blue].

These single-hue scales are great for basic charts and charts with multiple color scales. However, sometimes the steps in between color values are too small and it becomes hard to distinguish between values. In this case, `d3-scale-chromatic` has many **sequential** color scales that cycle through another hue, increasing the difference between values.

interpolateBuGn



interpolateBuPu



interpolateGnBu



color scales - sequential

We can see how much easier it is to tell the difference between colors 50% and 60% of the way through a single-hue scale (left) versus a multi-hue scale (right).

interpolateBlues



interpolateGnBu



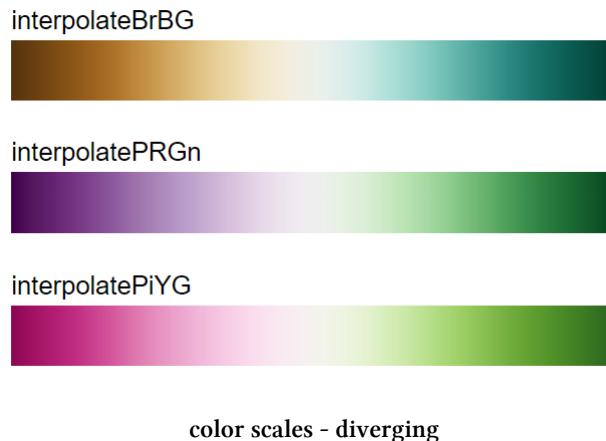
color scales - sequential - diff

3. Representing a diverging metric

Our **sequential** color scales get more visually clear as they approach the end of the scale. This is great to highlight values on the high end (for example, the highest **humidity** values will be the most visible).

Sometimes, however, we want to highlight both the lowest *and* highest metric values. For example, if we're looking at **temperature**, we might want to highlight the coldest days as a bright blue and the hottest days as a bright red.

Diverging scales start and end with very saturated/dark color and run through a less intense middle range.



We can see that we have both single-hue (per side) and multi-hue diverging scales. Again, it's helpful to cycle through more hues so users can pick up on smaller differences between metrics, but it can get overwhelming when we are already cycling through two hues. When getting a feel for color scales, try a few different options to get an idea for what will work in your specific scenario. This is not a one-size-fits-all decision.

Custom color scales

Sometimes the built-in color scales don't cut it. Maybe you have specific colors that you want to work in, or you are going for a particular tone with your chart. No

worries! We can make our own scales easily using `d3.interpolate`⁵⁹.

Let's say that we wanted to represent temperature values with a color, ranging from "cyan" to "red". There are several methods that we could use to interpolate between those two colors, including `d3.interpolateRgb()` and `d3.interpolateHsl()`.

Let's make our own scale, running from "cyan" to "tomato" — all we need to do is pass our starting and ending color as separate parameters to `d3.interpolateHcl()`.

We'll talk about the difference between the different color spaces (`rgb`, `hsl`, `hcl`) in the next section.

```
d3.interpolateRgb("cyan", "tomato")
```

That will give us this scale:



Our code has been started off with a few different custom scales.

interpolate-rgb



interpolate-hsl



interpolate-hcl



color scales - custom

⁵⁹<https://github.com/d3/d3-interpolate>

If we open up the `07-data-visualization-basics/draw-scales.js` file in our text editor, we can test our own custom scales at the bottom. In the `index.html` file, we are importing other JavaScript files that list the built-in scale types and define a `drawColorRange()` function that creates an `<svg>` element with a `linear-gradient`.

We also have a utility function, `addCustomScale()`, that accepts two parameters:

1. the scale's name (which needs to be a string without spaces, since we're using it to create and reference an `id`), and
2. a color scale (a function that returns a color when passed a value between `0` and `1`)

We've already created a few custom scales — check out the first three which have the same range but use different color spaces for interpolation.

We can also create new discrete color schemes. The last two custom scales are using a function called `interpolateWithSteps()` defined on line 33. `interpolateWithSteps(n)` returns a new array of `n` elements interpolating between `0` and `1`. For example, `interpolateWithSteps(3)` returns `[0, 0.5, 1]`.

We can use this function to make a new color scheme by stepping through a color scale and returning equally spaced through the range. For example:

```
1 interpolateWithSteps(6).map(  
2     d3.interpolateHcl("cyan", "tomato")  
3 )
```

will create the following color scheme.



We can look at the color scheme by passing a unique id and our scale to `addCustomScale()`.

```
addCustomScale(  
  "interpolate-hcl-steps",  
  interpolateWithSteps(6).map(  
    d3.interpolateHcl("cyan", "tomato")  
  )  
)
```



color scales - custom - discrete

If we wanted to create discrete color schemes with many colors, we could use one of **d3-scale-chromatic's** cyclical color scales.



color scales - cyclical

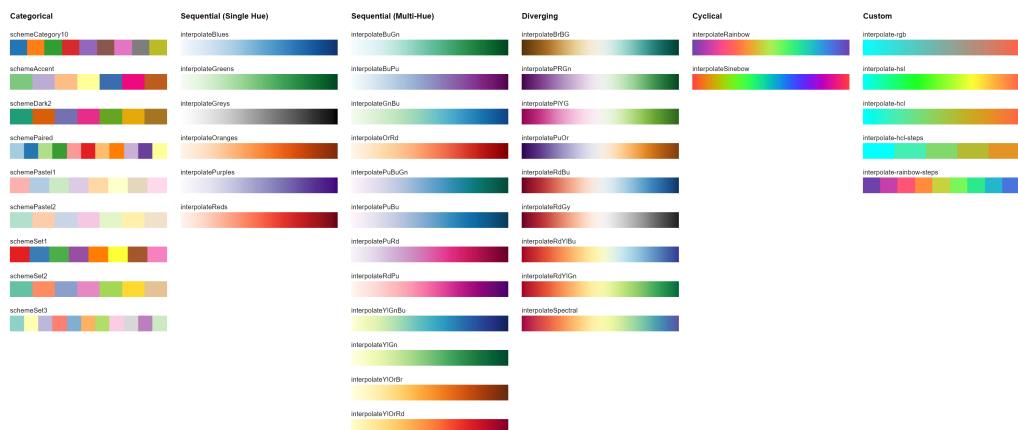
We can use our `interpolateWithSteps()` function to split this continuous color scale into a color scheme.

```
interpolateWithSteps(10).map(  
  d3.interpolateRainbow  
)
```



color scales - custom - discrete

Play around and make a few color scales or color schemes to get the hang of it.



color scales

Creating our own colors

Colors can be represented in various formats, each with different strengths. But creating our own colors requires care and the *color space* we use affects how well our chart can be interpreted.

Below, we're going to look at three color spaces:

- rgb
- hsl and
- hcl

While `rgb` is generally the most familiar, there are good reasons to use `hcl` when we're programmatically creating color scales. Read on to find out why.

keywords

In our chart code, we've used a lot of **keywords** which map to specific, solid colors. For example, we made our histogram bars `cornflowerblue`. Color keywords are great because they are easy to remember and mirror the way we refer to colors in natural language.

`currentColor`

Most color keywords are ways we would describe a color in English, but there is one handy special case: `currentColor`. `currentColor` will set the CSS property to the current `color` property. This is great for controlling one-color charts from outside a charting component. For example, we could make a line chart component whose `stroke` is `currentColor`. This way, we could easily create multiple line charts with different line colors, just by setting the `color` higher up in the DOM.

`transparent`

Another useful color keyword is `transparent`. This is great when creating invisible SVG elements that still capture mouse events, such as the listening rect we used to capture mouse movement in [Chapter 5](#).

When creating data visualizations, we'll often need to manipulate colors. In this case, we can't use the color keywords because there is no way to make `cornflowerblue` 10% darker. In order to manipulate our colors, we'll need to learn about color spaces.

`rgb`

The color space you're most likely to come across when developing for the web is `rgb`. `rgb` colors are composed of three values:

`r`: red

`g`: green

`b`: blue

For each of these values, a higher number uses more of the specified color. Then these values are combined to create one color.

This is essentially how LCD screens work: each pixel is made up of a red, green, and blue light. The values are combined in an additive fashion, starting at black and moving towards white with higher values. This may be counter-intuitive to anyone who's used paint on paper, where more paint = a darker color. For example, `rgb(0,0,0)` represents **black** and `rgb(255,255,255)` represents **white**.

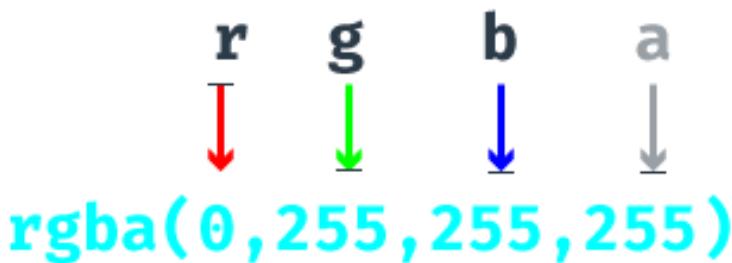
There is another optional value for all of the color spaces we'll talk about:

a: alpha

The alpha value sets the opacity of the color — a color with an alpha value of 0 will be transparent. If the alpha value isn't specified, the color will be fully opaque.

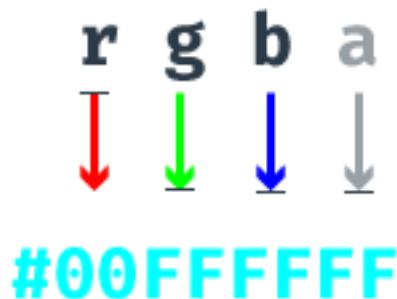
`rgb` can be expressed in two formats. The first, **functional notation**, starts with `rgb` and lists each color value, in order, within parentheses. Each value will be within a range from 0 to 255. If we wanted to specify an alpha value, we would switch the prefix to `rgba`.

For example, we would represent a **cyan** color as a combination of **green** and **blue** colors.



color space - `rgb`

`rgb` colors can also be expressed with **hexadecimal notation**, which begins with # and lists each value with two characters in a range from 00 to FF.



color space - rgb as hex

rgb can be an unintuitive color space to work in — if I have a blue color and want an orange of the same brightness, all of the values have to change.

rgb(39, 204, 199)

rgb(204, 118, 41)

color space - rgb shift

Next, let's look at a color space that is closer to our mental model of color.

hsl

In the hsl color space, the values don't refer to specific colors — instead, they refer to color properties:

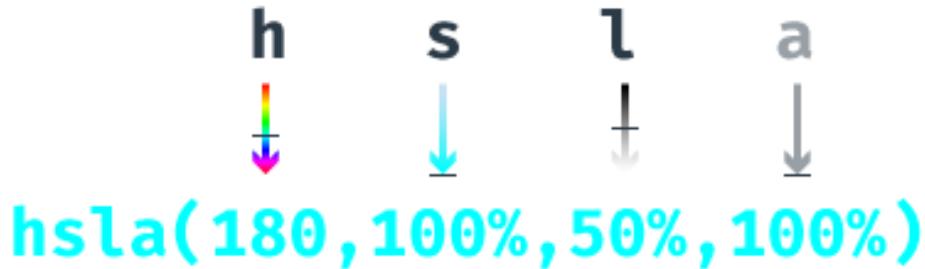
h: hue. The hue represents the angle around a circular color wheel that starts at red (0 degrees) and cycles through orange, yellow,, back around to red (360 degrees).

s: saturation. The saturation value starts at gray (0%) and ramps up to a completely saturated color (100%).

l: lightness. The lightness value starts at white (0%) and ramps up to black (100%).

(a: alpha. Again, the alpha channel is optional and defaults to full opacity (100%))

In **hsl**, our cyan color would be partially around the color wheel, fully saturated, and of medium lightness.



color space - hsl

hsl more closely matches our mental model of the relationship between colors — to switch from a blue to a similarly dark & saturated orange, we would only have to update the **hue** value.



color space - hsl shift

hcl

The **hcl** color space is similar to **hsl**, with the same values (**c** is for **chroma**, which is an alternative measure of colorfulness).



hcl(180, 100, 150, 100)

color space - hcl

Instead of being spatially uniform, the values are **perceptually uniform**, so that a red and a blue with a lightness of 50% will *look* the same amount of light. Let's look at the hue spectrum at 100% saturation and 50% lightness for both **hsl** (top) and **hcl** (bottom).



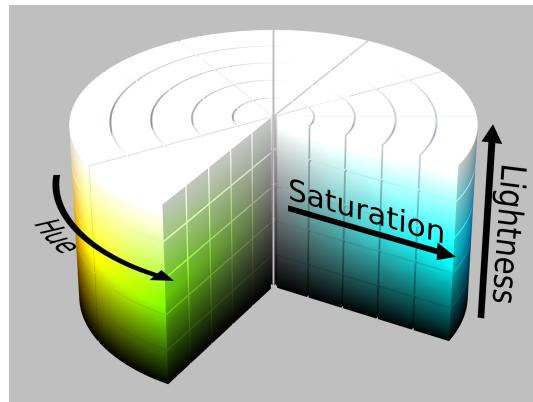
color space - hsl hues



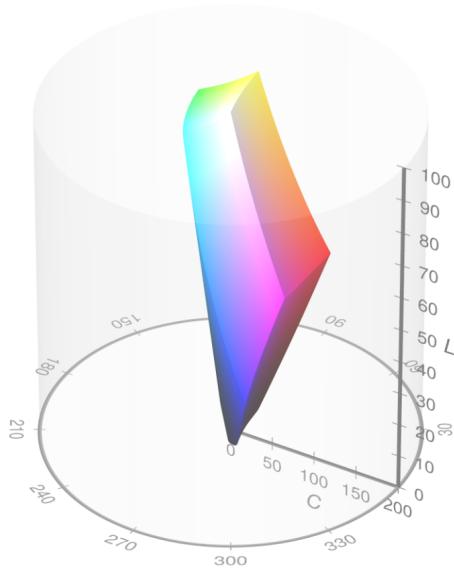
color space - hcl hues

We can see bands of color in the **hsl** hues that look lighter than the other hues (even though they mathematically aren't). Those bands are not visible in the **hcl** spectrum.

If we create a 3d cylinder of the **hsl** color space, we would have a perfect cylinder. But when we visualize the **hcl** color space this way, we can see that it's not a perfect cylinder. This is because humans can't detect changes in saturation for all colors equally.



color space - hsl 3d. From https://en.wikipedia.org/wiki/HSL_and_HSV



color space - hcl 3d. From https://en.wikipedia.org/wiki/HCL_color_space

This makes **hcl** to ideal for creating color scales — if we’re using color to represent different categories of precipitation, those colors should have the same amount of visibility. This will help prevent one color from dominating and skewing our read of the data.

d3-color

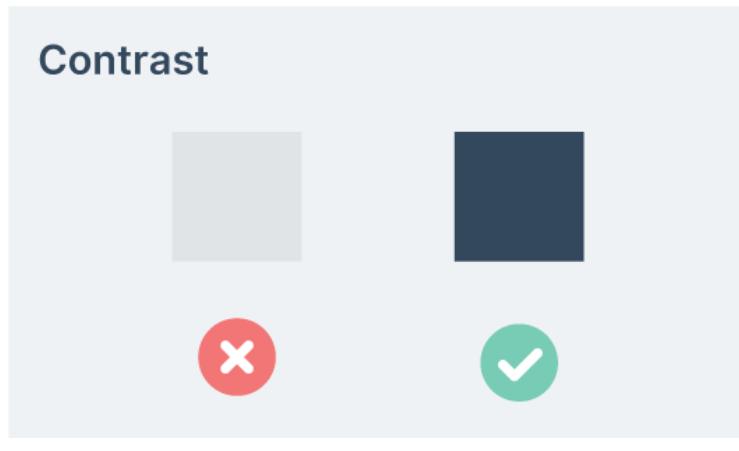
While the browser will recognize colors in `rgb` and `hsl` formats, there isn't much native functionality for manipulating colors. The `d3-color`⁶⁰ module has methods for recognizing colors in the `hcl` format, for converting between formats, and for manipulating colors along the color space dimensions.

We won't go into the nitty gritty of `d3-color`, but it's good to be aware of the library. If you find yourself needing to manipulate colors manually, check out the docs at github.com/d3/d3-color⁶¹.

Color tips

Here are a few tips to keep in mind when choosing a color scale.

Contrast

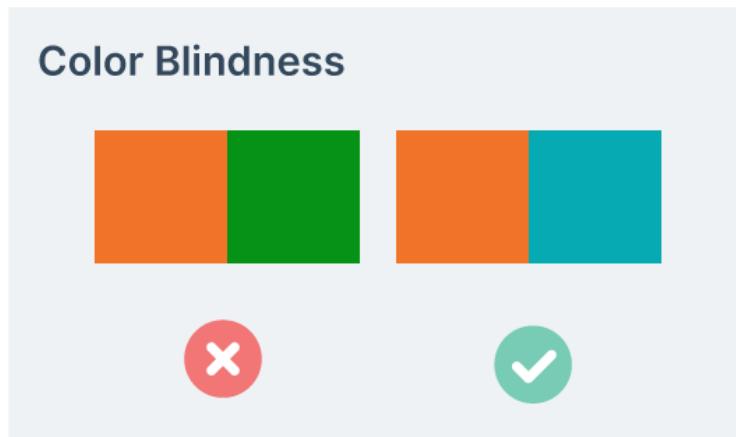


Make sure that your colors have enough contrast with their surroundings in all cases. If you're using Chrome, there is a great tool to check that your colors have enough contrast right in your dev tools. Learn how to use it in the Appendix.

⁶⁰<https://github.com/d3/d3-color>

⁶¹<https://github.com/d3/d3-color>

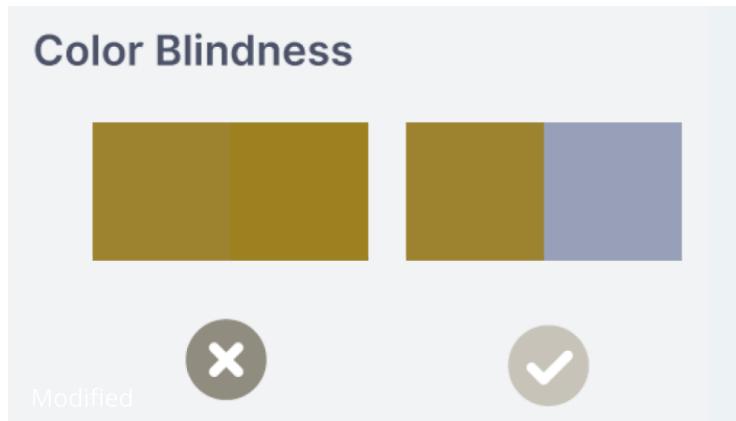
Color blindness



color blindness example

Don't assume that your users can see your charts as clearly as you can. Almost 8 percent of males of North American descent have red-green color blindness. To make sure most people can encode the information from your color scale, stay away from scales where users have to distinguish between red and green.

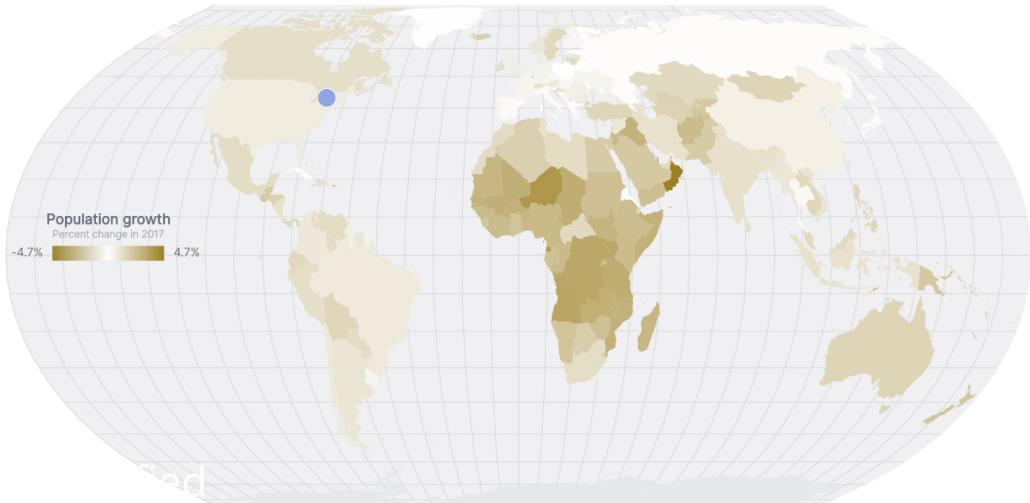
Here's a simulation of what the above picture looks like for people with red-green color blindness.



color blindness simulation example

It's impossible to see which countries have negative growth and which have positive

growth if we use the red/green colors on the left for our color scale on the map we created in **Chapter 6**.



