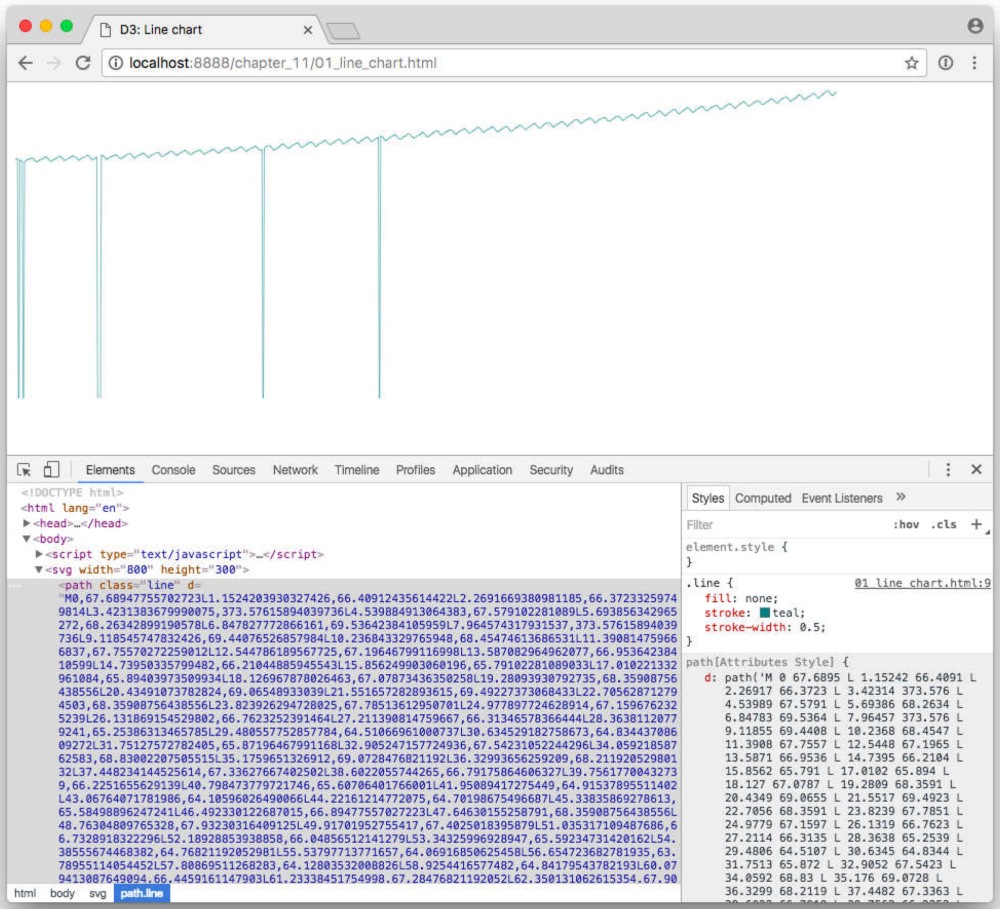
CHAPTER 11

Using Paths

`path` elements are SVG's answer to drawing irregular forms. Anything that's not a `rect`, `circle`, or another simple shape can be drawn as a `path`. The catch is that the syntax for defining `path` values is not particularly human-friendly. For example, here is a line that we'll generate from data in this chapter. Note the `path` syntax, as specified in the element's `d` attribute and shown in Figure 11-1.



```
<!DOCTYPE html>
<html lang="en">
▶<head>…</head>
▼<body>
  ▶<script type="text/javascript">…</script>
  ▼<svg width="800" height="300">
      <path class="line" d=
      "M0,67.68947755702723L1.1524203930327426,66.40912435614422L2.2691669380981185,66.3723325974
      9814L3.4231383679990075,373.57615894039736L4.539884913064383,67.579102281089L5.693856342965
      272,68.26342899190578L6.847827772866161,69.53642384105959L7.964574317931537,373.57615894039
      736L9.118545747832426,69.44076526857984L10.236843329765948,68.45474613686531L11.39081475966
      6837,67.75570272259012L12.544786189567725,67.19646799116998L13.587082964962077,66.953642384
      10599L14.73950335799482,66.21044885945543L15.856249903060196,65.79102281089033L17.010221332
      961084,65.89403973509934L18.126967878026463,67.07873436350258L19.28093930792735,68.35908756
      438556L20.43491073782824,69.06548933039L21.551657282893615,69.49227373068433L22.70562871279
      4503,68.35908756438556L23.823926294728025,67.78513612950701L24.977897724628914,67.159676232
      5239L26.131869154529802,66.7623252391464L27.211390814759667,66.31346578366444L28.3638112077
      9241,65.25386313465785L29.480557752857784,64.51066961000737L30.634529182758673,64.834437086
      09272L31.75127572782405,65.87196467991168L32.905247157724936,67.54231052244296L34.059218587
      62583,68.83002207505515L35.1759651326912,69.0728476821192L36.32993656259209,68.211920529801
      32L37.448234144525614,67.33627667402502L38.6022055744265,66.79175864606327L39.7561770043273
      9,66.2251655629139L40.798473779721746,65.60706401766001L41.95089417275449,64.91537895511402
      L43.06764071781986,64.10596026490066L44.22161214772075,64.70198675496687L45.33835869278613,
      65.58498896247241L46.492330122687015,66.89477557027223L47.64630155258791,68.35908756438556L
      48.76304809765328,67.93230316409125L49.91701952755417,67.4025018395879L51.035317109487686,6