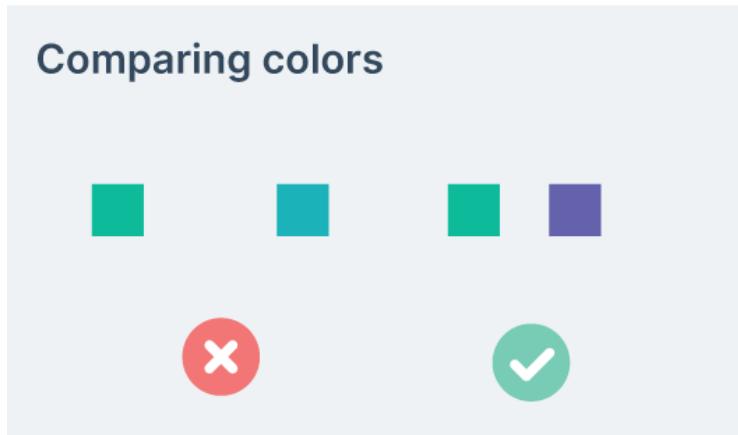
color blindness simulation map

There are other forms of color blindness (yellow-blue and total), but they are less common (under 1% of the population). When creating a chart with a color scale, it helps to have a second way to extract the information, such as adding tooltips or in-place labels.

There are sites where you can simulate different types of color blindness to test your data visualizations, such as [Color Flip⁶²](https://www.canvasflip.com/color-blind.php).

⁶²<https://www.canvasflip.com/color-blind.php>

Comparing colors

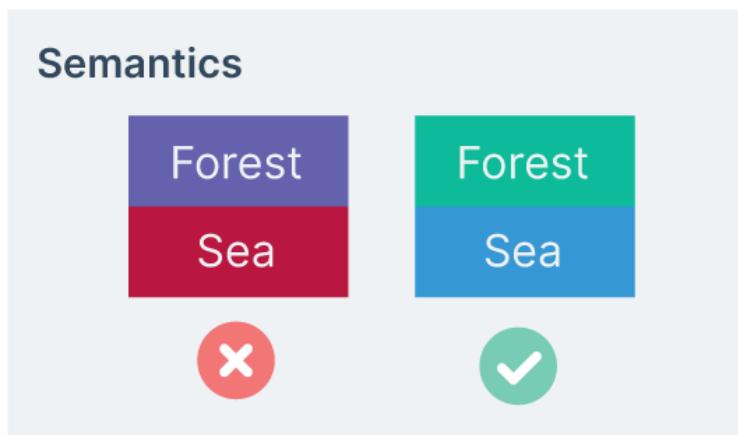


comparing colors example

Smaller areas of color need are harder to compare. When using colors as an indicator, make sure the colored elements are large enough.

Additionally, far-apart areas of color are harder to compare. Make sure the colors you use have enough contrast that users can easily tell the difference.

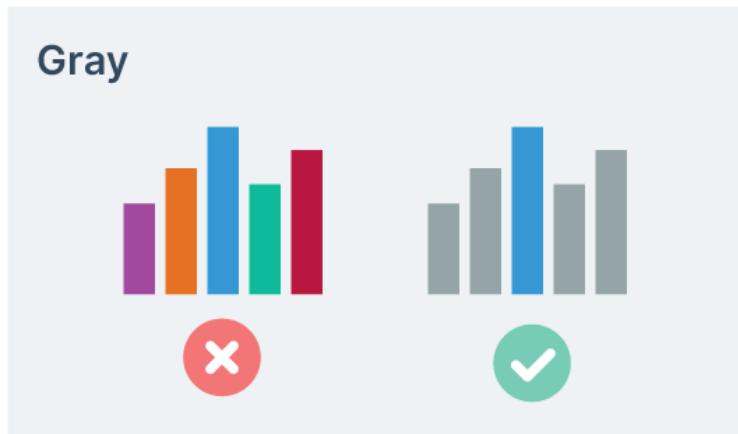
Semantics



semantics example

Choose semantically meaningful colors when possible. For example, blue for lower and red for higher temperatures.

Gray



gray example

Gray can be the most important color. When you want to highlight part of a chart, keeping other elements gray helps viewers focus on that element.

Wrapping up

Great work making it through this chapter! Fundamentals can be tedious to learn, but understanding the bare bones will give you a good foundation to making your own data visualizations.

If you want a handy way to remember these basics, we included a cheat sheet PDF with the advanced package – feel free to print it out or store it somewhere easy-to-find!

Data visualization basics cheat sheet

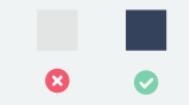
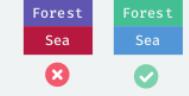
Types of data

Qualitative  Usually a string	Qualitative Binary  EXAMPLE: raining, not raining	Qualitative Nominal  EXAMPLE: sunny, cloudy, windy	Qualitative Ordinal  EXAMPLE: not windy, very windy
Quantitative  Usually a number	Quantitative Discrete  EXAMPLE: 2 tornados, 3 tornados	Quantitative Continuous  EXAMPLE: 70 degrees, 71.4 degrees	

Data dimensions

Dimension Size 	Dimension Position 	Dimension Color 
--	--	---

Data visualization tips

Contrast 	Color Blindness 	Comparing colors 
Semantics 	Gray 	

FULLSTACK D3
and DATA VISUALIZATION
AMELIA WATTENBERGER



Cheat sheet

Common charts

Let's talk about common chart types. This chapter can help you in a few ways:

Communication

When discussing part of your interface, it's important to have commonly understood names for things. Imaging trying to talk about a button without knowing the word "button".

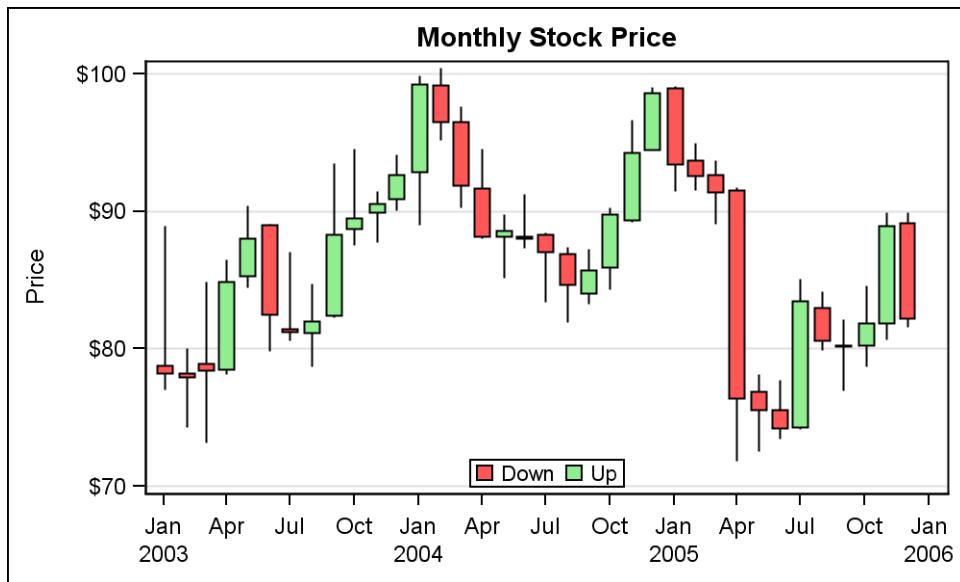
Having a common language for charts makes it easier to talk about existing and potential charts.

Quick readability

Every time a user sees a chart, they need to orient themselves to figure out how to read it. There are ways to help speed up that process, such as adding labels and legends. But the biggest help is using a familiar chart type.

Since most people have encountered a basic timeline before, showing data in a timeline will help users who don't want to spend much time learning a new visualization.

Common charts also have implicit connotations. For example, candlestick charts are often associated with stock data — an association that can speed up (or hinder) your users, depending on the usage.



Candlestick chart from blogs.sas.com/content/graphicallyspeaking/2014/09/27/candlestick-chart/

Inspiration

Common charts are a great jumping-off point for a more complex or unique chart. Only making the basic chart types can seem boring, but thinking about a dataset in the frame of one of these charts helps with brainstorming. What insights might your readers glean if you plotted your data as a histogram? What about a radar chart?

Additionally, sometimes the most basic chart is the best suited for a new feature.

Chart types

Let's go through some of the common types and discuss examples of each.

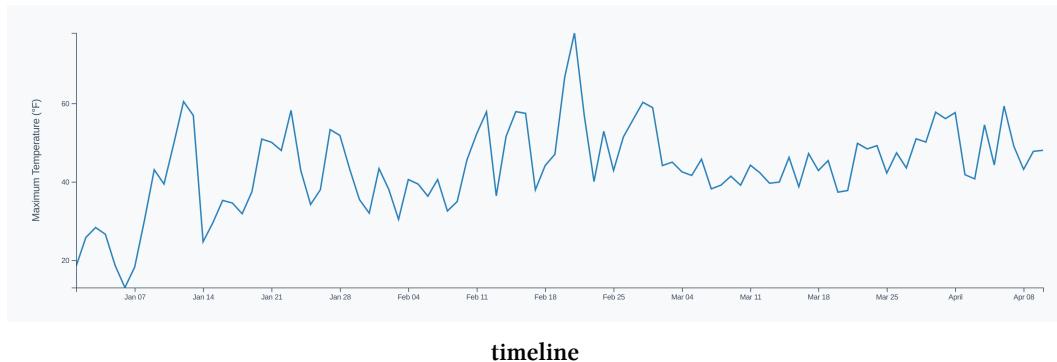


Each of these chart types has corresponding code in the `/code/08-common-charts/` folder. Be sure to check them out live in the browser and dig through the code. I guarantee you'll learn a lot from going through each example.

Please use the code as a resource — next time you want to implement a similar chart, the code will be here to guide you, using the same framework we've used throughout this book.

Timeline

Most dashboards have at least one timeline.



A timeline shows a **continuous** metric (y axis) across time (x axis).

For each of these charts, we'll practice breaking it down into its component parts — what kinds of metrics are we visualizing and in what dimensions are they visualized?

Dimensions: - vertical position of a point on the line represents a **continuous** metric value
- horizontal position of a point on the line represents a **continuous** metric value (**time**)

They are very useful for showing trends over time. For example: has this stock value increased over the past year? Are there weekly patterns in this stock value?

An important question to answer when creating a timeline is: what granularity do I want to see? A timeline that goes through one point per day can be too noisy when looking at yearly data. It's hard to pick up an overall trend when looking at a spiky chart.

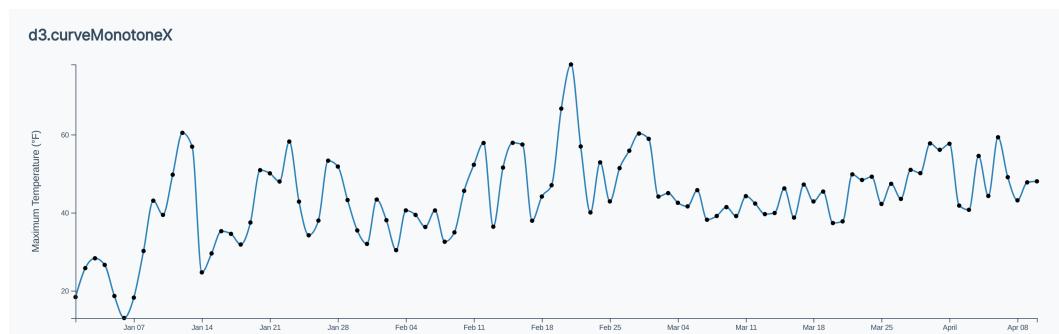
Alternatively, a large granularity can mask small outliers — showing monthly points can hide a day with a large value.

Another question to ask when creating a timeline is: how smooth do I want my line to be? If your audience wants to know the exact points of a timeline, stick with a linear interpolation (the default for `d3.line()`).

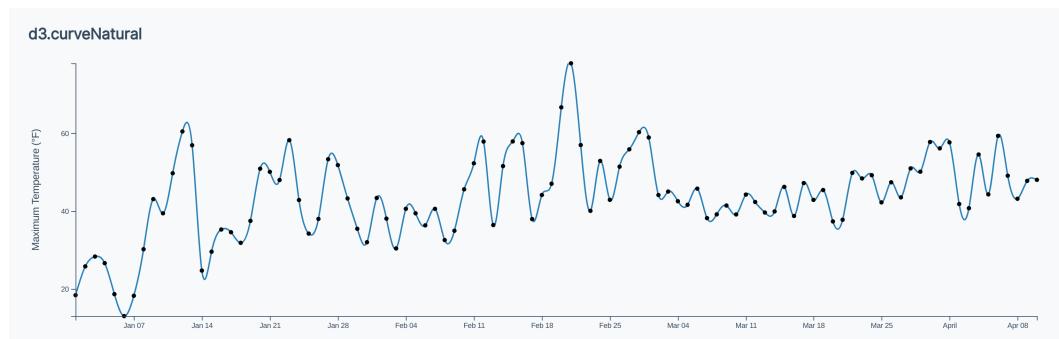
A smoother line can help with showing the bigger picture — in a sense, we're guessing what the missing values are between points.



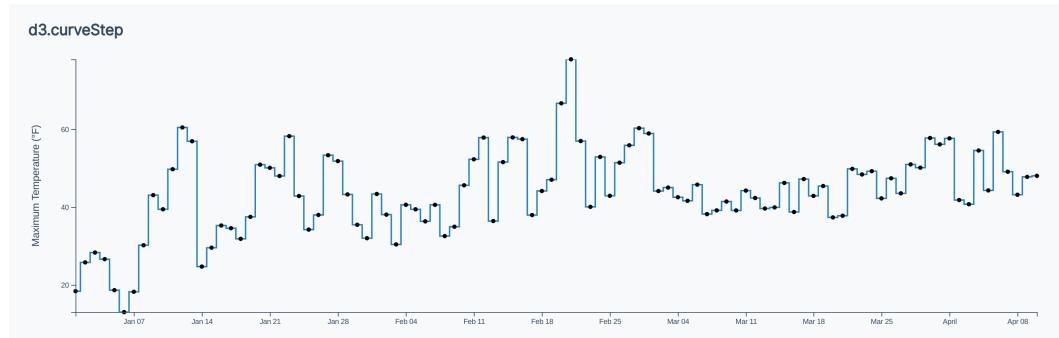
We can change the smoothness by using different interpolations and passing them to `d3.line().curve()`. Here is our line with different interpolations (the original points are visualized as dots):



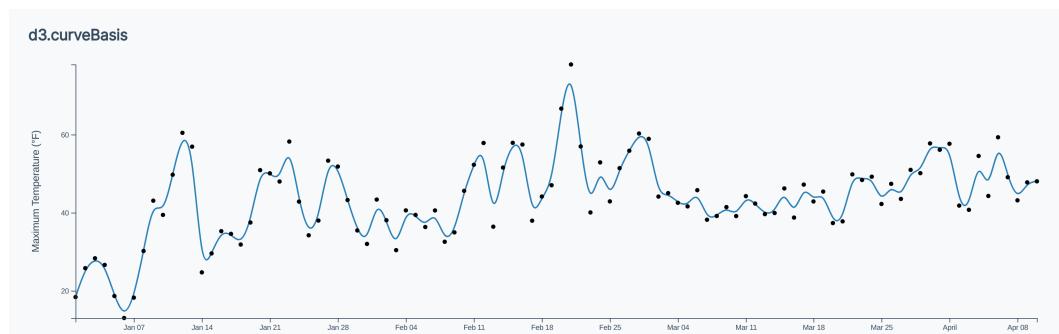
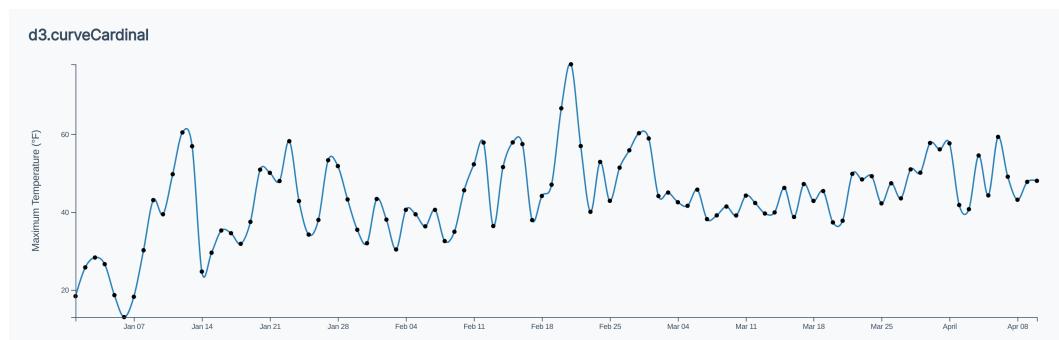
timeline with d3.monotone() interpolation



timeline with d3.curveNatural() interpolation

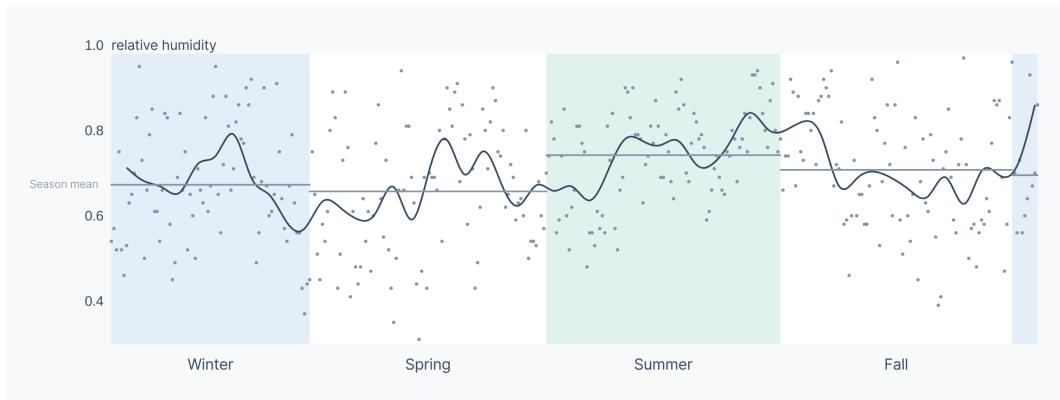


timeline with d3.curveStep() interpolation

timeline with `d3.curveBasis()` interpolationtimeline with `d3.curveCardinal()` interpolation

In general, `d3.curveMonotoneX()` is a good bet since it crosses through each point. With other interpolations, such as `d3.curveBasis()`, the line will “skip” points to create a smoother line, which can make adding tooltips tricky, and also won’t ensure that your line will stay within your bounds.

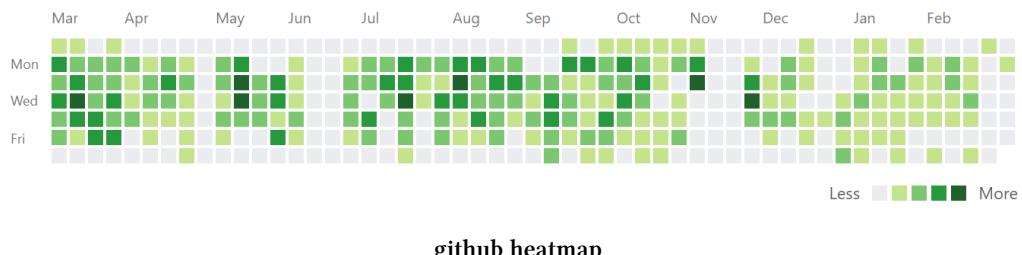
One thing to be cautious about is lying with data — make sure you aren’t misleading the reader into thinking that you have more data points than you do. One way of denoting a *trend* is by showing the original points as well as a trend line, like we did in the last chapter.



Humidity timeline, finish

Heatmap

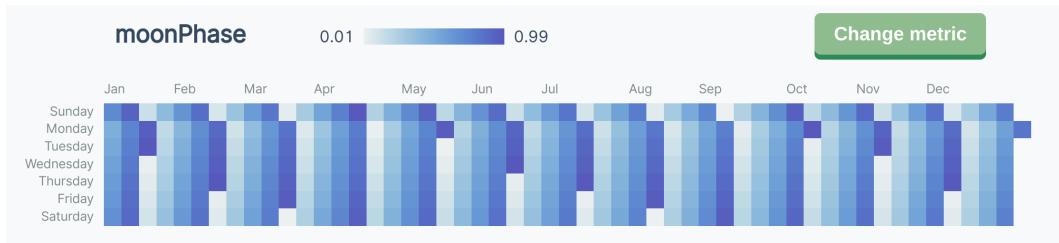
You might be most familiar with heatmaps from the Github contribution graph.



A heatmap has three dimensions: x, y, and a color scale. In this case, the x axis corresponds to week, the y axis corresponds to day of the week, and the color scale domain is the number of contributions per day.

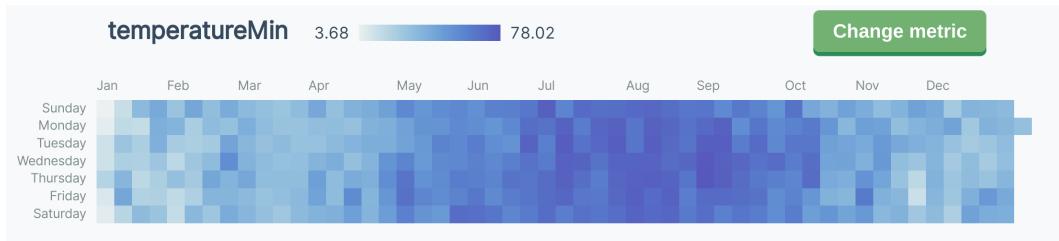
Dimensions: - vertical position of a square represents the day of the week (**discrete**)
 - horizontal position of a square represents the week of the year (**discrete**) - color of a square represents a **continuous or discrete metric value** (often using a **sequential color scale**)

In our code library's heatmap, we can cycle through different metrics to represent in our color scale. For example, the moon phase heatmap shows a very clear 27-day cycle.



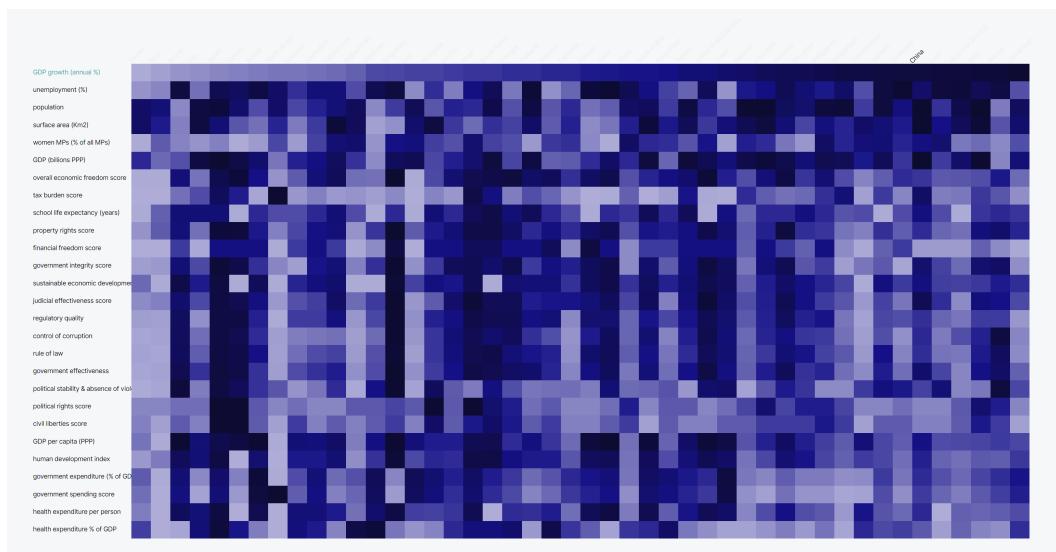
heatmap of moon phase

Alternatively, if we look at the minimum temperature heatmap, we can see the much more gradual change of temperature between the four seasons.



heatmap of min temperature

Heatmaps don't necessarily need to show one metric over time. For example, the following heatmap shows percentile for Asian countries (x axis) for different metrics (y axis). Note that heatmap cells don't necessarily need to be square.

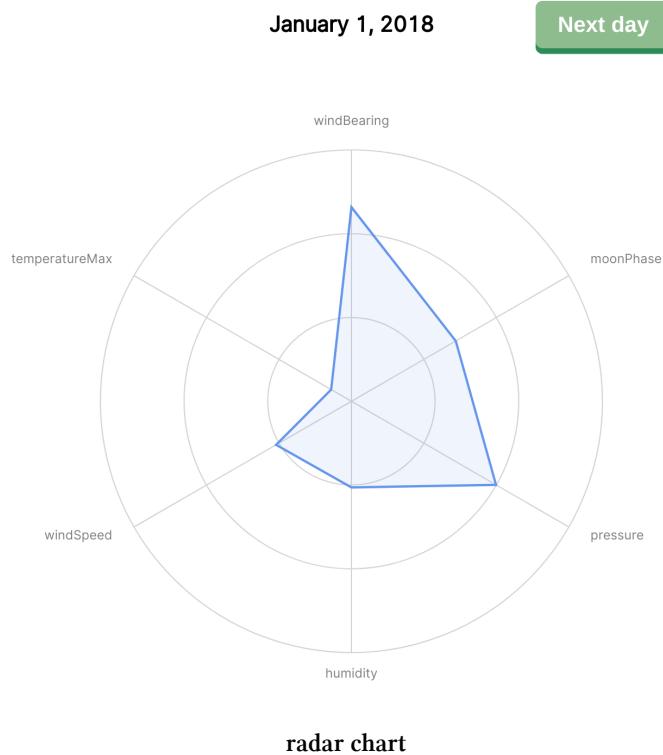


country metrics heatmap

Heatmaps are great at showing trends over time and correlations between metrics. It's harder to tell small differences in color scales than, say, bar sizes, so stay away from them or add informative tooltips if you need to show exact numbers.

Radar

Radar charts are not the most common chart type, but they can be very useful when looking at multiple metrics.



radar chart

They consist of at least three axes, where each axis represents one metric.

Dimensions: - position of a point on the line on each spoke represents a **continuous** or **discrete** metric value

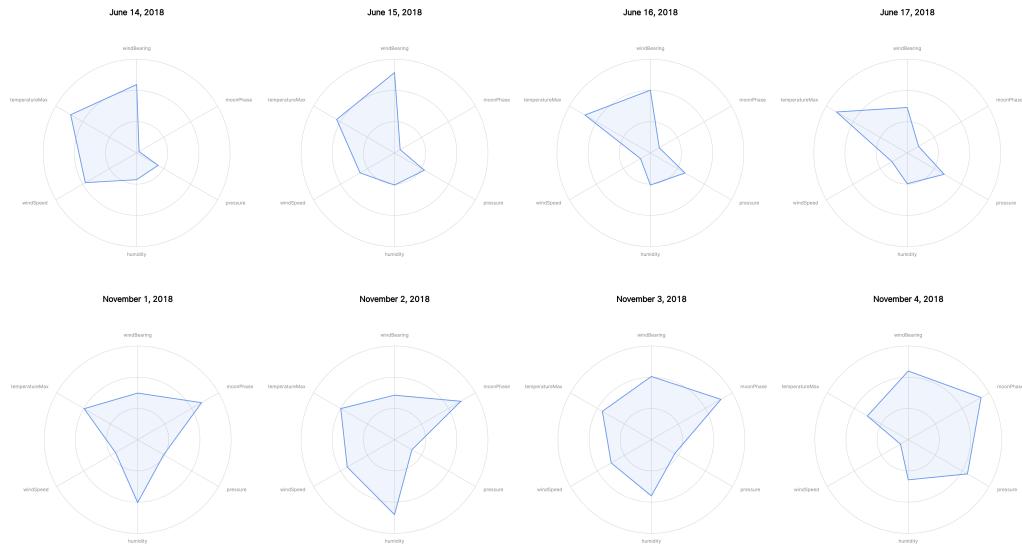
While they aren't very helpful for making exact measurements (it's hard to tell how much larger variable A is on chart 1 versus chart 2), they really shine when comparing many charts with many metrics. Each chart creates a *profile* of that instance, and multiple *profiles* can be easily compared to find similarities and outliers.

There are a few ways to present multiple *profiles*:

1. **Show multiple polygons of different colors, all on the same chart.** This works well when there are fewer than five *profiles*.
2. **Show one chart with a toggle between *profiles*.** This works well when you want to focus on one *profile* at a time, potentially with more information alongside the chart.

3. Show multiple radar charts. This works well when comparing more than five *profiles*, with no extra information.

If we look at a group of radar charts for our weather data, we can pick out patterns.



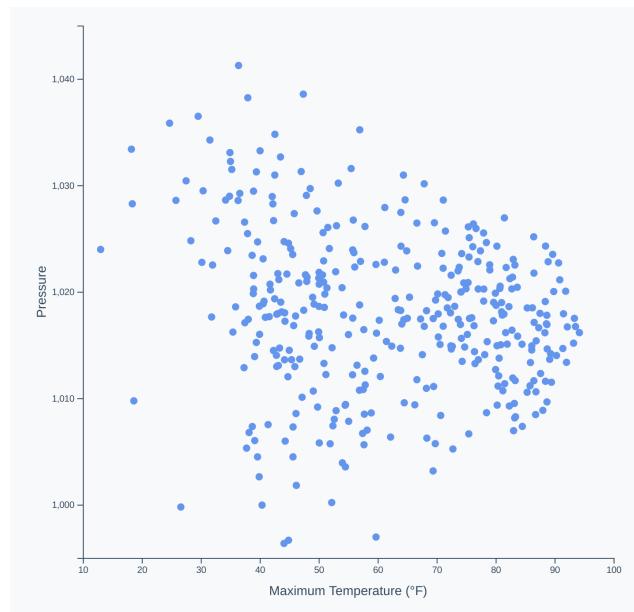
radar charts, small multiples

For example, days in June have a common pattern (up and to the right) and days in early November have a different pattern (mostly even all around). Additionally, we can see that November 4th is a bit of an outlier, with an unusually low wind speed.

These patterns seem obvious, given what we know about seasons — winter days have different weather than spring days. But we can dive in deeper to see more granular patterns — for example, the second set of days are windier and less humid. Additionally, we know a lot about how the weather changes between seasons. Imagine looking at radar charts for less familiar data — radar charts would help us quickly pick up patterns across many metrics.

Scatter

Scatter plots are great for looking at the relationship between two metrics (x and y axes).



scatter plot

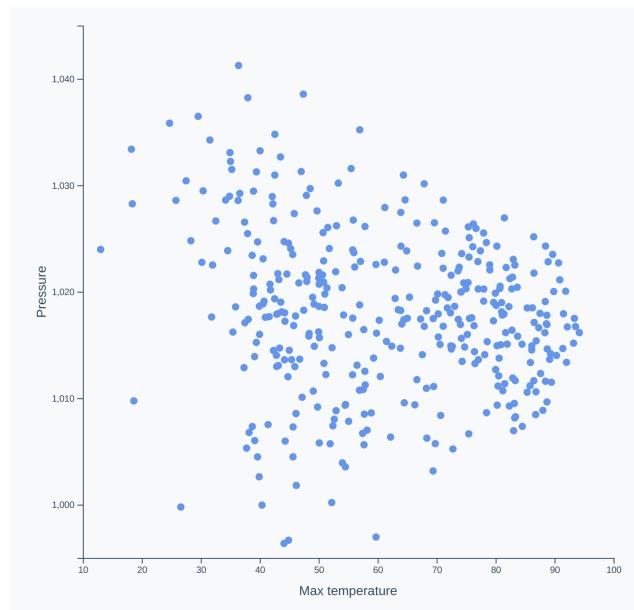
Dimensions: - horizontal position of a dot represents a **continuous** metric - vertical position of a dot represents a **continuous** metric

They work best when the chart is square, giving the same **range** to both metrics.

There are three basic patterns to look for in a scatter plot.

Positive

The two metrics are correlated and increase together in a linear fashion.

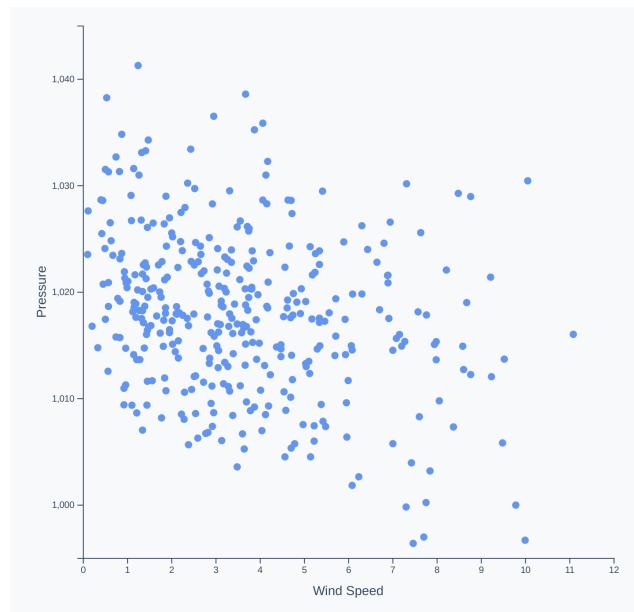


scatter plot, positive

For example, a scatter plot of minimum temperature and maximum temperature will have a positive pattern, since temperature varies much more across days than within days.

Negative

The two metrics are anti-correlated and one decreases while the other increases.

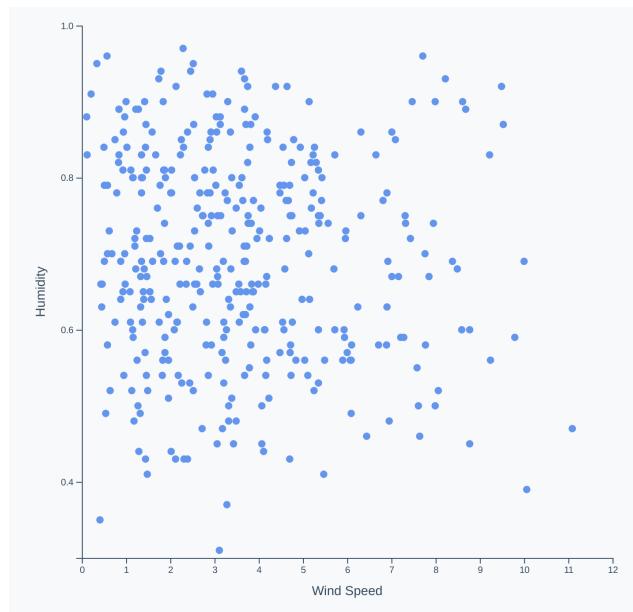


scatter plot, negative

For example, a scatter plot of wind speed and pressure will have a negative pattern, since windier days tend to have less more pressure.

Null

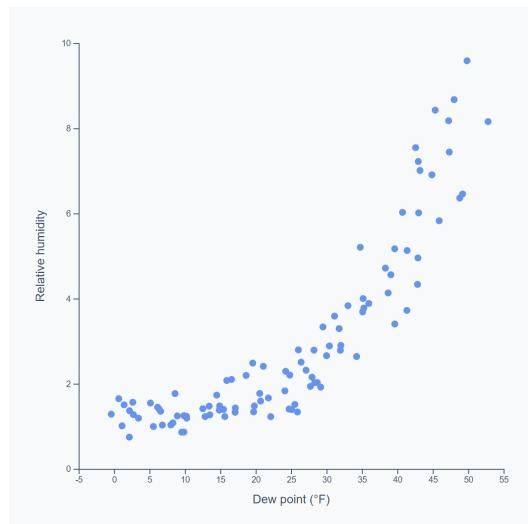
The two metrics are not related.



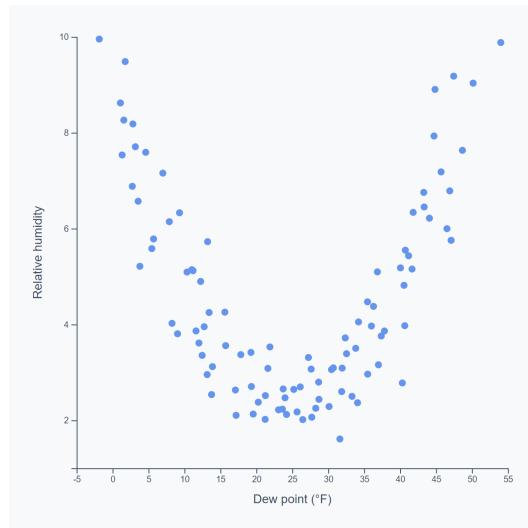
scatter plot, null

For example, a scatter plot of wind speed and humidity will have no pattern, since wind speed and humidity are not at all correlated.

There are many, more complex patterns that might be visible in a scatter plot. For example, scatter plots can be **exponential** or **u-shaped**.



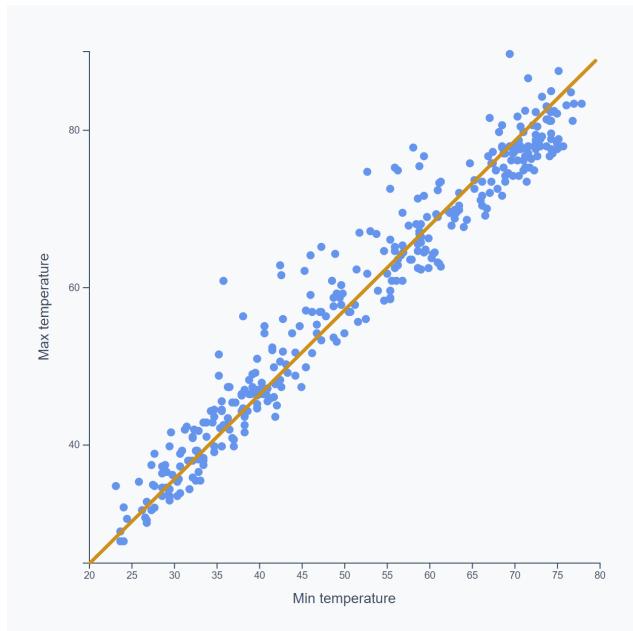
scatter plot, exponential



scatter plot, u-shaped

These kinds of patterns are less common and harder to interpret.

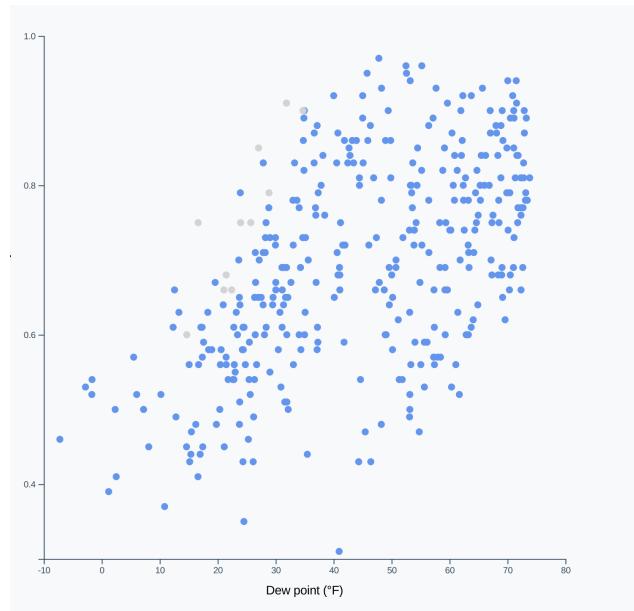
A scatter plot might also have a **line of best fit** that is the best guess of how the two metrics are correlated.



scatter plot with line of best fit

When looking at a scatter plot, keep one of the golden rules of data analysis in mind: **correlation does not imply causation**. Even if a scatter plot has a strong positive pattern, you cannot conclude that metric A causes metric B, or vice versa.

Sometimes an extra dimension is added to a scatter plot — a color scale or different shapes per category. For example, we gave different kinds of precipitation a different color in [Chapter 2](#).

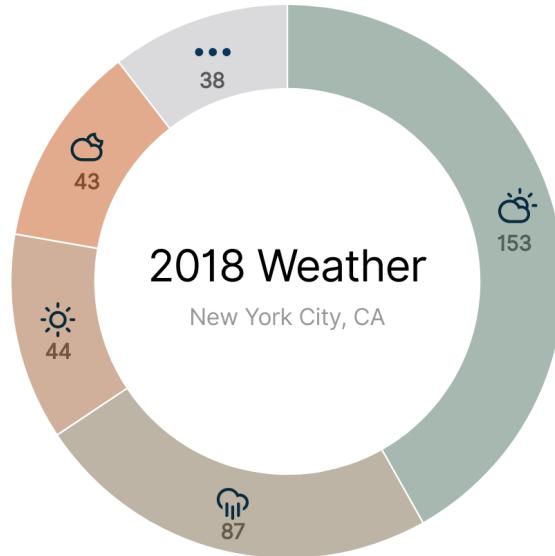


scatter plot with precipitation

This can help identify groups within your dataset.

Pie charts & Donut charts

Here is one of the most controversial types of charts — the pie chart! A pie chart and a donut chart are very similar — a donut chart is just a pie chart with the middle cut out.



Icons from [Climacons](#) by Adam Whitcroft

donut chart

Dimensions: - angle of a slice represents a **continuous or discrete metric**

I bet you've heard lots of opinions about pie charts. Part of the controversial stigma comes from their overuse in inappropriate situations, or the popularity of showing them in 3d. I have yet to find an appropriate use for a 3d pie chart.

That being said, pie charts can be useful — mainly because they are intrinsically tied to the idea of “parts of a whole”. We’ve all cut a pie or a pizza into multiple parts before, while focusing on creating equally-sized slices. When a user sees a pie chart, it’s immediately obvious that they are looking at the composition of a whole.