      6.7328918322296L52.18928853938858,66.04856512141279L53.34325996928947,65.59234731420162L54.
      38555674468382,64.76821192052981L55.53797713771657,64.06916850625458L56.654723682781935,63.
      78955114054452L57.80869511268283,64.12803532008826L58.9254416577482,64.84179543782193L60.07
      9413087649094,66.4459161147903L61.23338451754998,67.28476821192052L62.350131062615354,67.90
```

```
element.style {
}
.line {                          01 line chart.html:9
   fill: none;
   stroke: 🟩teal;
   stroke-width: 0.5;
}
path[Attributes Style] {
   d: path('M 0 67.6895 L 1.15242 66.4091 L
      2.26917 66.3723 L 3.42314 373.576 L
      4.53989 67.5791 L 5.69386 68.2634 L
      6.84783 69.5364 L 7.96457 373.576 L
      9.11855 69.4408 L 10.2368 68.4547 L
      11.3908 67.7557 L 12.5448 67.1965 L
      13.5871 66.9536 L 14.7395 66.2104 L
      15.8562 65.791 L 17.0102 65.894 L
      18.127 67.0787 L 19.2809 68.3591 L
      20.4349 69.0655 L 21.5517 69.4923 L
      22.7056 68.3591 L 23.8239 67.7851 L
      24.9779 67.1597 L 26.1319 66.7623 L
      27.2114 66.3135 L 28.3638 65.2539 L
      29.4806 64.5107 L 30.6345 64.8344 L
      31.7513 65.872 L 32.9052 67.5423 L
      34.0592 68.83 L 35.176 69.0728 L
      36.3299 68.2119 L 37.4482 67.3363 L
```

Figure 11-1. A path and its d attribute

If you can read that, then you don't need this book.

Fortunately, D3 has lots of built-in functions that generate `path`s for you. You've already met the axis functions, which express scales as `path`, `line`, and `text` elements. In later chapters, you'll learn about `d3.arc()` and `d3.geoPath()`, both of which also generate `path`s for different purposes. In this chapter, we'll cover two other common uses of `path`s: drawing line and area charts.

Line Charts

Let's start with a simple line chart. Actually generating the line is quite simple, but we need some data in place first. For this chapter, I'm going to use a real-world dataset.

Data Preparation

Line charts are great for time series, so I've decided to chart carbon dioxide measurements over time. I've downloaded the "Mauna Loa CO_2 monthly mean data", as provided by the National Oceanic & Atmospheric Administration's Earth System Research Laboratory. (See the *README.md* in this chapter's examples for details and a full citation.)