However, because of human perception, it’s way easier to compare bar lengths than slice sizes. In most cases a pie chart is used, another chart type would have been more effective. Here are a few tips for making effective pie charts:

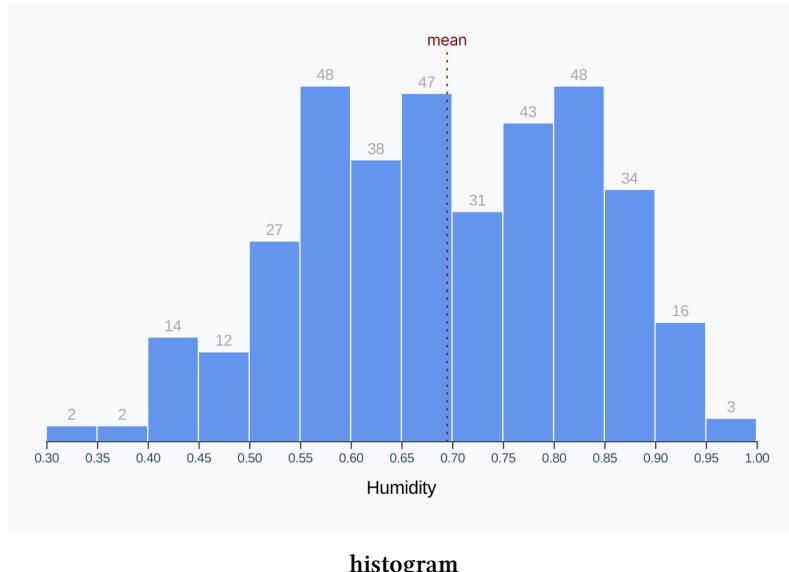
- Restrict the number of slices to five or fewer. Note that we combined the last few slices into an “other” category to keep the focus on the larger slices.

- Order the slices by size
- Label each slice directly

Make sure to check out the code example in `/code/08-common-charts/pie/` folder to see d3's arc-drawing functionality in action!

Histogram

We're already familiar with this next chart type — we made our own histograms in **Chapter 3**. Histograms are unique in that they are only concerned with one metric.



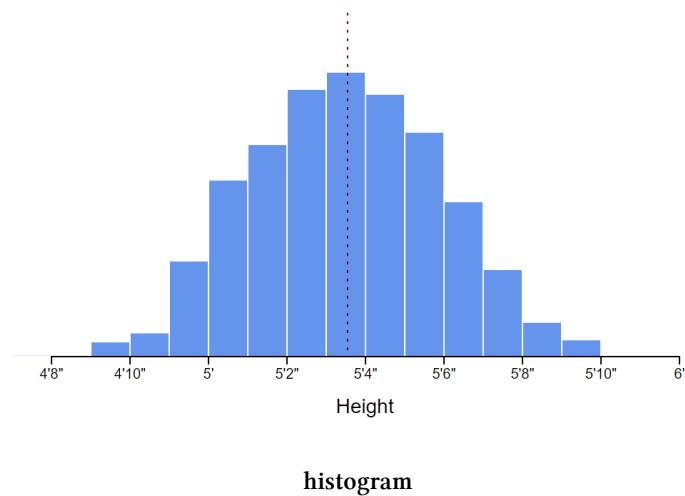
Dimensions: - horizontal position of each bar represents a **discrete metric** (usually a binned **continuous metric**) - height of the bar represents a **discrete metric** (count of the data points that fall in the bucket)

Use a histogram when you want to look at the variation of values for a single metric. Do we have mostly high or low temperatures, or does temperature vary closely around one value?

There are five basic types of distributions:

Normal

A histogram with a normal distribution is centered around a mean. Usually, this data . Most metrics that rely on chance end up with a normal distribution, such as per-sex height.

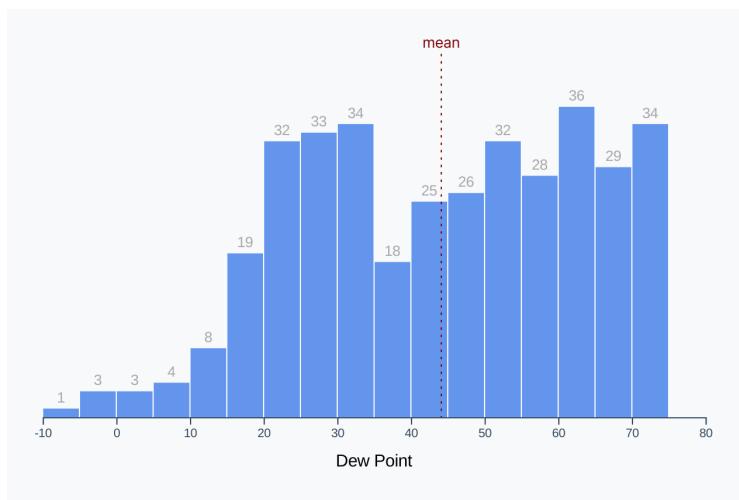


histogram

For example, the range of heights in an all-female class is likely to be a normal distribution.

Skewed right

In a histogram that is skewed to the right, a value is more likely to lie in the upper half of values.

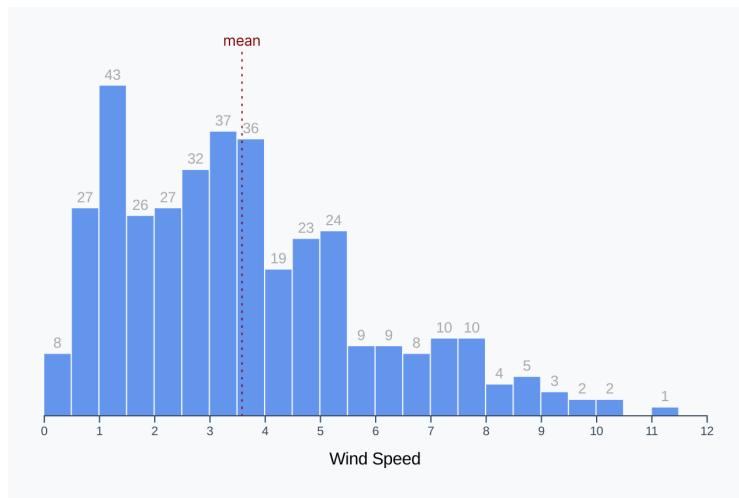


histogram - dew point

We saw an example of a skewed right histogram in [Chapter 3](#) when we looked at dew point. This histogram is only slightly skewed to the right, but you can see the tail off to the left.

Skewed left

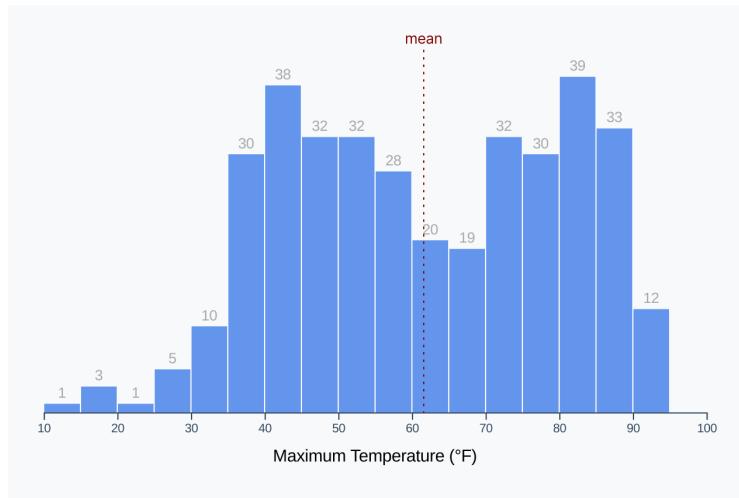
The opposite of **skewed right**, in a histogram that is skewed to the left, a value is more likely to lie in the lower half of values. This often happens when a metric has a lower bound, since values can't dip below a certain number.



histogram - wind speed

Bimodal

A bimodal histogram has two clear peaks.

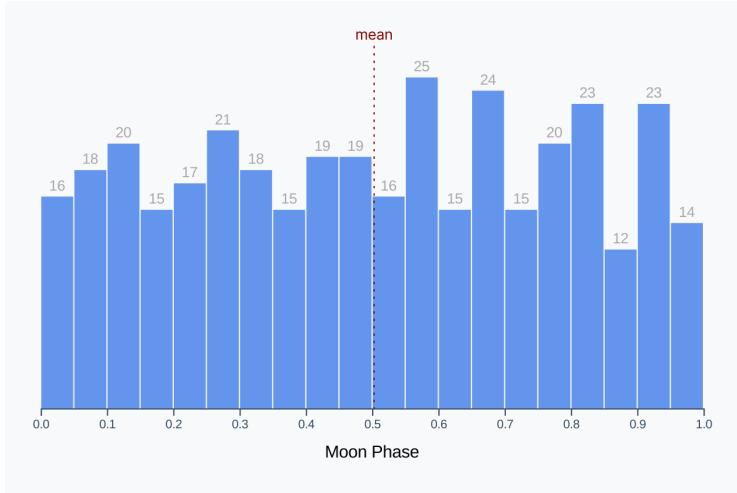


histogram - min temperature

We saw an example of a bimodal histogram in **Chapter 3** when we looked at max temperature. The daily temperature seems to either be around 45 degrees or 80 degrees - representing the extremes of winter and summer.

Symmetric

A symmetric histogram no clear peaks and looks fairly flat.



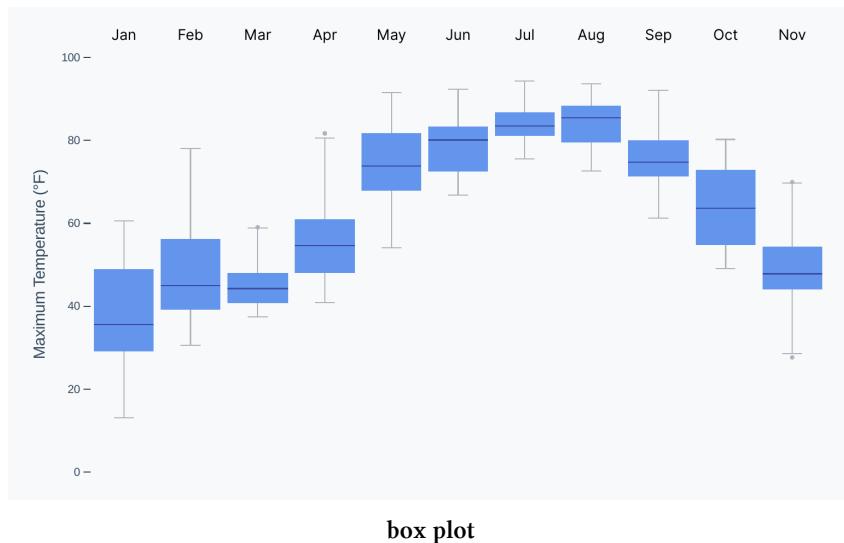
histogram - moon phases

We saw an example of a symmetric histogram in [Chapter 3](#) when we looked at moon phase. The moon phase is cyclic, with the value incrementing equally every day, so every value is equally as likely.

Try to revisit your histograms from [Chapter 3](#) and see which category each of your metrics' distributions falls under.

Box plot

Box plots are great for showing discrete or binned continuous metrics — usually over time or relative to another metric.

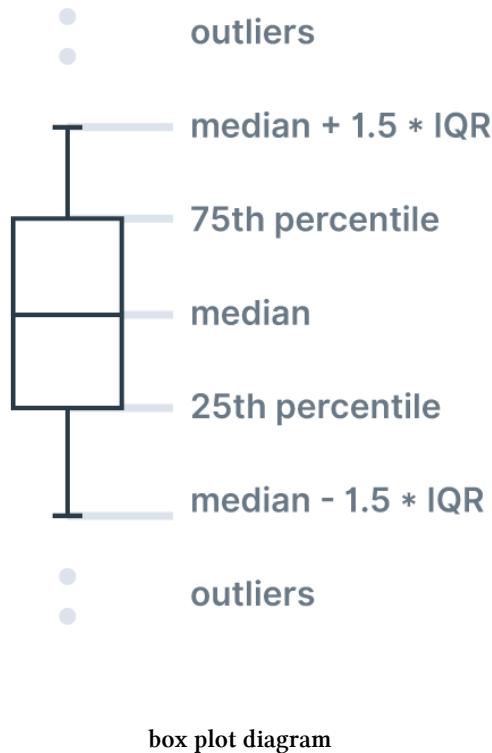


box plot

Each **box** shows the distribution of values for a category. For example, the first **box** in this chart shows the range of all humidity values in January.

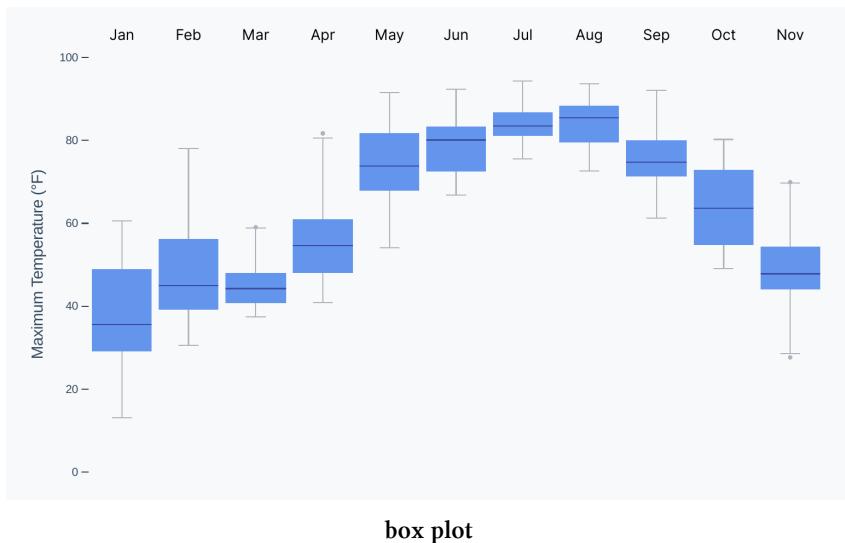
Dimensions: - **horizontal position** of each box represents a **discrete metric** - **vertical position** of each box and whisker represents a **continuous metric** - **height** of each box and whisker represents a **continuous metric**

The middle line represents the median (the middle value) and the box covers values from the 25th percentile (25% of values are lower) to the 75th percentile (75% of values are lower). The “whiskers” extend 1.5 times the inter-quartile range (**IQR**) from the median in either direction. By **IQR**, we mean the difference between the 25th and 75th percentiles. Any outliers that lie outside of the whiskers are shown as dots.



Note that there are different conventions for where to place the ends of the whiskers. For example, some charts cover all data points with the whiskers, whereas some charts cover one standard deviation above and below the mean. Our definition of **1.5 times the IQR from the median** is a good middle-ground that will show the variation, but will also be resistant to outliers.

Now that we understand the different parts, let's take another look at our weather box plot.



box plot

By looking for outlier dots, we can see that both March and April had two abnormally hot days. Looking at the whiskers, we can see that there is no overlap between max temperatures in March and June. Additionally, the maximum temperatures in March are all very similar, whereas February has a lot of variability.

If we were just looking at the mean or median temperature of each month, we wouldn't be able to make any conclusions about the range of values. A box plot has added complexity and can take more practice to read, but packs a lot of information about variability.

Box plots are more complex than the charts we've created before — be sure to check out the code in the `/code/08-common-charts/box-plot/` folder. Note that we create a `<g>` element for each of the months' elements to keep them together and make things easier with our data binding enter/exit pattern.

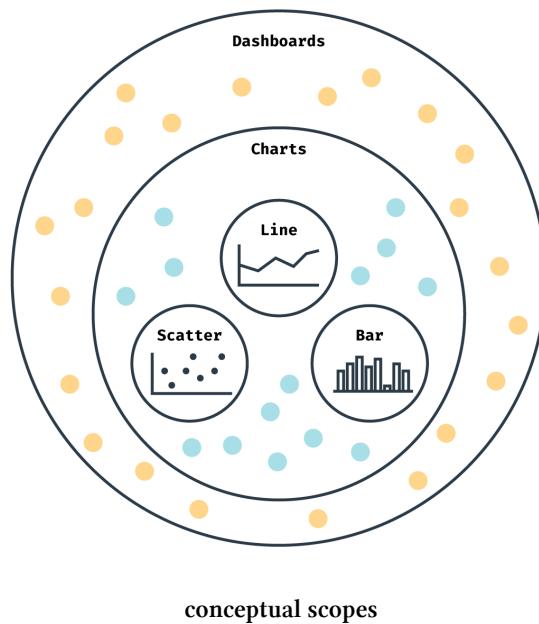
Conclusion

This is just the tip of the iceberg. There are many more types of charts and many chart types that have yet to be discovered! Remember to use this chapter as a starting off

point, or as a source of inspiration. And make sure to look through the code for each example or try to recreate the chart on your own to solidify your skills.

Dashboard Design

So far, we've talked about visualizing data at the **chart** level. Let's zoom out another level and talk about how to design effective **dashboards**.



What is a dashboard?

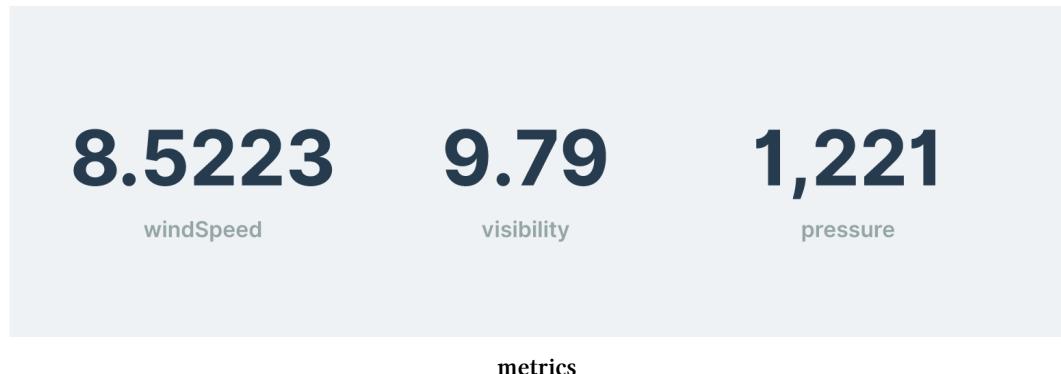
Good question! There are many definitions of “dashboard” - in this chapter, I’ll be using the word to refer to any web interface that makes sense out of dynamic data. This is by no means the official definition of **dashboard**, but it is a handy definition for our use here.

We’ll talk about ways to display a simple metric effectively, how to deal with different data states when loading data, how to design tables, and then we’ll run through a

design exercise. You'll come away with tangible strategies for displaying data in an actionable manner. Let's dig in!

Showing a simple metric

Let's say we have an overview screen and we want to display the three most important metrics. The straightforward solution would be to show numbers with the name of the metric, right? Maybe we'll format the number nicely and call it a day.



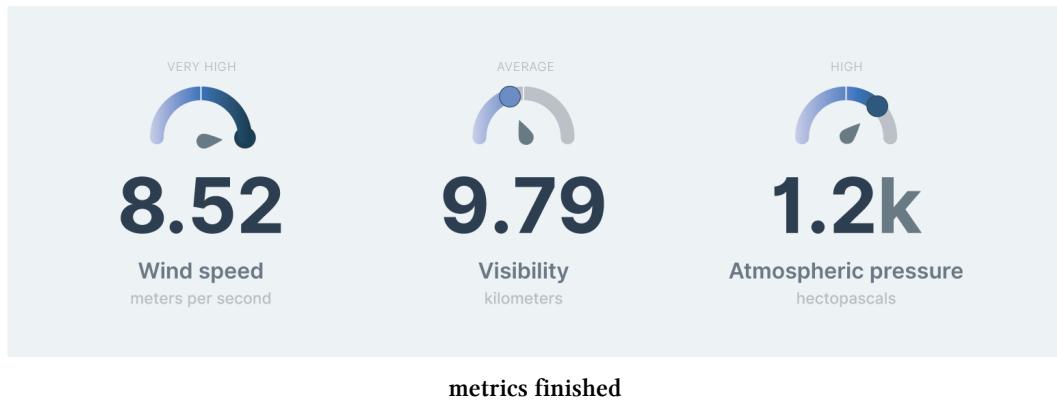
This certainly works! Your users can read the numbers and go from there. But what does “go from there” mean? Let’s try to anticipate some questions that need to be answered in order to convert these numbers into insights.

- What units are these numbers in?
- Is 8.52 meters per second fast? How does this number compare to other values?

If you take anything away from this chapter, let it be that **context is king**. A number is meaningless if we don't know:

1. What it means — are we looking at miles per hour or meters per second?
2. How it compares — is this an average value, or is it abnormally high?