This data includes monthly average values of CO_2 in parts per million, as measured at the Mauna Loa Observatory in Hawaii. An excerpt of the raw-text file looks like this:

```
# CO2 expressed as a mole fraction in dry air, micromol/mol, abbreviated as ppm
#
#  (-99.99 missing data;  -1 no data for #daily means in month)
#
#            decimal     average   interpolated    trend    #days
#             date                                (season corr)
1958   3    1958.208      315.71      315.71      314.62     -1
1958   4    1958.292      317.45      317.45      315.29     -1
1958   5    1958.375      317.50      317.50      314.71     -1
1958   6    1958.458      -99.99      317.10      314.85     -1
1958   7    1958.542      315.86      315.86      314.98     -1
1958   8    1958.625      314.93      314.93      315.94     -1
1958   9    1958.708      313.20      313.20      315.91     -1
1958  10    1958.792      -99.99      312.66      315.61     -1
1958  11    1958.875      313.33      313.33      315.31     -1
1958  12    1958.958      314.67      314.67      315.61     -1
1959   1    1959.042      315.62      315.62      315.70     -1
1959   2    1959.125      316.38      316.38      315.88     -1
1959   3    1959.208      316.71      316.71      315.62     -1
1959   4    1959.292      317.72      317.72      315.56     -1
1959   5    1959.375      318.29      318.29      315.50     -1
1959   6    1959.458      318.15      318.15      315.92     -1
1959   7    1959.542      316.54      316.54      315.66     -1
1959   8    1959.625      314.80      314.80      315.81     -1
1959   9    1959.708      313.84      313.84      316.55     -1
1959  10    1959.792      313.26      313.26      316.19     -1
1959  11    1959.875      314.80      314.80      316.78     -1
1959  12    1959.958      315.58      315.58      316.52     -1
1960   1    1960.042      316.43      316.43      316.51     -1
```

Note that the first column indicates a year, while the second indicates a month (1–12). This version of the file contains values from March 1958 through January 2017. The fourth column, "average," is the CO_2 value we're interested in.

You probably noticed that these values aren't comma-separated; rather, they are provided in fixed-width columns in an otherwise unformatted text file. D3 needs a little more structure than this, so my first step is to wrangle this into CSV form, using the free, amazing tool OpenRefine.

I won't detail the steps involved here, as there are other books and tutorials on using OpenRefine. In short, I cut out everything but the three relevant columns, and my resulting file, *mauna_loa_co2_monthly_averages.csv*, looks like this:

```
year,month,average
1958,3,315.71
1958,4,317.45
1958,5,317.5
1958,6,-99.99
1958,7,315.86
1958,8,314.93
1958,9,313.2
1958,10,-99.99
1958,11,313.33
1958,12,314.67
1959,1,315.62
1959,2,316.38
1959,3,316.71
1959,4,317.72
1959,5,318.29
1959,6,318.15
1959,7,316.54
1959,8,314.8
1959,9,313.84
1959,10,313.26
1959,11,314.8
1959,12,315.58
1960,1,316.43
…
```

Much better! Next, I'll define a row conversion function, as described in Chapter 5.

```
var rowConverter = function(d) {
    return {
        //Make a new Date object for each year + month
        date: new Date(+d.year, (+d.month - 1)),
        //Convert from string to float
        average: parseFloat(d.average)
    };
}
```

This will convert the three columns of data in our CSV into just two: `date` and `average`. The `date` consists of a `new Date()`, a new JavaScript `Date` object, into which we are passing the year and the month. The `+` operator, in this context, forces the `d.year` and `d.month` values to be typed as numbers instead of strings. And because JavaScript's month counting begins at zero (as in, 0 = January, 1 = February, and 2 = March) but the data's month counting begins at one (1 = January), I've subtracted one from each month value. (See MDN's docs on the `Date` object for more.)

The `average` value is parsed from a string to a float.