Here is a more useful view of our metrics:



We've added a gauge on top of our numbers — the range could be either:

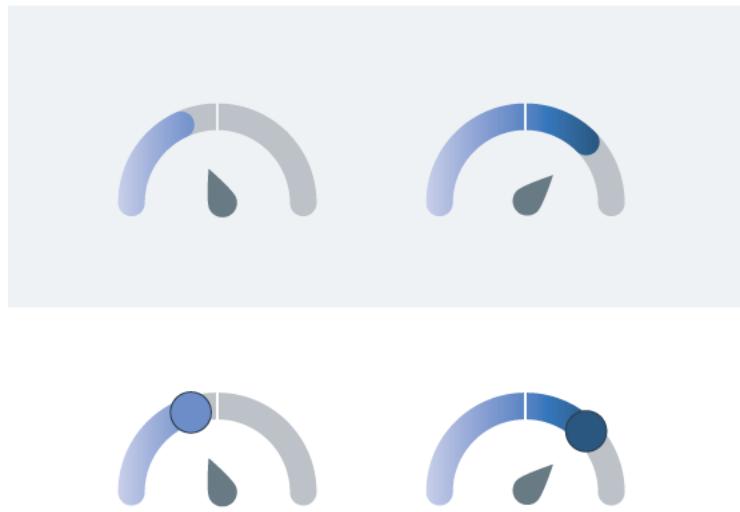
- the total range of possible values, or
- centered around the mean or median and extending two standard deviations in either direction.

The gauge lets the user know how the value compares to other values. Almost as important, though, is letting the user “read” the metric more quickly.

Both the position and the color of the bubble give a fast impression of how the metric compares. A lighter or more left-ward bubble means a low value — that's easy to scan for.

Adding gauges also makes metric comparisons easier — the **atmospheric pressure** is high, but not as abnormally high as the **wind speed**.

But why did we add bubbles to the gauges? Let's take a look at the gauges with and without a bubble. Notice how quickly you can scan both sets.

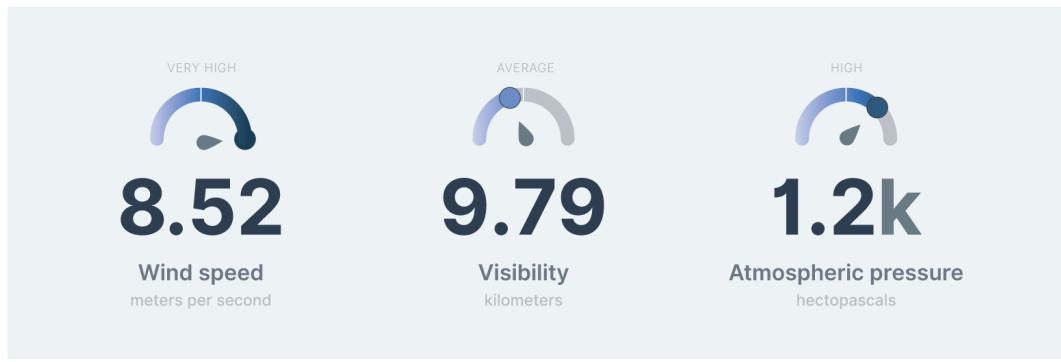
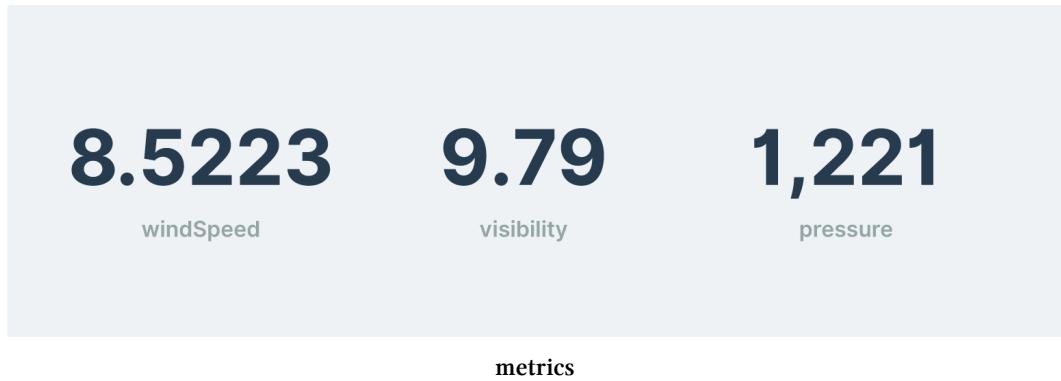


gauges with and without bubbles

With the background gradients, we're a little overwhelmed with colors — it's hard to isolate the color of the actual value. Sometimes adding an extra, redundant element helps with focusing the user on what matters.

There are many types of gauges and visual indicators that can help with quickly interpreting numbers. This gauge is a simple example — put into terms we learned in Chapter 7, we're representing the metric value with both a **position** and **color** dimension. Think of some other ways we could represent such a value, and what the pros and cons would be of each choice.

Let's look at the other changes we made to make our metrics easier to process:



We added categorical labels

For example: **very high**, **average**, etc.

Adding descriptive language primes your user with the correct language to use. Sometimes users aren't comfortable making their own conclusions from the data, and turning a chart into words can go a long way towards giving them confidence.

We formatted the numbers to provide just enough information

Figure out what specificity your users will need (eg. two decimal places) and don't show more than that. If necessary, you can add some kind of progressive disclosure, such as showing the full value in a tooltip or in an export.

We dimmed less important information

Decorators such as % or \$ can be dimmed since they're not as important as the numbers themselves. This is especially true when comparing metrics - 3 is the

important part when comparing \$3 to \$2.

We added units

For example, **meters per second**, **kilometers**, etc.

Even when it seems obvious what a number represents, more contexts is better. Having units everywhere you display a number also helps keep your dashboard shareable — ask yourself: “Would a screenshot be self-explanatory?”

We added human-friendly labels

It's a good idea to centralize a mapping of metric **keys** (to access the numbers) and your **labels**. Any person would rather read **Wind Speed** than **windSpeed**.

Another perk of a centralized mapping is that your metric names will be consistent throughout your dashboard.

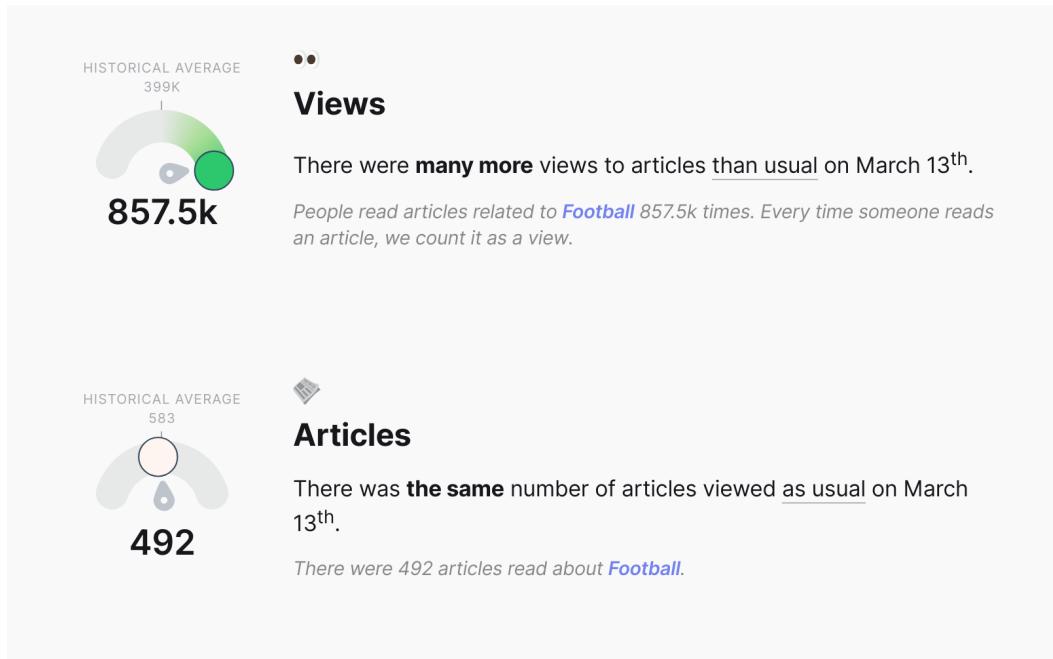
When adding a metric to your dashboard, consider all the ways you can provide context. Would it help to show:

- what the past values have been? Showing where this value lies in a distribution could give the user lots of information.
- what the value was one week ago?
- what the value is in another location?
- what the value is likely to be in the future?

Another way to make simple numbers more insightful is to combine them into a new metric. For example, I was recently working on a dashboard that showed two things:

1. how many times people read an article about a topic, and
2. how many articles were read about that topic.

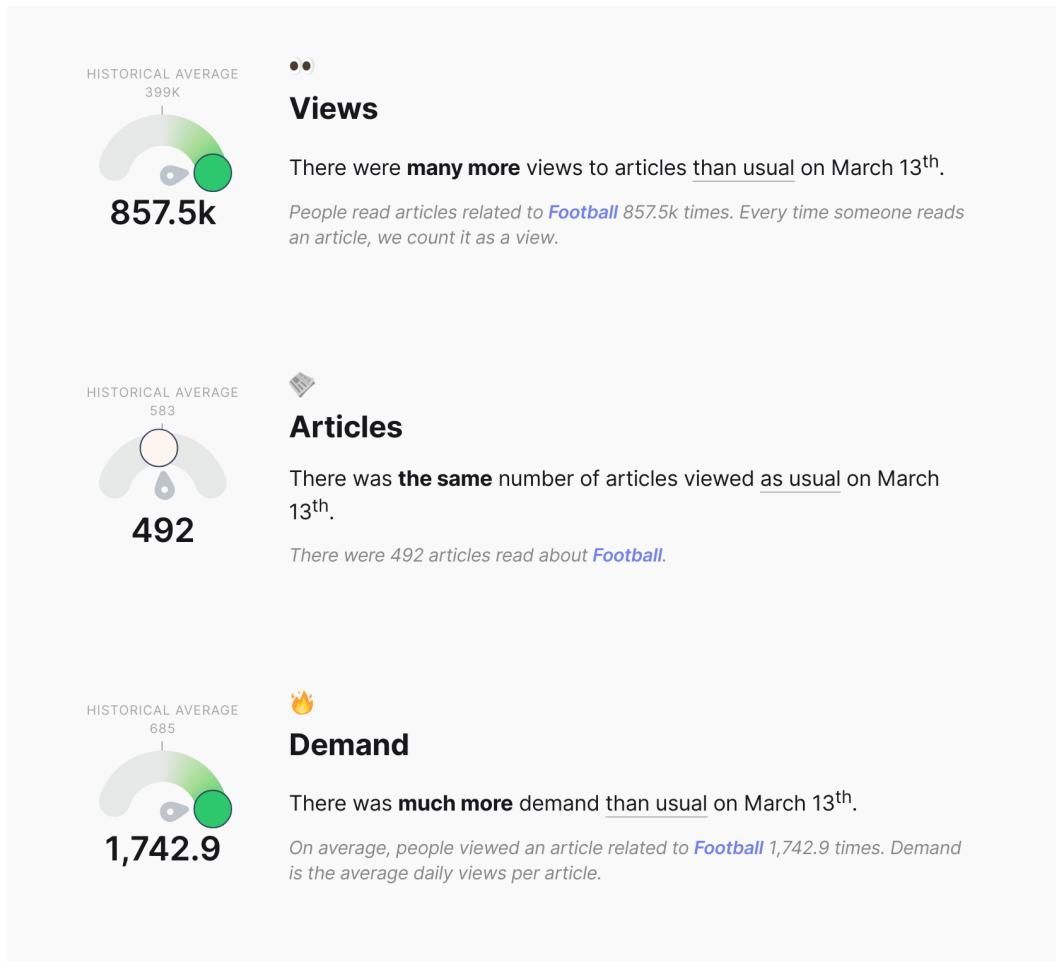
Here they are for the topic **Football**.



views and articles viewed, from the Parse.ly Currents dashboard

Even with the historical context, these values are not very actionable. You might think “Wow, a lot of people are reading about football”, then move on with your day.

In order to change **numbers** into **insight**, we created a new metric — **views per article**.



views, articles viewed, and views per article, from the Parse.ly Currents dashboard

Now, a journalist could see this and think “Wow, articles about football are getting a lot of attention (relative to other topics), they must be high in demand!” That’s actionable — the journalist can take that information and write that football article they’ve always wanted to write, knowing that it will interest more people than another topic.

Dealing with dynamic data

Building a chart with static data is a great feat — you need to wrangle your data, position all of the elements, and create an effective design. When creating a dashboard, the amount of work at least doubles. Let's talk about some concerns that come up when creating dynamic charts and how to solve them.

Data states

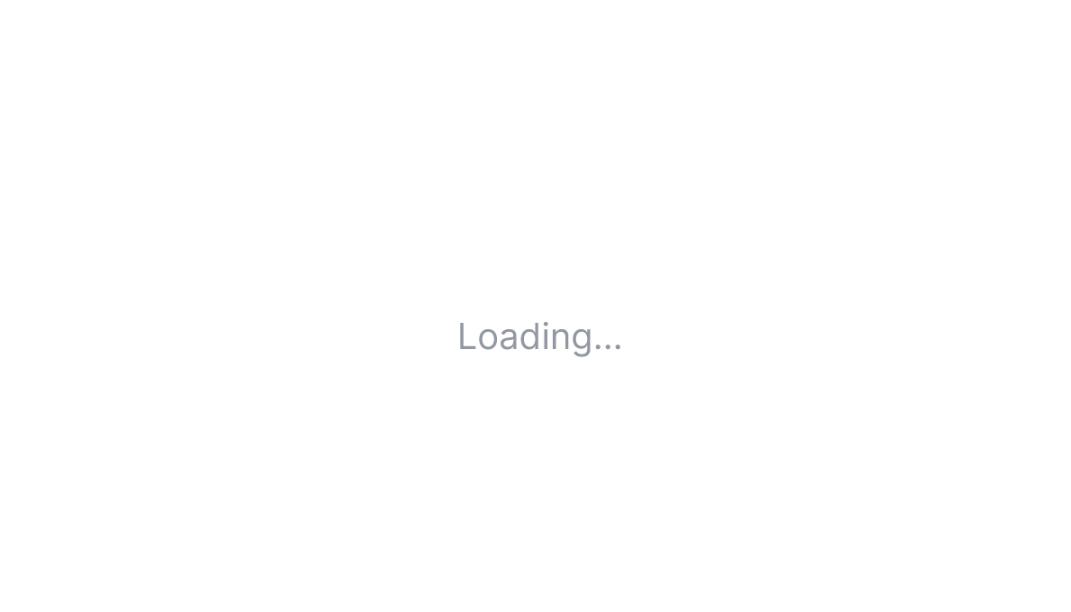
In addition to the final design, our dashboard charts will need to handle multiple states:

- loading
- loaded
- error
- empty

But what should our charts look like during each of these states?

Loading

The simplest solution for a loading chart is to display `Loading...` or a loading indicator.

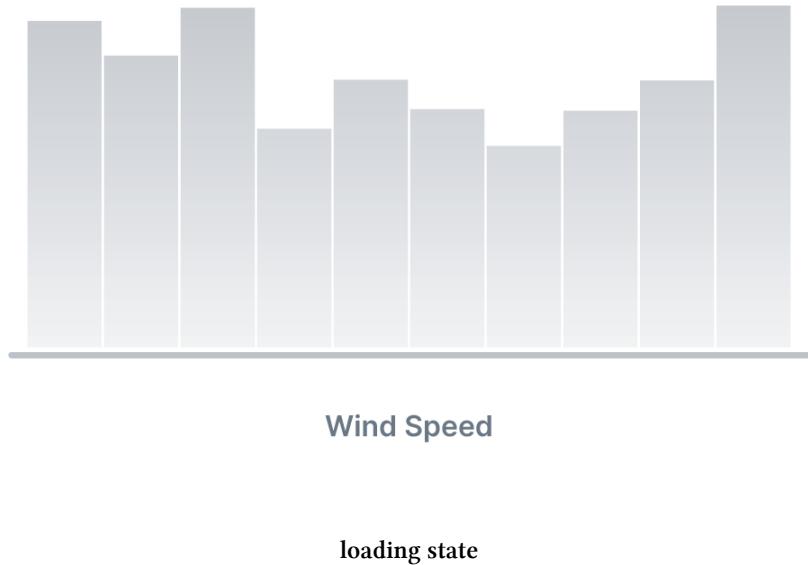
A large, light gray rectangular placeholder occupies the upper two-thirds of the page, centered horizontally.

Loading...

loading state (minimal)

While this is better than nothing, we can give our users more information. In the case of data that takes more than a few seconds to load, we should help users decide whether they want to wait, and prepare them for when the data are available.

What extra information can we give our users without distracting them from anything else on the page? A greyed-out version of your final chart often works well.

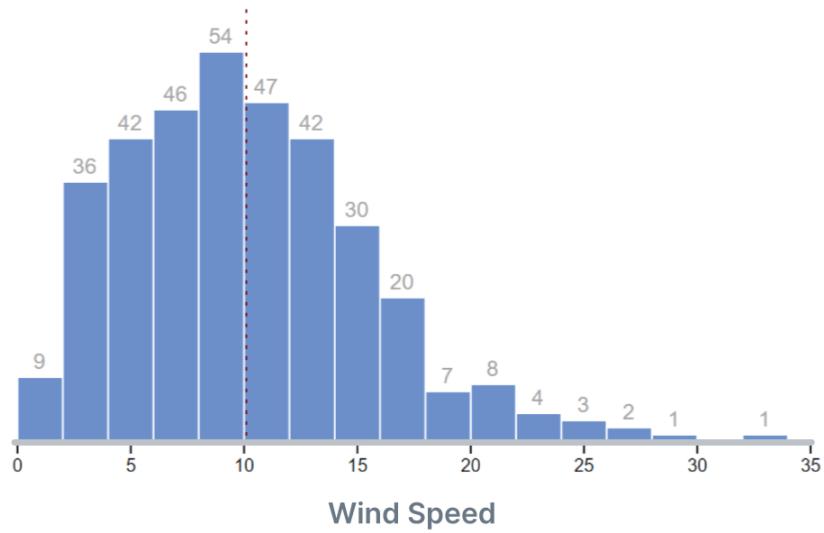


Additionally, an animation can be helpful to indicate that this state is not final.

Another solution is to render the chart with empty or random data. This is generally a bad idea, though, since it too closely mimics the loaded state and could confuse the user.

Loaded

When your data have loaded, animating the final state in looks great and also alerts your user that the wait is over!

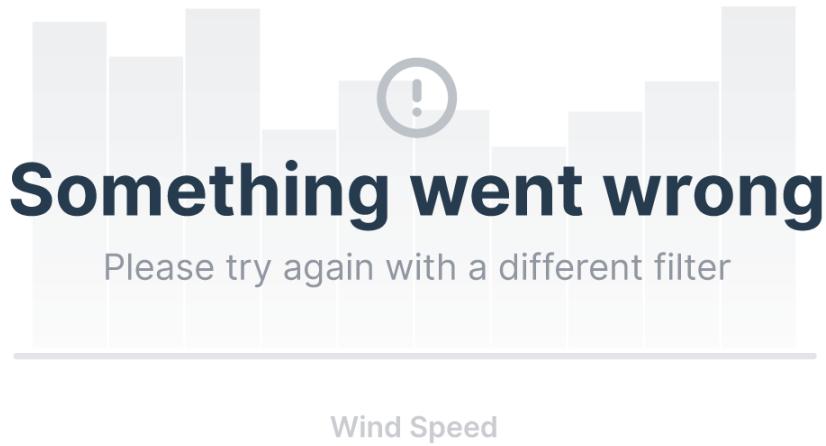


loaded state

Make sure that your loading and loaded states are the same size, so your content isn't jumping around on the page. This is especially important if the user could be looking at something further down on the page.

Error

Error states are important signals that something is wrong and the chart won't be appearing. Make sure these are visually distinct enough from your loading state that a waiting user is alerted quickly.



Wind Speed

error state

Error states can be jarring — we've all had the experience of waiting for a page to load, just to be rejected. Make sure to tell your users what they can do to fix the issue: try again later, upgrade their account, change the filter, etc.

Empty

Empty states are a unique error state — instead of signalling that there was an issue, they signal that the data does not exist.



No data yet

Check out [these docs](#) for help integrating

Wind Speed

empty state

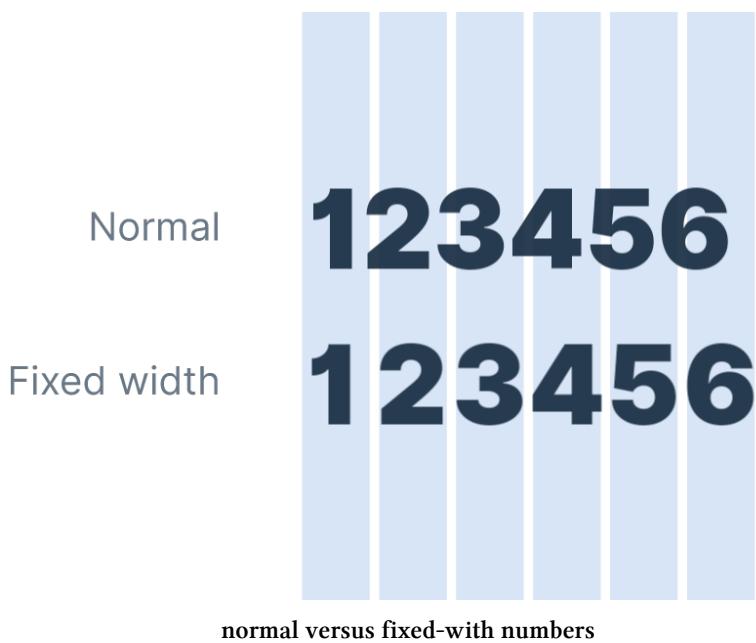
Empty states require an extra check to see if there are any data values. For example, we might receive an array of empty values for a timeline — in this case, we'll have to run through the whole array to check whether or not we have any non-zero values.

Dancing numbers

If we're showing numbers on our dashboard, it's a good idea to use a monospace font — especially when those numbers will be changing when the data updates or filters change. This will prevent layout changes when the numbers change, plus numbers in a column will be easier to compare (such as in a table).

This might require bringing in a new font, although some fonts have a fixed-width setting that can be toggled via CSS. For example, the font we use in our chart code, Inter, can be switched to fixed width with the following CSS.

```
font-feature-settings: 'tnum' 1;
```



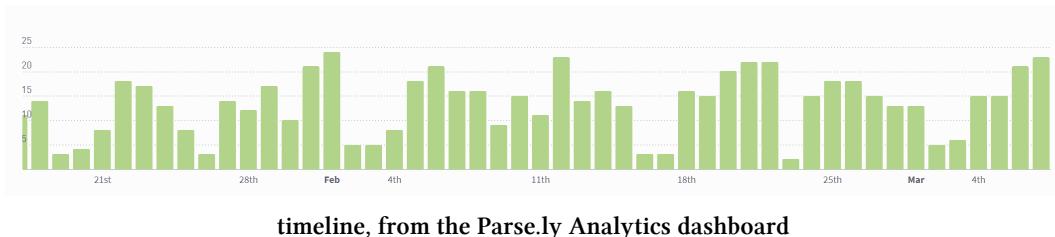
Dealing with outliers

The data we use for our chart can vary wildly — the values might cluster together tightly or we might have to display very different values. When there are extreme outlier values in our data, the scale can be exaggerated and drown out smaller values.

As usual, the solution will depend on the goal of the chart. If the goal is to show a birds-eye-view of the data, letting the outliers skew the whole chart might be ideal.

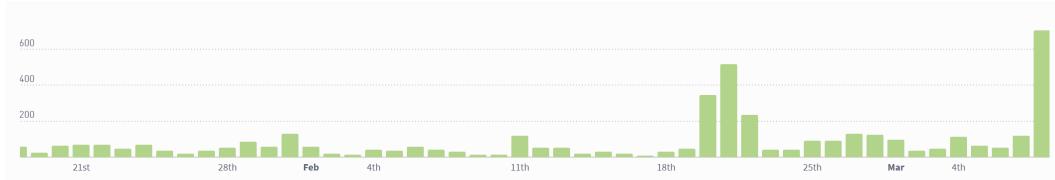
In most cases, though, the goal is to see the pattern for most of the data (and be notified that outliers exist). Let's look at a few examples.

This timeline (in the Parse.ly Analytics dashboard) is designed to show daily views over time for a specific website.



timeline, from the Parse.ly Analytics dashboard

Setting the y scale domain to `[0, max daily views]` usually works, but if the website publishes an article that goes viral, the site might receive four times the typical daily traffic. While exciting, this will blow out the y scale and make daily trends hard to see.

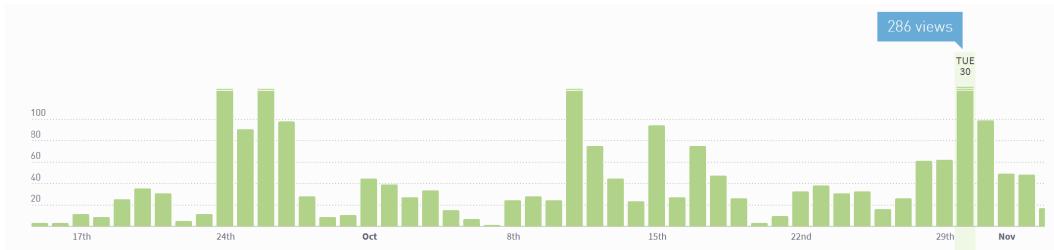


timeline with outliers, from the Parse.ly Analytics dashboard

To make the solution clear, let's lay out the goal for this chart: to show website owners how much traffic their site is getting — is that number increasing or decreasing over time? Are there weekly or monthly patterns? Are there any recent changes that they need to be aware of?

With this in mind, we'll apply two rules:

1. If an outlier exists in recent data (say, the past week), we'll use the default y scale domain of `[0, max daily views]`. This enables viewers to see the extremity of an ongoing traffic spike — something to celebrate!
2. If an outlier exists more than 7 days ago, it will get cropped and we'll define our y scale domain as `[0, mean daily views + 2.5 standard deviations]`. We don't want to lose this data, though! We'll show a stripe on top of the day's bar, with extra stripes for more extreme outliers. This way we essentially have two y scales — one for most of the bars and one for the outlier stripes, enabling us to see both the general pattern and get a sense of extreme traffic patterns.



timeline with cropped outliers - cropped, from the Parse.ly Analytics dashboard

This pattern is mimicked in Parse.ly's other dashboard (Parse.ly Currents). The following chart shows traffic over time (the line/area) *and* content production (the dots).



timeline, from the Parse.ly Currents dashboard

In an effort to prevent overwhelming the main chart with the enhancement of content production data, the dot scale is restricted to have one dot represent one article. We also want to keep the dots from getting too tall, so we'll cap the dot's y scale at 1/3 the height.

Unfortunately, some data have more articles than we have room to display. The solution here is similar to the previous solution — have another, abbreviated, y scale for outliers. In this case, arrows are used instead of stripes, since they are similar to the dots but different enough to stand out.



timeline with cropped outliers, from the Parse.ly Currents dashboard

There are many creative ways to show outliers while preserving the rest of the chart, but how do we detect outliers?

There are various definitions of an “outlier”. A good general-use definition is any value that is more than 2.5 standard deviations from the mean.

Designing tables

Often we’ll want to show the exact values in our dataset. In this case, it’s hard to beat a simple table. Although designing a table sounds straightforward, there are ways to make a simple table more engaging and easier to read.

Let’s redesign a table of our weather data — feel free to follow along at
</code/09-dashboard-design/table/draft/>.

Writing the code along with these changes is a great way to practice your d3 and CSS skills. If you get stumped, check out the finished code at

</code/09-dashboard-design/table/completed/>

Even if you’re not following along, check out the original table at <http://localhost:8080/09-dashboard-design/table/draft/>⁶³.

⁶³<http://localhost:8080/09-dashboard-design/table/draft/>

We'll start out with an unstyled table showing several metrics (columns) over multiple days (rows).

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
2017-01-03	Clear throughout the day.	45.34	1483423200	7.63	rain	1
2017-01-04	Foggy in the evening.	46.96	1483509600	8.92	rain	2
2017-01-05	Mostly cloudy until evening.	43.53	1483628400	13.42	rain	2
2017-01-06	Mostly cloudy starting in the afternoon, continuing until evening.	47.13	1483682400	3.4	rain	1
2017-01-07	Overcast until evening.	45.5	1483736400	6.7	rain	2
2017-01-08	Overcast until evening.	42.47	1483812000	13.07	rain	2
2017-01-09	Partly cloudy until afternoon.	39.22	1483909200	14.66	rain	2
2017-01-10	Mostly cloudy until evening.	35.61	1484006400	11.76	snow	2
2017-01-11	Partly cloudy in the morning.	38.83	1484125200	6.46	rain	2
2017-01-12	Mostly cloudy until evening.	37.95	1484168400	6.19	rain	2

Our table

This is pretty hard to read, with cramped cells and dark borders. To start, we'll add padding to each cell and take away the borders.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
2017-01-03	Clear throughout the day.	45.34	1483423200	7.63	rain	1
2017-01-04	Foggy in the evening.	46.96	1483509600	8.92	rain	2
2017-01-05	Mostly cloudy until evening.	43.53	1483628400	13.42	rain	2
2017-01-06	Mostly cloudy starting in the afternoon, continuing until evening.	47.13	1483682400	3.4	rain	1
2017-01-07	Overcast until evening.	45.5	1483736400	6.7	rain	2
2017-01-08	Overcast until evening.	42.47	1483812000	13.07	rain	2
2017-01-09	Partly cloudy until afternoon.	39.22	1483909200	14.66	rain	2
2017-01-10	Mostly cloudy until evening.	35.61	1484006400	11.76	snow	2
2017-01-11	Partly cloudy in the morning.	38.83	1484125200	6.46	rain	2
2017-01-12	Mostly cloudy until evening.	37.95	1484168400	6.19	rain	2

Our table, padded

Unfortunately, without those borders it's hard to separate each row. We could add the borders back and lighten them, but that will still add visual clutter. Instead, let's add zebra striping: alternating rows are a different color. This will help us follow a row from the left to the right.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
2017-01-03	Clear throughout the day.	45.34	1483423200	7.63	rain	1
2017-01-04	Foggy in the evening.	46.96	1483509600	8.92	rain	2
2017-01-05	Mostly cloudy until evening.	43.53	1483628400	13.42	rain	2
2017-01-06	Mostly cloudy starting in the afternoon, continuing until evening.	47.13	1483682400	3.4	rain	1
2017-01-07	Overcast until evening.	45.5	1483736400	6.7	rain	2
2017-01-08	Overcast until evening.	42.47	1483812000	13.07	rain	2
2017-01-09	Partly cloudy until afternoon.	39.22	1483909200	14.66	rain	2
2017-01-10	Mostly cloudy until evening.	35.61	1484006400	11.76	snow	2
2017-01-11	Partly cloudy in the morning.	38.83	1484125200	6.46	rain	2
2017-01-12	Mostly cloudy until evening.	37.95	1484168400	6.19	rain	2

Our table, with zebra striping

Much better! Now we can focus on the values. There are a few good rules of thumb for text alignment in tables:

1. Align text to the left. This helps users scan the table, since we're used to reading left to right.
2. Align numerical values to the right. This helps with comparing different numbers within the column because their decimal points will line up.
3. Align the column headers with their values. This will help with scanning and keep the label close to the metric value.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Did Snow	UV Index
2017-01-03	Clear throughout the day.	45.34	1483423200	7.63	rain	1
2017-01-04	Foggy in the evening.	46.96	1483509600	8.92	rain	2
2017-01-05	Mostly cloudy until evening.	43.53	1483628400	13.42	rain	2
2017-01-06	Mostly cloudy starting in the afternoon, continuing until evening.	47.13	1483682400	3.4	rain	1
2017-01-07	Overcast until evening.	45.5	1483736400	6.7	rain	2
2017-01-08	Overcast until evening.	42.47	1483812000	13.07	rain	2
2017-01-09	Partly cloudy until afternoon.	39.22	1483909200	14.66	rain	2
2017-01-10	Mostly cloudy until evening.	35.61	1484006400	11.76	snow	2
2017-01-11	Partly cloudy in the morning.	38.83	1484125200	6.46	rain	2
2017-01-12	Mostly cloudy until evening.	37.95	1484168400	6.19	rain	2

Our table, aligned

Our numbers are still hard to compare, though, since they have different amounts of decimal points. Let's use `d3.format()` to ensure that all of our numbers are the same granularity. The number of decimal points we want will depend on the metric — note that we keep one decimal point for temperature but two decimal points for wind speed.

Let's also format our dates to make them more human friendly and get rid of redundant information.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
1/03	Clear throughout the day.	45.3	1 AM	7.63	rain	1
1/04	Foggy in the evening.	47.0	1 AM	8.92	rain	2
1/05	Mostly cloudy until evening.	43.5	10 AM	13.42	rain	2
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1	1 AM	3.40	rain	1
1/07	Overcast until evening.	45.5	4 PM	6.70	rain	2
1/08	Overcast until evening.	42.5	1 PM	13.07	rain	2
1/09	Partly cloudy until afternoon.	39.2	4 PM	14.66	rain	2
1/10	Mostly cloudy until evening.	35.6	7 PM	11.76	snow	2
1/11	Partly cloudy in the morning.	38.8	4 AM	6.46	rain	2
1/12	Mostly cloudy until evening.	38.0	4 PM	6.19	rain	2

Our table, formatted

These numbers are *still* hard to compare! The value `47.1` looks smaller than the next

value, `45.5`, because it takes up less space.

In most fonts, numbers have different widths to make them flow together. The font we're using, Inter UI, has a CSS property that makes each character equal-width:

```
font-feature-settings: 'tnum' 1;
```

If the font you're using doesn't have this feature, switch to a monospace font for your numbers.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
1/03	Clear throughout the day.	45.3	1 AM	7.63	rain	1
1/04	Foggy in the evening.	47.0	1 AM	8.92	rain	2
1/05	Mostly cloudy until evening.	43.5	10 AM	13.42	rain	2
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1	1 AM	3.40	rain	1
1/07	Overcast until evening.	45.5	4 PM	6.70	rain	2
1/08	Overcast until evening.	42.5	1 PM	13.07	rain	2
1/09	Partly cloudy until afternoon.	39.2	4 PM	14.66	rain	2
1/10	Mostly cloudy until evening.	35.6	7 PM	11.76	snow	2
1/11	Partly cloudy in the morning.	38.8	4 AM	6.46	rain	2
1/12	Mostly cloudy until evening.	38.0	4 PM	6.19	rain	2

Our table, with fixed-width numbers

This table is much more readable than it was, but we can do more to help users process it faster. Remember the dimensions in which we can represent data from [Chapter 7](#)? Those dimensions aren't restricted to charts. Let's create a color scale for **Max Temp** and **Wind Speed** and fill the background of each cell with its value's color.

We'll make sure to use semantic colors for quicker association: blue to red for temperature, and white to slate gray for wind speed. Additionally, having two different color scales helps keep these columns visually distinct.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Precipitation	UV Index
1/03	Clear throughout the day.	45.3	1 AM	7.63	rain	1
1/04	Foggy in the evening.	47.0	1 AM	8.92	rain	2
1/05	Mostly cloudy until evening.	43.5	10 AM	13.42	rain	2
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1	1 AM	3.40	rain	1
1/07	Overcast until evening.	45.5	4 PM	6.70	rain	2
1/08	Overcast until evening.	42.5	1 PM	13.07	rain	2
1/09	Partly cloudy until afternoon.	39.2	4 PM	14.66	rain	2
1/10	Mostly cloudy until evening.	35.6	7 PM	11.76	snow	2
1/11	Partly cloudy in the morning.	38.8	4 AM	6.46	rain	2
1/12	Mostly cloudy until evening.	38.0	4 PM	6.19	rain	2

Our table, with background colors

Great! Now we can quickly see that the first few days were warm and then cooled down, and the days are windy in little spurts.

Another way to make our table more scannable is to convert **UV Index** into symbols. Since it is a **discrete** metric, we can display a number of symbols to represent the value.

There is another column that we can convert to symbols: **Precipitation**. But we also notice that it's a **binary** metric, and there are only two possible values. Let's change it into a **Did Snow** column and only display a symbol if the value is `snow`.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Did Snow	UV Index
1/03	Clear throughout the day.	45.3	1 AM	7.63	*	
1/04	Foggy in the evening.	47.0	1 AM	8.92	**	
1/05	Mostly cloudy until evening.	43.5	10 AM	13.42	**	
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1	1 AM	3.40	*	
1/07	Overcast until evening.	45.5	4 PM	6.70	**	
1/08	Overcast until evening.	42.5	1 PM	13.07	**	
1/09	Partly cloudy until afternoon.	39.2	4 PM	14.66	**	
1/10	Mostly cloudy until evening.	35.6	7 PM	11.76	*	**
1/11	Partly cloudy in the morning.	38.8	4 AM	6.46	**	
1/12	Mostly cloudy until evening.	38.0	4 PM	6.19	**	

Our table, with symbols

Let's see if we can improve on **Max Temp Time**. Since the times span both AM and PM, they are hard to compare quickly. Instead, we can use another data dimension: **position**. Instead, let's display a line that is further right if the max temp occurred later in the day.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Did Snow	UV Index
1/03	Clear throughout the day.	45.3		7.63	*	
1/04	Foggy in the evening.	47.0		8.92	**	
1/05	Mostly cloudy until evening.	43.5		13.42	**	
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1		3.40	*	
1/07	Overcast until evening.	45.5		6.70	**	
1/08	Overcast until evening.	42.5		13.07	**	
1/09	Partly cloudy until afternoon.	39.2		14.66	**	
1/10	Mostly cloudy until evening.	35.6		11.76	*	**
1/11	Partly cloudy in the morning.	38.8		6.46	**	
1/12	Mostly cloudy until evening.	38.0		6.19	**	

Our table, with time lines

This is much easier to scan! We can quickly see that the max temp was earlier in the day for the first four rows, then in the afternoon for the next four rows. As usual, how

you represent this metric will depend on your goals. If the specific time is important, maybe adding another column with the exact value would be helpful.

The last column we want to update is our **Summary** metric. The strings can be quite long and take up lots of space. Let's bump that down a font size to save space and decrease its visual importance.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Did Snow	UV Index
1/03	Clear throughout the day.	45.3		7.63	*	
1/04	Foggy in the evening.	47.0		8.92	**	
1/05	Mostly cloudy until evening.	43.5		13.42	**	
1/06	Mostly cloudy starting in the afternoon, continuing until evening.	47.1		3.40	*	
1/07	Overcast until evening.	45.5		6.70	**	
1/08	Overcast until evening.	42.5		13.07	**	
1/09	Partly cloudy until afternoon.	39.2		14.66	**	
1/10	Mostly cloudy until evening.	35.6		11.76	*	**
1/11	Partly cloudy in the morning.	38.8		6.46	**	
1/12	Mostly cloudy until evening.	38.0		6.19	**	

Our finished table

If we have a long table, we can remind our users what each column is for by sticking the header to the top. Thankfully, modern css makes this simple, we only need these lines of CSS:

```
thead th {
  position: sticky;
  top: 0;
}
```

To make the stickied header stand out while it's stuck, let's give it a dark background color and light text.

Additionally, let's highlight a row when the user hovers over it. This will allow them to easily compare values all the way across a row.

Day	Summary	Max Temp	Max Temp Time	Wind Speed	Did Snow	UV Index
2/13	Partly cloudy in the morning.	42.1		5.93		***
2/14	Partly cloudy until evening.	39.8		2.24		***
2/15	Windy starting in the evening.	44.4		20.56		***
2/16	Mostly cloudy until evening.	60.8		12.02		***
2/17	Windy in the morning and mostly cloudy until evening.	34.5		22.87	*	***
2/18	Partly cloudy until afternoon.	41.5		9.36		***
2/19	Mostly cloudy until evening and windy until afternoon.	44.3		15.44		***
2/20	Windy until afternoon and mostly cloudy until evening.	35.3		28.70	*	***
2/21	Mostly cloudy until evening.	36.1		11.55		****
2/22	Overcast until evening.	35.8		14.22		***

Table header sticks on scroll

Wonderful! If you haven't been following along, make sure to check out the final implementation at [/09-dashboard-design/table-completed/](#).

Even though creating a table may not sound exciting, there is room to use our new knowledge of data types and data dimensions to make it more effective and easier to read.

Designing a dashboard layout

We've talked about how to build and design parts of a dashboard, but how do we pull it all together?

Let's look at a few tips for designing the dashboard's layout:

Have a main focus

Many designs of dashboards throw all of the data on the page at once. While this might work in some cases, users can be easily overwhelmed. **Aim for one main chart on screen at a time.**

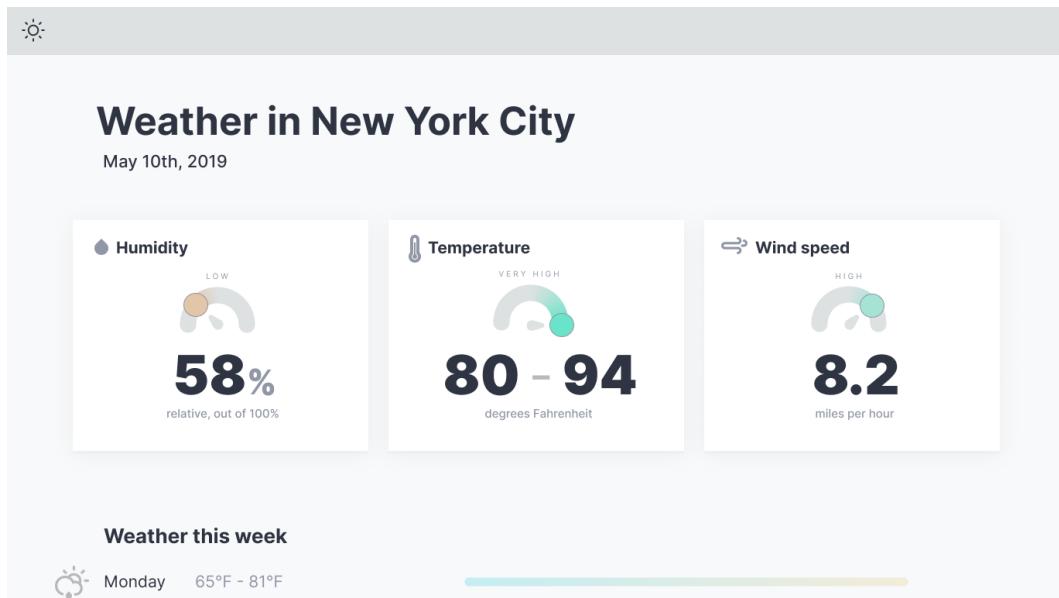
For example, if the main goal is to show how a metric changed over time, or to look for temporal trends, lead with a large timeline that the user can focus on.