We call the row conversion function when we load in the CSV data:

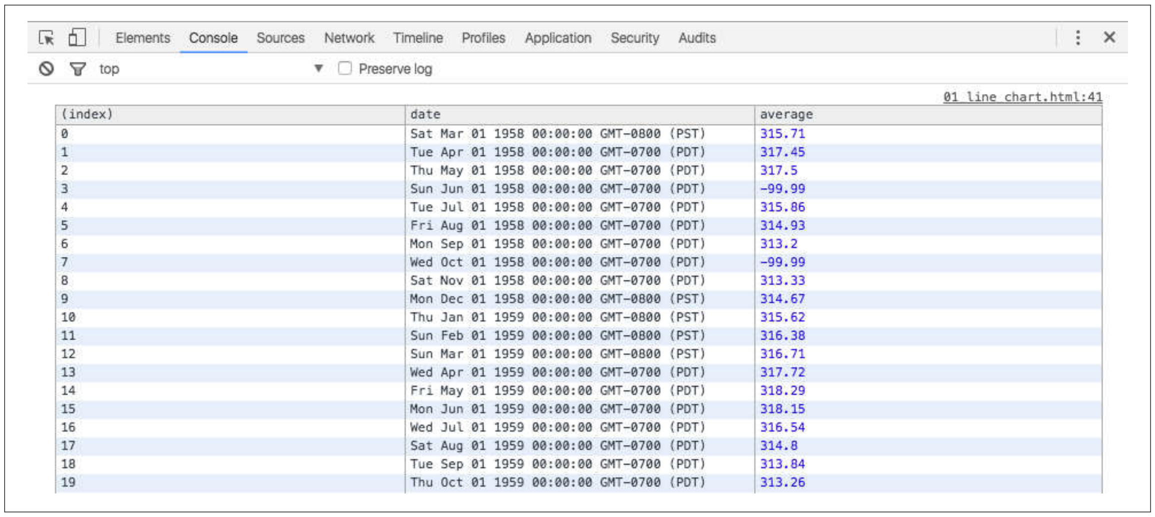
```
//Load in data
d3.csv("mauna_loa_co2_monthly_averages.csv", rowConverter, function(data) {

    var dataset = data;

    //Print data to console as table, for verification
    console.table(dataset, ["date", "average"]);

    //…
```

Feast your eyes on `console.table()`, a fancy improvement on `console.log()` that prints everything in nice, pretty columns that are easier on your brain. Here I've (optionally) specified the names of the two columns of interest: `date` and `average`. You can verify for yourself in Figure 11-2 that the values match that from our original dataset.



```
                                                                     01_line_chart.html:41
(index)   date                                          average
0         Sat Mar 01 1958 00:00:00 GMT-0800 (PST)       315.71
1         Tue Apr 01 1958 00:00:00 GMT-0700 (PDT)       317.45
2         Thu May 01 1958 00:00:00 GMT-0700 (PDT)       317.5
3         Sun Jun 01 1958 00:00:00 GMT-0700 (PDT)       -99.99
4         Tue Jul 01 1958 00:00:00 GMT-0700 (PDT)       315.86
5         Fri Aug 01 1958 00:00:00 GMT-0700 (PDT)       314.93
6         Mon Sep 01 1958 00:00:00 GMT-0700 (PDT)       313.2
7         Wed Oct 01 1958 00:00:00 GMT-0700 (PDT)       -99.99
8         Sat Nov 01 1958 00:00:00 GMT-0700 (PDT)       313.33
9         Mon Dec 01 1958 00:00:00 GMT-0800 (PST)       314.67
10        Thu Jan 01 1959 00:00:00 GMT-0800 (PST)       315.62
11        Sun Feb 01 1959 00:00:00 GMT-0800 (PST)       316.38
12        Sun Mar 01 1959 00:00:00 GMT-0800 (PST)       316.71
13        Wed Apr 01 1959 00:00:00 GMT-0700 (PDT)       317.72
14        Fri May 01 1959 00:00:00 GMT-0700 (PDT)       318.29
15        Mon Jun 01 1959 00:00:00 GMT-0700 (PDT)       318.15
16        Wed Jul 01 1959 00:00:00 GMT-0700 (PDT)       316.54
17        Sat Aug 01 1959 00:00:00 GMT-0700 (PDT)       314.8
18        Tue Sep 01 1959 00:00:00 GMT-0700 (PDT)       313.84
19        Thu Oct 01 1959 00:00:00 GMT-0700 (PDT)       313.26
```

Figure 11-2. Using console.table() to verify data

Isn't it beautiful?

Scale Setup

The process of setting up our scales should be familiar by now. First, `xScale` will handle time, and `yScale` handles the CO_2 values. I've set a zero baseline (low domain value of 0) to start.

```
    xScale = d3.scaleTime()
                   .domain([
                       d3.min(dataset, function(d) { return d.date; }),
                       d3.max(dataset, function(d) { return d.date; })
                   ])
                   .range([0, w]);
```

```
yScale = d3.scaleLinear()
           .domain([0, d3.max(dataset, function(d) { return d.average; })])
           .range([h, 0]);
```

Line 'em Up

With our data and scales in place, finally we get to draw a line! (It's the small things in life…)

Start by defining a line generator function, much as we defined axis generator functions to make axes. To do this, we call `d3.line()`, making sure to specify x and y accessors.

```
//Define line generator
var line = d3.line()
             .x(function(d) { return xScale(d.date); })
             .y(function(d) { return yScale(d.average); });
```

The x and y accessors tell the line generator how to decide *where* to place each point on the line. Note that for x, we specify the scaled value of `d.date`, and y gets the scaled value of `d.average`. Later, when the line generator is called, it will look for the bound data and loop through each value, using the accessor logic here to calculate where to position each point. Drawing the line itself is really just a matter of connecting those carefully positioned dots.

For this barebones example, I then create the SVG element and finally append a new `path` element.

```
//Create SVG element
var svg = d3.select("body")
            .append("svg")
            .attr("width", w)
            .attr("height", h);

//Create line
svg.append("path")
   .datum(dataset)
   .attr("class", "line")
   .attr("d", line);
```

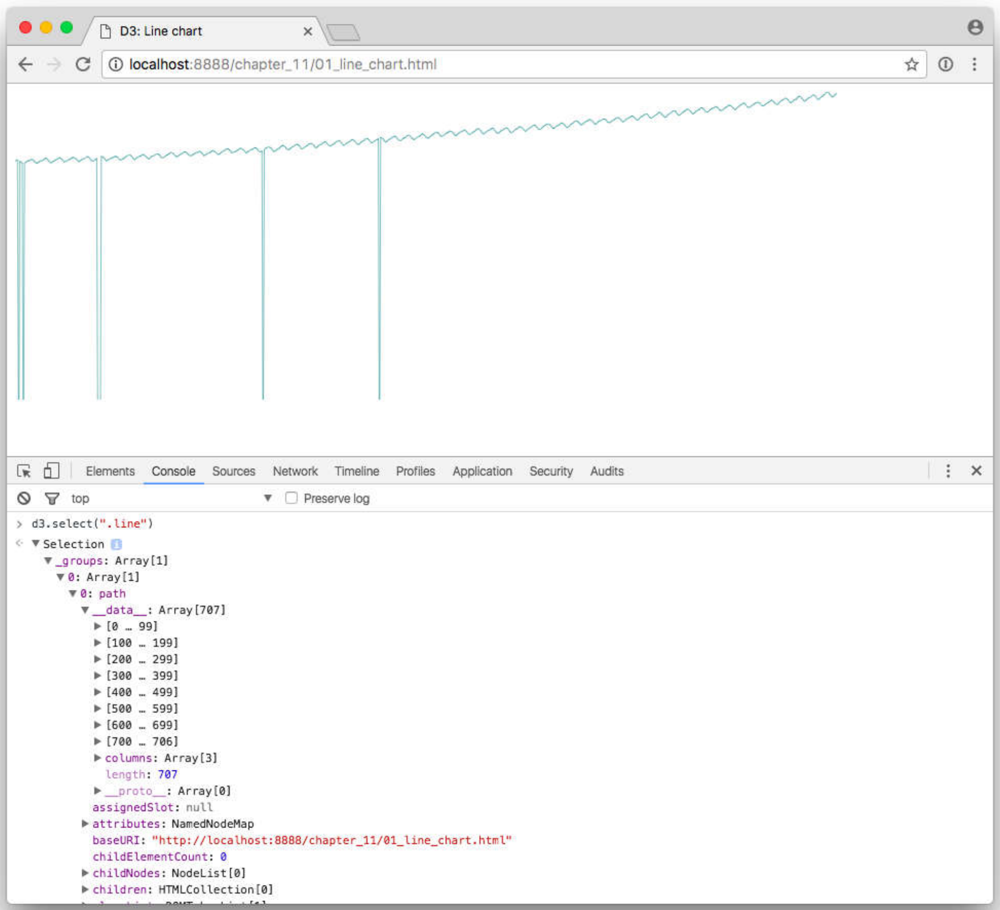
Huh? What happened to your old friend, the selectAll/data/enter/append pattern?