Dashboard with main chart

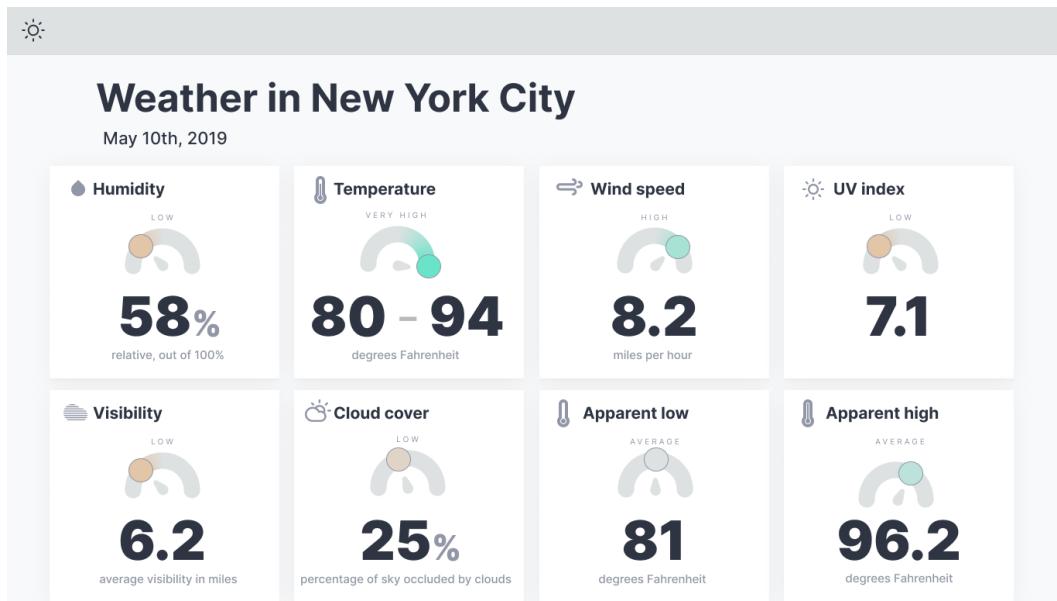
Keep the number of visible metrics small

If there isn't *one* main takeaway for users, showing multiple metrics can help to quickly convey an idea of the bigger picture. Make sure to give as much context as possible – how do these numbers compare to historical averages, or similar metrics?



Dashboard with three main metrics

Keep the number of metrics small though, to prevent making the user process a lot of information at once. Here's the same dashboard with eight metrics shown at once – a user could easily feel overwhelmed with this design. What should they look at first? When everything is important, nothing is.



Dashboard with eight main metrics

Data density can increase with a user's comfort – maybe add an option for a denser layout for power users?

Let the user dig in

We want to reduce the amount of information we show at once, but we still want to give our users the information they need. Once you're showing only what's most important, let the user **dig in** to a piece of information. The interaction will depend on the amount of extra information you want to show. If could involve any progressive disclosure technique, the main ones being:

A tooltip

to show an explanation of a metric, or 1-2 sentences of extra information.

A modal

to show a longer explanation that might include a detailed chart.

Expand in place

to show a longer explanation and/or a chart. This is a good option when the user wants to compare multiple items.

Switch to a detail page

to show much more information. A whole page gives enough space for multiple detailed charts and paragraphs of information. They also feel more concrete and allow users to bookmark or share the url.

Deciding questions

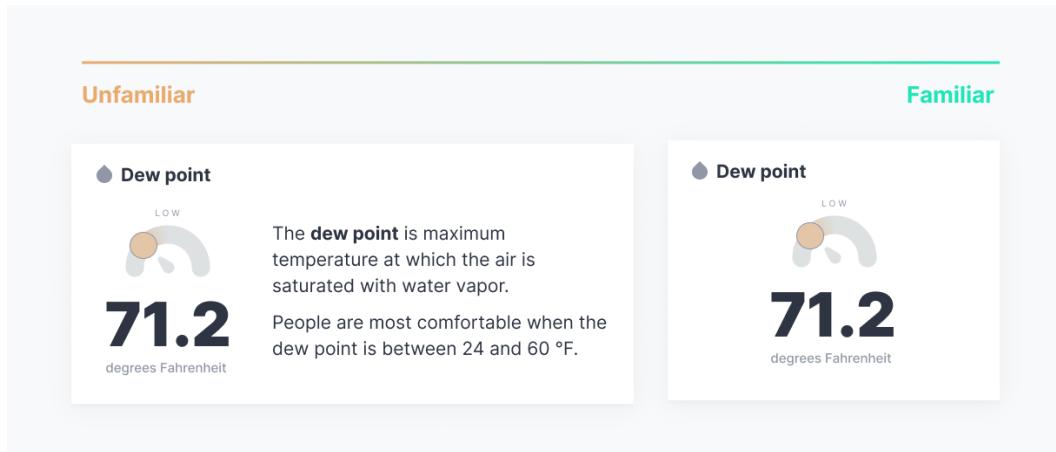
There isn't one layout that will fit any purpose, but a few choice questions can guide you towards a more suitable layout.

How familiar are users with the data?

For a dashboard that is also introducing users to the dataset, take the time to explain the nuances of the data. Make sure the answers to these questions are clear:

- where is the data from?
- how was the data collected?
- what are non-obvious nuances that might skew the data?
- what do these words mean? It's easy to use jargon that makes sense to *you*, but not to uninitiated users.

For example, the layout on the left might be more accessible for unfamiliar users, but might be totally unnecessary for a domain expert.

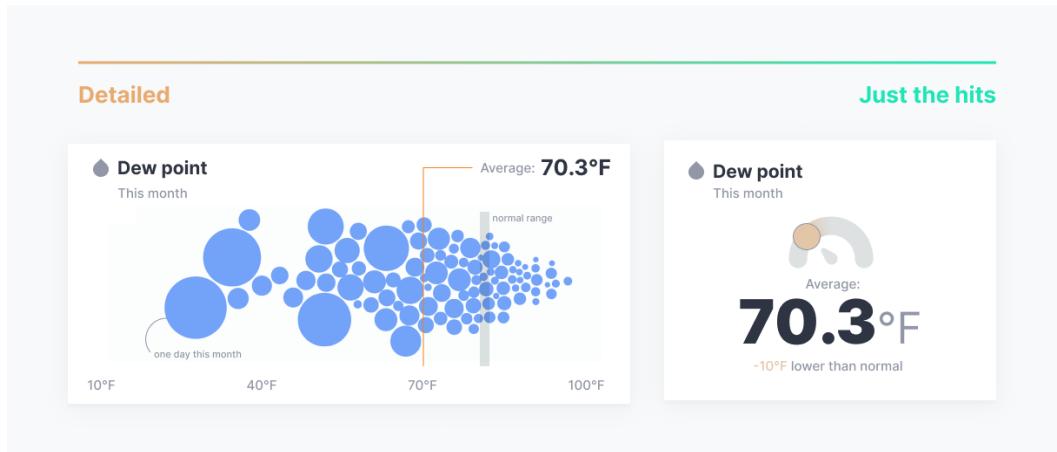


Metric layouts depending on user familiarity

How much time do users have to spend?

If your users are CEOs or people without much time on their hands, you want to give them the takeaways, and fast. What insights are most important to your users? Lead with those metrics, and save the secondary information for further down the screen, or a detail page.

For example, a user with more time on their hands might appreciate the extra detail from the left layout. They might even learn something extra! But for users in a hurry, the layout on the right gets to the point quickly and gives them only as much information as they need.

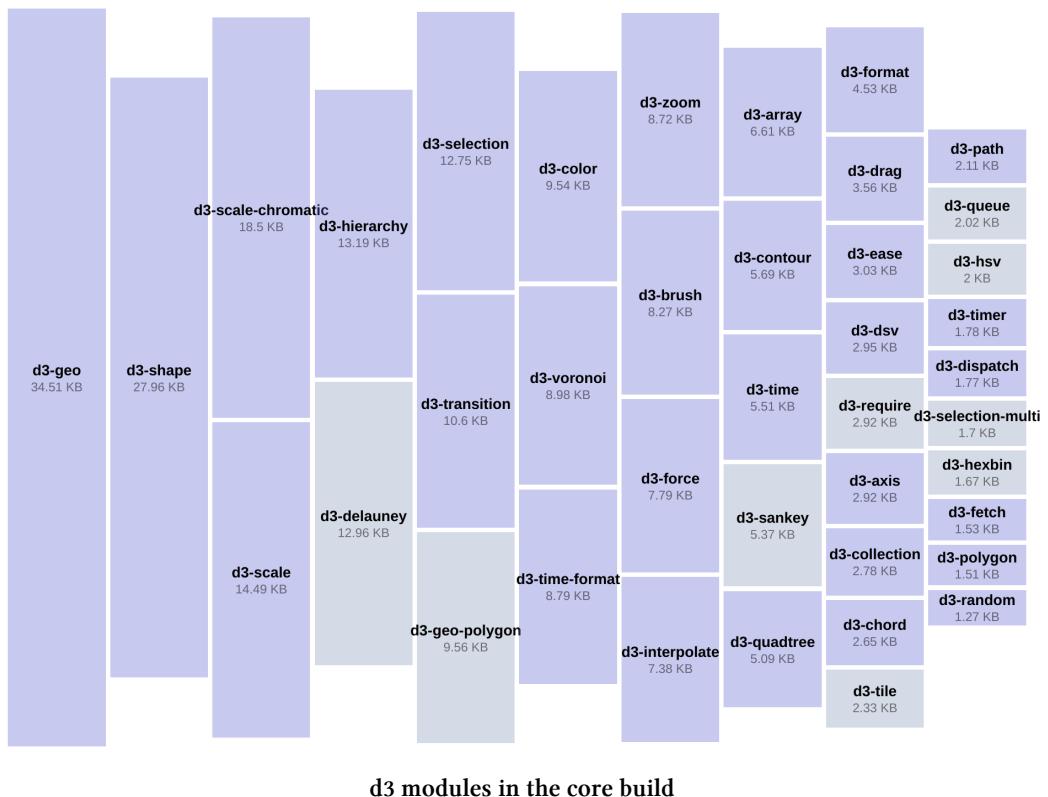


Metric layouts depending on how much time a user has

There are many ways to build a dashboard – hopefully this information will help on your next project. Remember to choose a main focus for each screen, give metrics context, and start with the goals of the dashboard.

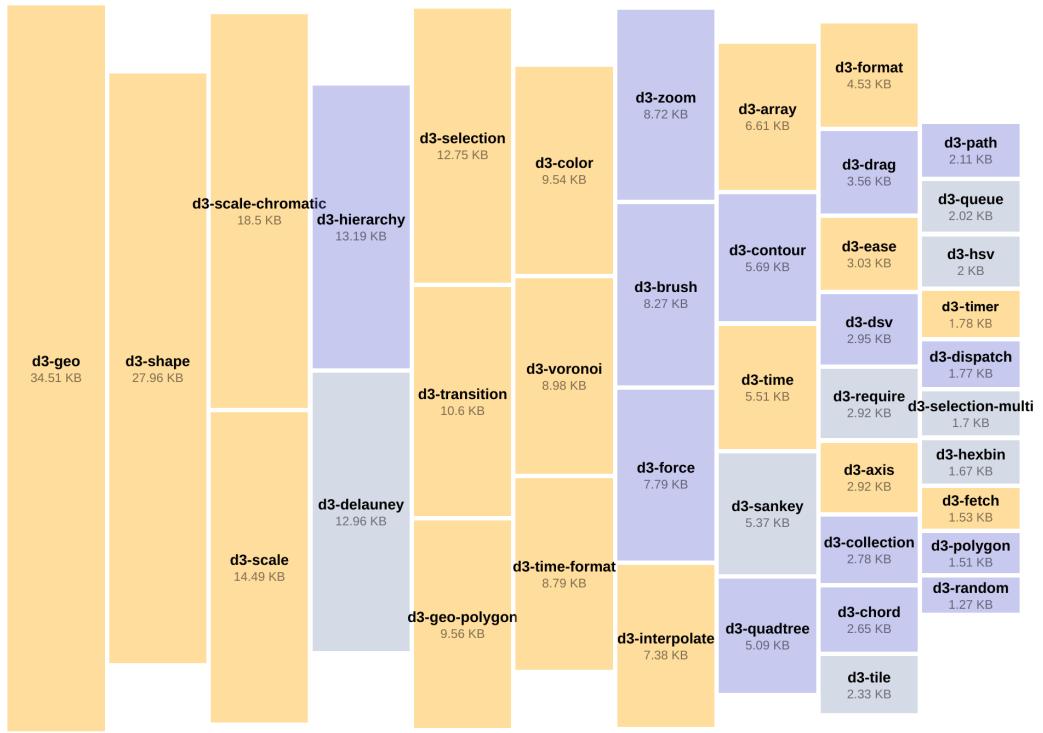
D3.js

Let's step back and look at all of the ground we've covered since we started this book. Here is a diagram of all of the d3.js modules, scaled by their minified size:



d3 modules in the core build

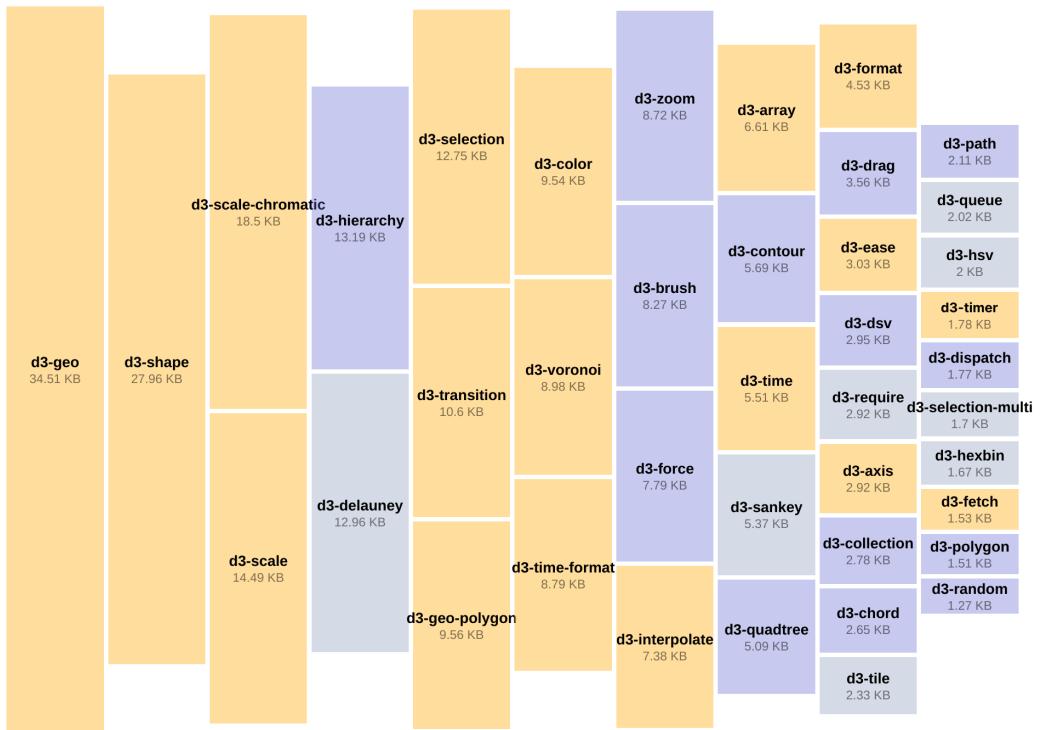
That's pretty overwhelming! But we've actually talked about almost half of them — here are the ones we've covered in yellow:



d3 modules that we've covered

What did we cover?

Let's look at the modules *we have* talked about:



d3 modules that we've covered

- **d3-geo**⁶⁴ helps with turning GeoJSON into 2D svg paths, and **d3-geo-polygon**⁶⁵ adds extra projections.
- **d3-shape**⁶⁶ helps with creating shapes — like our timeline and pie chart.
- **d3-scale-chromatic**⁶⁷ provides many color scales to use and **d3-color**⁶⁸ lets us manipulate colors to create our own color scales.
- **d3-scale**⁶⁹ helps us create scales to map from one domain to another. Be sure to explore the docs⁷⁰ to see what scale types we haven't covered, such as `d3.scaleLog()`, `d3.scaleThreshold()`, and `d3.scaleOrdinal()`, among others.

⁶⁴<https://github.com/d3/d3-geo>

⁶⁵<https://github.com/d3/d3-geo-polygon>

⁶⁶<https://github.com/d3/d3-shape>

⁶⁷<https://github.com/d3/d3-scale-chromatic>

⁶⁸<https://github.com/d3/d3-color>

⁶⁹<https://github.com/d3/d3-scale>

⁷⁰<https://github.com/d3/d3-scale>

- **d3-selection**⁷¹ helps with selecting and modifying elements on the webpage.
 - **d3-transition**⁷² helps with animating changes to our chart, and **d3-ease**⁷³ lets us change the timing of those animations.
 - **d3-voronoi**⁷⁴ helps create voronoi polygons that cover our whole **bounds**, making it easy to find the closest data point from any location.
 - **d3-format**⁷⁵ helps convert our data values into human-friendly strings, and **d3-time-format**⁷⁶ helps convert date time objects into friendly strings.
 - **d3-interpolate**⁷⁷ helps us with interpolating in different color spaces, but it also works behind the scenes to interpolate other values for **d3-transition** (such as numbers, strings, and dates).
 - **d3-array**⁷⁸ gives us utility methods for modifying and querying arrays, such as `d3.min()`, `d3.scan()`, and `d3.histogram()`.
-
- **d3-axis**⁷⁹ helps with creating axes — check out the docs⁸⁰ to see how to customize your ticks.
 - and lastly, the first module we learned: **d3-fetch**⁸¹, which helps to fetch and parse data files.

What did we not cover?

The d3 modules we haven't covered are more specialized — let's look at a few examples:

- **d3-hierarchy**⁸² helps with creating hierarchical charts, such as treemaps and circle packing.

⁷¹<https://github.com/d3/d3-selection>

⁷²<https://github.com/d3/d3-transition>

⁷³<https://github.com/d3/d3-ease>

⁷⁴<https://github.com/d3/d3-voronoi>

⁷⁵<https://github.com/d3/d3-format>

⁷⁶<https://github.com/d3/d3-time-format>

⁷⁷<https://github.com/d3/d3-interpolate>

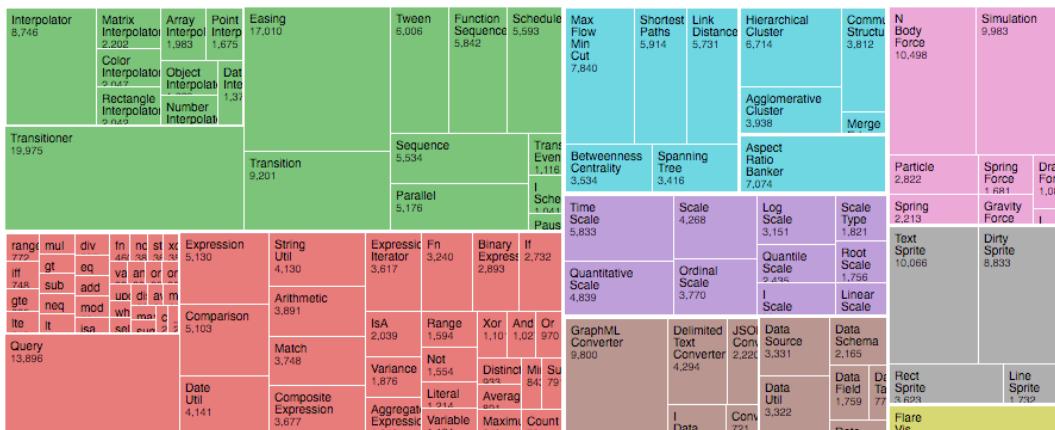
⁷⁸<https://github.com/d3/d3-array>

⁷⁹<https://github.com/d3/d3-axis>

⁸⁰<https://github.com/d3/d3-axis>

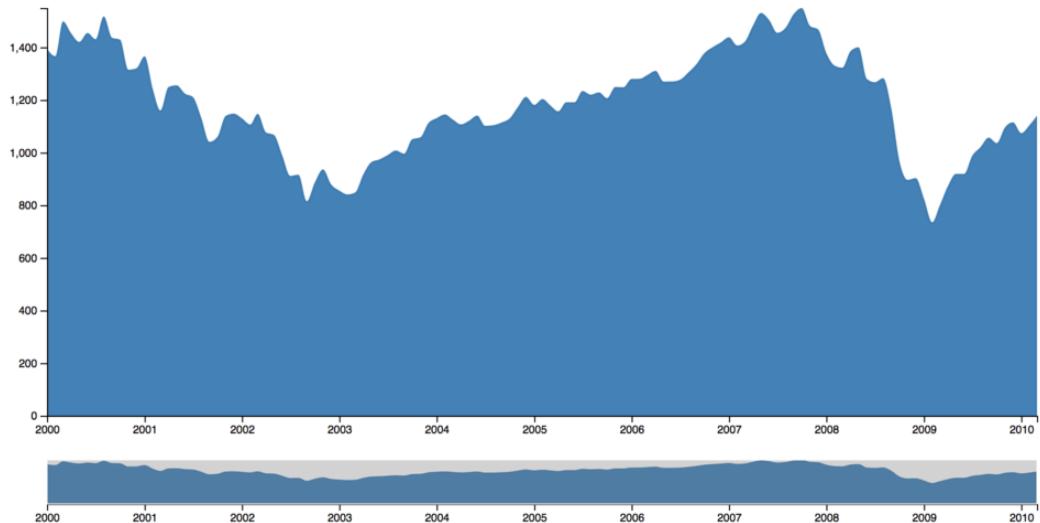
⁸¹<https://github.com/d3/d3-fetch>

⁸²<https://github.com/d3/d3-hierarchy>



treemap example from <https://github.com/d3/d3-hierarchy>

- **d3-zoom⁸³** and **d3-brush⁸⁴** help with creating interactive charts, where the user can “zoom in” on a specific part.