Until now, our examples involved one graphical mark corresponding to one data value (or one "row" of related values), as in our bar charts and scatterplots. A line chart, however, calls for a *single* graphical mark that represents *many* data values. We can skip the selectAll/data/enter/append pattern, because we already know how many new marks we need: just one.

Instead of using `data()` to bind each value in our `dataset` array to a different element, we use `datum()`, the method for binding a *single* data value to a single element. The entire `dataset` array is bound to the new `path` we just created.

Following that, we assign a class of `line` (to enable easy CSS selection and styling) and then—finally—set a `d` attribute, passing in our line generator function as an argument. Since the data has already been bound to the `path`, the line generator simply grabs that data, plots the points as we specified, and draws a line connecting them. This produces the output we saw in Figure 11-1, at the start of this chapter. The final code is in *01_line_chart.html*.

To verify how the entire `dataset` was bound to one element, try selecting that line in the console using `d3.select(".line")`. Note the complete, 707-value-long array that appears under `__data__` in Figure 11-3.



```
> d3.select(".line")
< ▼Selection
    ▼_groups: Array[1]
      ▼0: Array[1]
        ▼0: path
          ▼__data__: Array[707]
            ▶[0 … 99]
            ▶[100 … 199]
            ▶[200 … 299]
            ▶[300 … 399]
            ▶[400 … 499]
            ▶[500 … 599]
            ▶[600 … 699]
            ▶[700 … 706]
            ▶columns: Array[3]
             length: 707
            ▶__proto__: Array[0]
           assignedSlot: null
          ▶attributes: NamedNodeMap
           baseURI: "http://localhost:8888/chapter_11/01_line_chart.html"
           childElementCount: 0
          ▶childNodes: NodeList[0]
          ▶children: HTMLCollection[0]
```

Figure 11-3. Verifying the data array bound to a single path element

Dealing with Missing Data

In *02_line_chart_axes.html*, I've added some padding on the left and bottom edges plus axes, as shown in Figure 11-4.

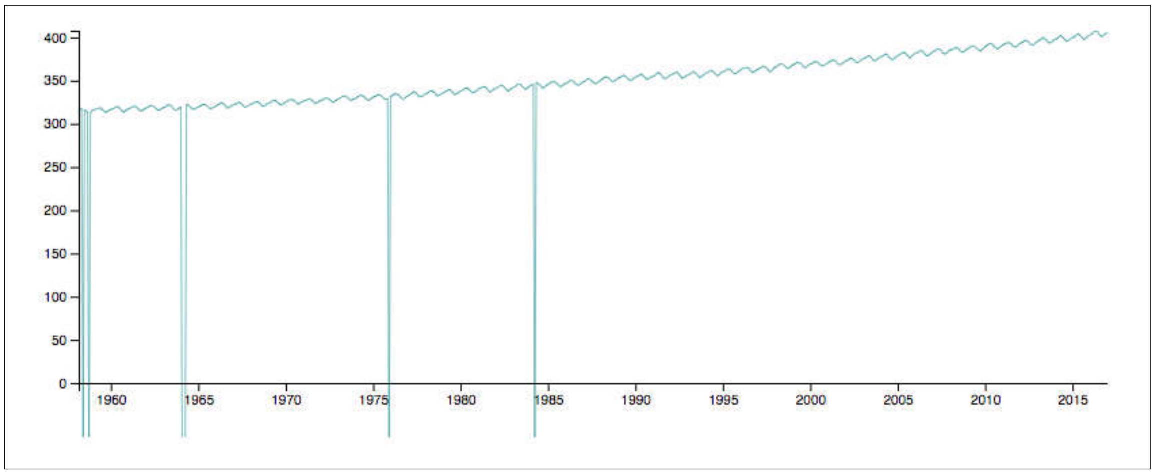


Figure 11-4. Added padding and axes

Now, with the axes labeled, it's time for a sanity check: could CO_2 levels *really* have dropped below zero, as those downward spikes seem to indicate? (Are negative CO_2 ppm values even *possible?* Answer: no.)

This spiky anomaly is explained by a note in the original datafile:

```
#  (-99.99 missing data;  -1 no data for #daily means in month)
```

Ah, so carbon-measuring machines (and even scientists themselves) are not infallible! A –99.99 value is not a true measurement, but stands in for "no data available for that month."

We could manually remove those –99.99 values from our dataset. Or we could leave the data untouched, and use the line generator's `defined()` method to determine, on the fly, whether or not each individual value is defined (or valid). `defined()` is just another configuration method, like `x()` and `y()`. If the result of its anonymous function is true, then that data value is included. If not, the value is excluded.

This simple use of `defined()` checks merely to see if any value exists at all:

```
.defined(function(d) { return d; })
```

So, if `d` exists, this will be interpreted as true.

But in our case, the –99.99 values do *exist*; it's just that we humans know they aren't valid measurements. To exclude them, we could use a comparison operator, which will return a logical result (true or false):

```
//Define line generator
var line = d3.line()
```

```
                     .defined(function(d) { return d.average >= 0; })
                     .x(function(d) { return xScale(d.date); })
                     .y(function(d) { return yScale(d.average); });
```

For every row in the dataset, if `d.average` is greater than or equal to zero, this will return true, and the value will be included. All negative values, including –99.99, will be thrown out.

See *03_line_chart_missing.html* for that final code, which renders the chart shown in Figure 11-5.

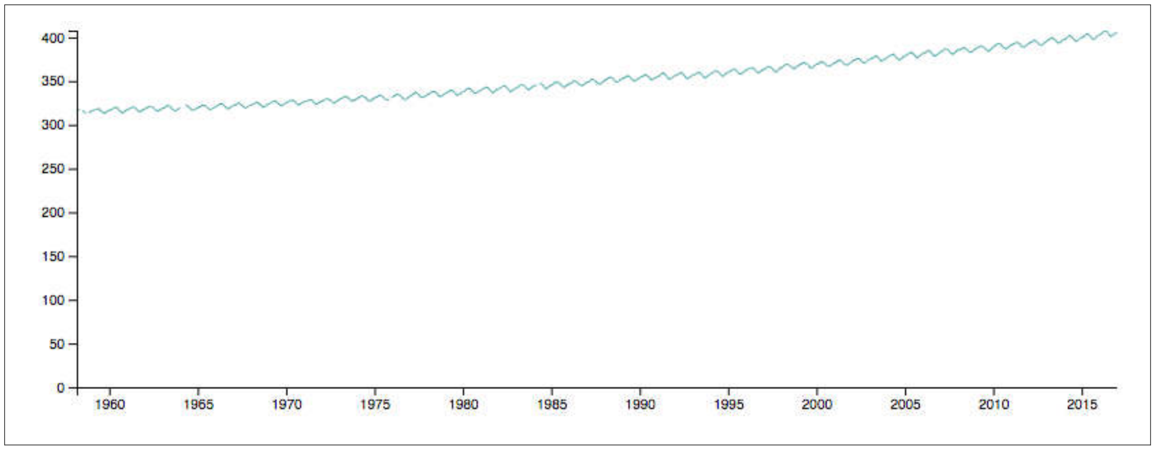


Figure 11-5. Line chart, invalid CO_2 values excluded

Note the *veeeeery tiny* gaps, where there used to be (graphically dishonest) spikes.

Actually, this would be a nice time to show off SVG's fundamentally scalable nature. Use your browser's zoom functionality to zoom in to this chart as much as possible. Note that the entire chart—consisting of vector SVG elements—remains as sharp and clear as ever. Figure 11-6 shows our chart at 500% zoom, where the gaps are clearly visible.