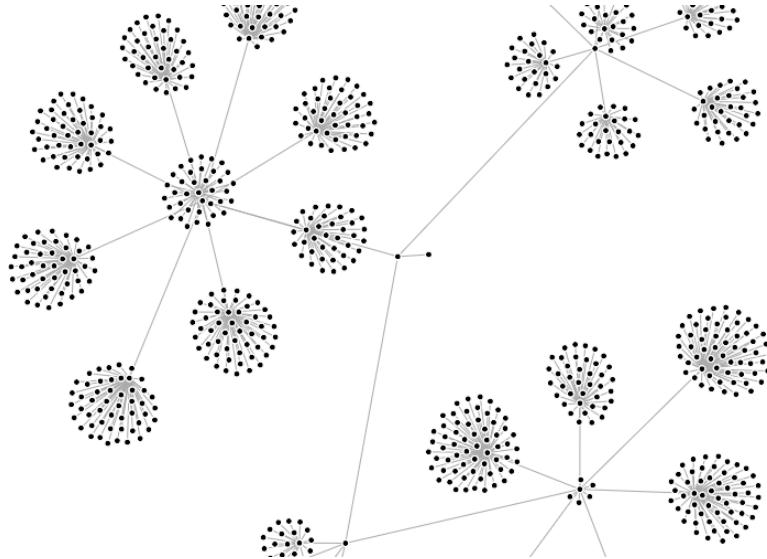


brushing example from <https://github.com/d3/d3-zoom>

⁸³<https://github.com/d3/d3-zoom>

⁸⁴<https://github.com/d3/d3-brush>

- **d3-force⁸⁵** helps with placing nodes in network-like charts and beeswarm charts.

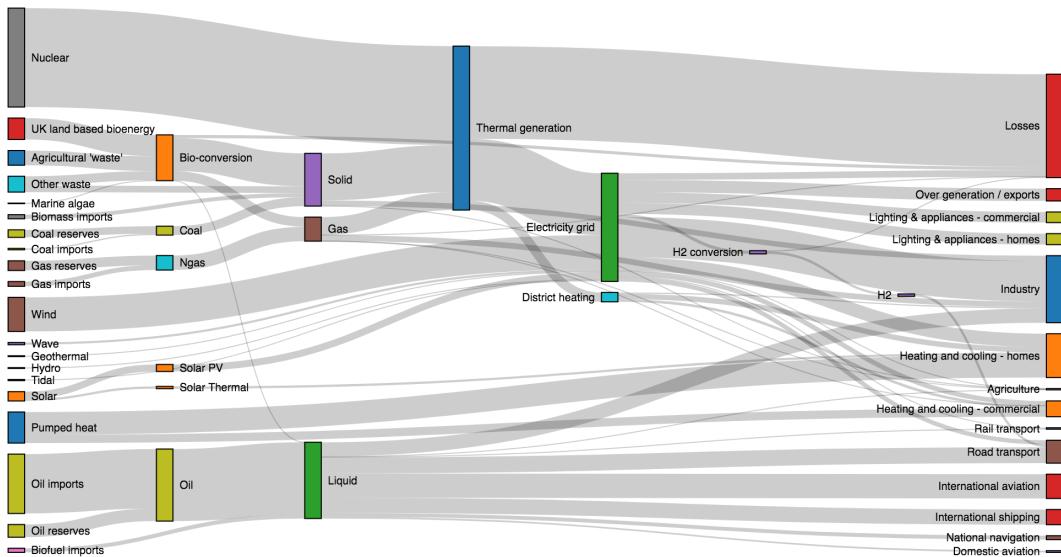


network diagram from <https://github.com/d3/d3-force>

- **d3-sankey⁸⁶** specifically helps with creating Sankey diagrams, which can visualize flow and breakdowns.

⁸⁵<https://github.com/d3/d3-force>

⁸⁶<https://github.com/d3/d3-sankey>



Now that you understand the basics of how to create a chart with d3.js and how the API generally works, I encourage you to explore these different modules. It's at least good to be aware of them — they might come in handy in the future.

Going forward

Great work learning all of that! As you can see, we covered *a lot* of concepts in this book — both how to implement it and also the theory behind how to design a good data visualization.

I'm sure you have tons of data you want to visualize — get out there and do it! Don't be overwhelmed by starting a chart by yourself — take it step by step (the chart steps are available in [Appendix B](#)), apply the fundamentals we've learned here, and refer back to the book when you need to pointer.

If you want ideas, here is a list of places to find inspiration:

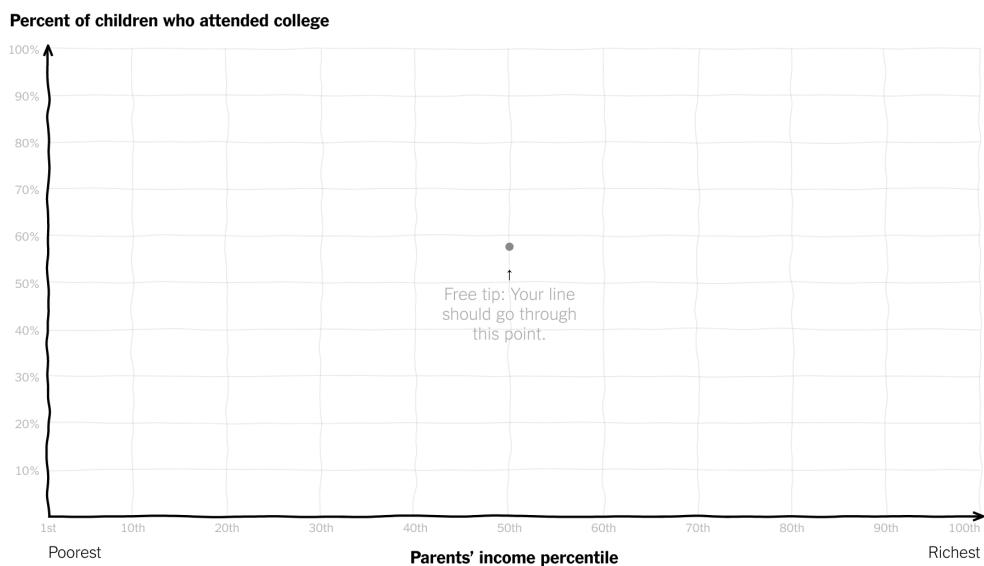
[Flowing Data](#)⁸⁷ is a great, frequently-updated blog by Nathan Yau with original data visualizations and links to other complex data visualizations.

⁸⁷<https://flowingdata.com/>

The Pudding⁸⁸ is a digital publication that explores popular concepts with data visualizations. This is a great place to find visualizations that are interactive and easily digestable.

The New York Times Upshot⁸⁹ often has interactive pieces that are linked to news events. Some of my favorite examples come from here, like their **You Draw It** article from 2015⁹⁰ where *you guess what the data look like*.

Draw your line on the chart below



You Draw It from (<https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html>)

⁸⁸<https://pudding.cool/>

⁸⁹<https://www.nytimes.com/section/upshot>

⁹⁰<https://www.nytimes.com/interactive/2015/05/28/upshot/you-draw-it-how-family-income-affects-childrens-college-chances.html?mtrref=wattenberger-book>

How was your experience?

I hope you found the book informative and fun! Did you find any part confusing or especially insightful? Send me an email at [us@fullstack.io⁹¹](mailto:us@fullstack.io) or tweet at me at [@wattenberger⁹²](https://twitter.com/wattenberger).

Send me your thoughts, feedback, and screenshots of your completed charts! If you've made anything you want to brag about with your new knowledge, I'd love to see it! Show it off and tag me in a tweet.

⁹¹<mailto:us@fullstack.io>

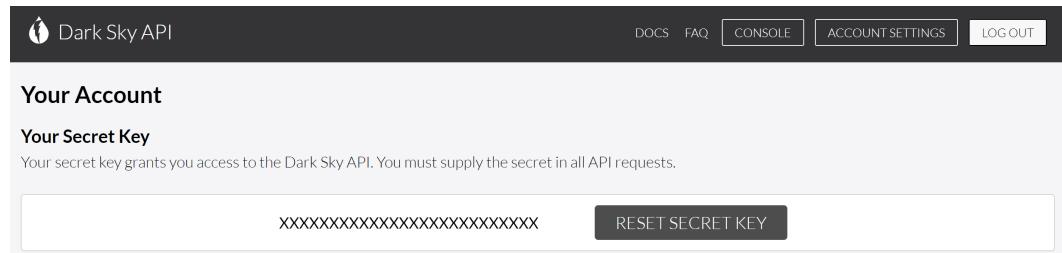
⁹²<https://twitter.com/wattenberger>

Appendix

A. Generating our own weather data

Let's make sure that we have python 3 installed by running `python --version` in the terminal. If we get `command not found` or if the version is lower than 3.0.0, we'll need to upgrade first.

The Dark Sky API is a simple weather API that can tell what the weather was like at a specific time and place. The first 1,000 calls per day are free, which is perfect for our purposes. Go to darksky.net/dev/register and register for an account, then grab your apikey on the account screen.



The screenshot shows the 'Your Account' section of the Dark Sky API interface. At the top, there are links for 'DOCS', 'FAQ', 'CONSOLE', 'ACCOUNT SETTINGS', and 'LOG OUT'. Below this, the 'Your Account' section is titled 'Your Secret Key'. A note states: 'Your secret key grants you access to the Dark Sky API. You must supply the secret in all API requests.' A large input field contains a long string of characters: 'xxxxxxxxxxxxxxxxxxxxxxxxxxxxxx'. To the right of the input field is a button labeled 'RESET SECRET KEY'.

Dark Sky apikey interface

Let's open up the `get_weather_data.py` file and update the `apikey` value, then grab our latitude and longitude from gps-coordinates.org⁹³. Make sure to preserve the sign of the longitude (if it's negative, keep the minus sign).

Once we update those values we can run the script.

The script will use 365 of our 1,000 free daily api calls, so double-check the values before we run it.

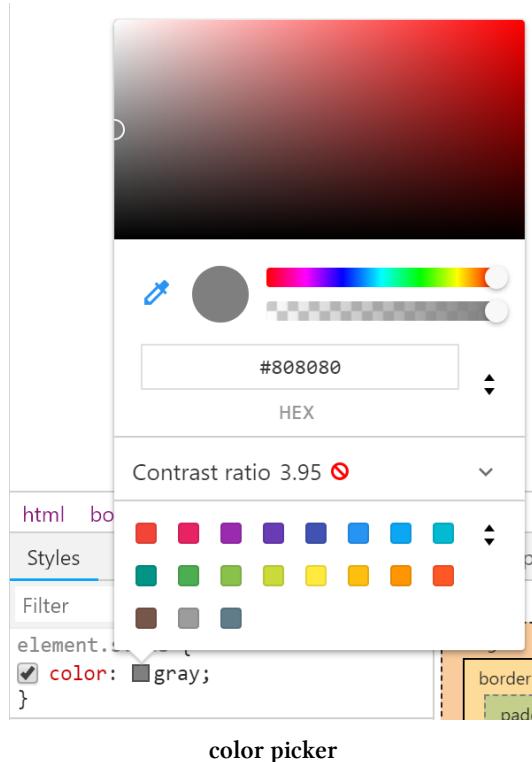
⁹³gps-coordinates.org

```
$ python get_weather_data.py
```

Now we should have a new `my_weather_data.json` file with the past 365 days of local weather. Let's take a closer look!

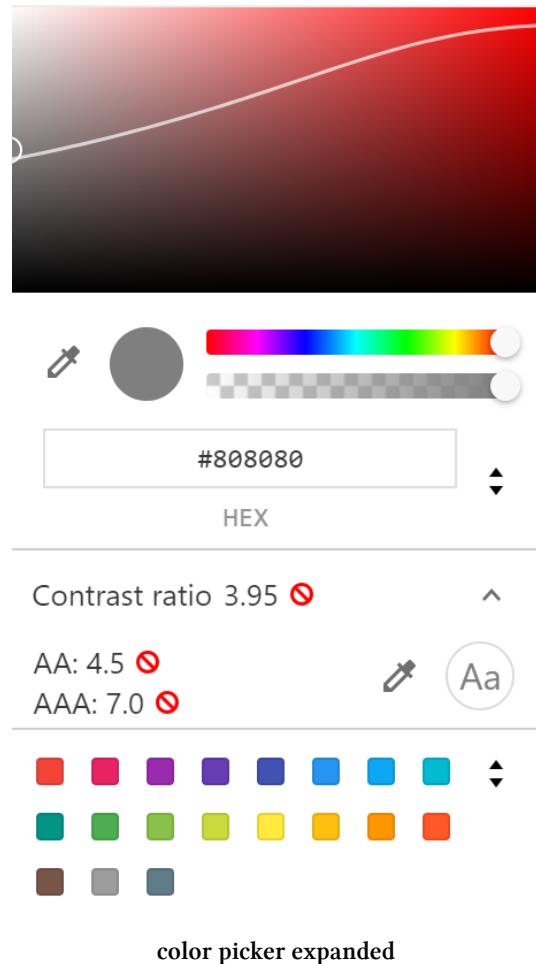
Chrome's Color Contrast Tool

If you're using Chrome, there is a great tool to check that your colors have enough contrast right in your dev tools. To check it out, you can load up one of our chart examples and inspect the `wrapper` element in the Elements panel. Give it a `color` property of `gray` and click the colored box to the left of `gray`.



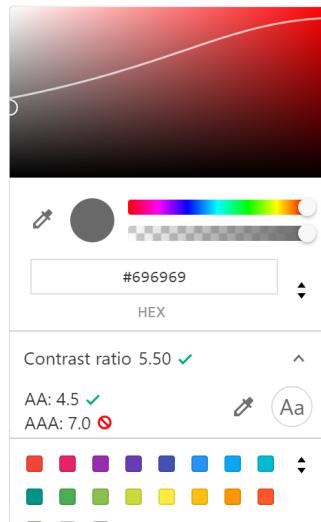
color picker

We can see the normal color picking controls, but also a **Contrast ratio** section with a warning symbol. Let's click the arrow to the right of the section to expand it.

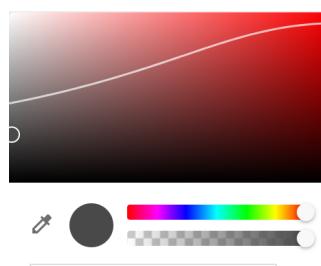


Now we can see that the dev tools are making two checks: AA and AAA. These are checking for different contrast ratio levels — by **contrast ratio**, we're talking about the difference in luminance between the text and its surroundings (higher means more contrast). AA is testing for at least a 4.5:1 contrast ratio (for people with 20/40 vision) and the AAA threshold is even higher at 7:1.

As we change the color by clicking around the color map at the top of the popup, we can see that values below the curved white line pass the AA threshold (one check mark) and values that are further below the white line pass the AAA threshold (two check marks).



color picker check



color picker checks

When making charts for your dashboard, it's a good idea to check your main colors every now and then to make sure they are visible to all users.

B. Chart-drawing checklist

These are the basic steps to follow to create a chart.

1. Access data

Look at the data structure and declare how to access the values we'll need

2. Create chart dimensions

Declare the physical (i.e. pixels) chart parameters

3. Draw canvas

Render the chart area and bounds element

4. Create scales

Create scales for every data-to-physical attribute in our chart

5. Draw data

Render your data elements

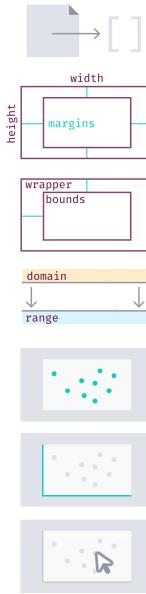
6. Draw peripherals

Render your axes, labels, and legends

7. Set up interactions

Initialize event listeners and create interaction behavior

Chart drawing checklist



Access data

Look at the data structure and declare how to access the values we'll need

Create chart dimensions

Declare the physical (i.e. pixels) chart parameters

Draw canvas

Render the `wrapper` and `bounds` element

Create scales

Create scales for every data-to-physical attribute in our chart

Draw data

Render your data elements

Draw peripherals

Render your axes, labels, legends, annotations, etc

Set up interactions

Initialize event listeners and create interaction behavior

Checklist

C. SVG elements cheat sheet

SVG Elements

✓ can use CSS transitions

svg

The Grand Poobah.

Use this to surround all other SVG elements or to create a new coordinate system.

rect

✓ x,y ✓
width ✓
height ✓

defs

The definitions element.

Use this to store elements to be used elsewhere. For example:

line

x1,y1
x2,y2

clipPath

Store in defs. reference: url(#id)

Used to clip other elements outside of its children elements' shape.

circle

cx,cy ✓
r ✓

linearGradient, radialGradient

Store in defs. reference: url(#id)

Used to define a gradient, using:

stop

offset
✓stop-color
✓stop-opacity

g

A container to group other SVG elements.

Similar to an HTML <div>.

polygon

points

path

d ✓

ellipse

cx,cy ✓
rx ✓
ry ✓

text

The only way to create text within SVG.

x,y
dx
dy
rotate
textLength
lengthAdjust

SVG elements cheat sheet

Changelog

Revision 8

06-26-2019

Chapter 1

Fix a typo - update the id to wrapper

Thanks to Anne W for reporting

Chapter 3

Add a missing space

Thanks to Bruno Z for reporting

Chapter 12

Update old variable names

Thanks to Dohyeon for reporting

Chapter 14

Make Typescript improvements and improve consistency

Thanks to Jacob C for reporting

Revision 7

06-14-2019

Chapter 3

Fix “48 days in our dataset” typo

Update text to make it clear that we want to add code before the end of our function

Clarify that we'll be working on the single histogram code for the **Accessibility** section

Thanks to Steve for reporting

Chapter 6

Add missing `maxChange` definition line

Fix “Zambia” typo

Thanks to Jose M. for reporting

Chapter 13

Rename `useAccessor()` to `callAccessor()` to prevent from triggering hooks lint rules

Thanks to Andreas S. for reporting

Chapter 14

Upgrade to Angular 8

Revision 6

06-06-2019

Chapter 2

Replace old accessor name (`precipitationAccessor` to `colorAccessor`)

Thanks to Steve for reporting

Chapter 5

Update line example to work around Firefox bug

Thanks to Hanno P. for reporting

Revision 5

06-03-2019

Chapter 5

Fix text to match the updated example using d3-delaunay instead of d3-voronoi

Thanks to Miguel C. for reporting

Chapter 6

Fix missing boundedWidth/boundedHeight code in Chapter text

Thanks to Christian H. for reporting

Revision 4

05-24-2019

Chapter 1

Fix a typo and create a `<g>` element for our y axis.

Thanks to Steve L. for reporting

Define `dimensions.boundedHeight`

Update images to match the `wrapper` id

Remove extra svg code from `/draft/index.html`

Thanks to Steve for reporting

Revision 3

05-17-2019

Chapter 6

Add `data_bank_data.csv` files (csv files were listed in the `.gitignore` file).

Thanks to Guillaume C. for reporting

Chapter 11

Switch the `temperatureColorScale` to a sequential scale instead of a linear scale.

Thanks to Guillaume C. for reporting

Chapter 13

Reset the React dashboard to its initial state.

Thanks to Fred T. for reporting

Appendix

Add extension to `python get_weather_data.py` command.

Thanks to Steve for reporting