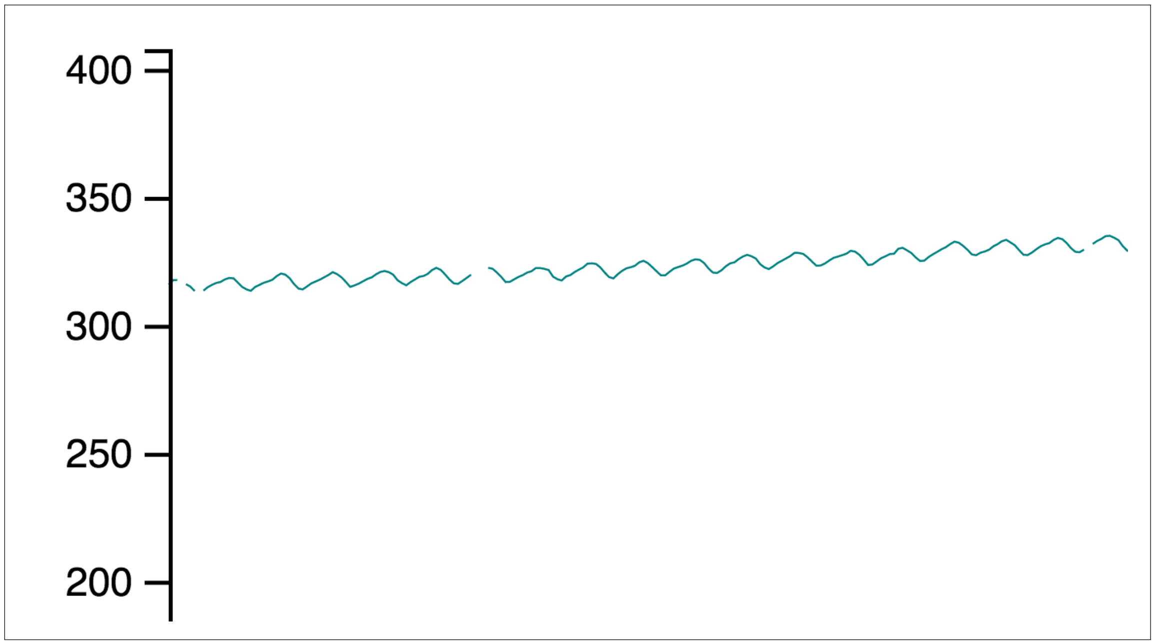


Figure 11-6. Zoomed in to reveal the gaps

Refining the Visuals

In *04_line_chart_adjusted.html*, I've adjusted the `yScale` domain to emphasize the upward trajectory, so there is no longer a zero baseline. I've also manually added a red line corresponding to 350 ppm, considered the maximum "safe" level of atmospheric CO_2 by many scientists, and used the `stroke-dasharray` property to make it a dashed line. See Figure 11-7.

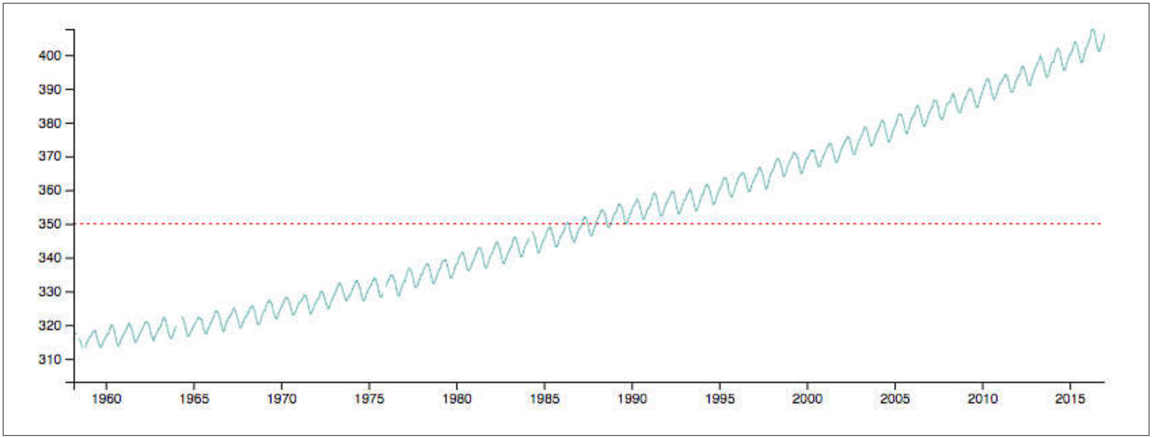


Figure 11-7. Adjusted yScale and new red line

At this point, we are dealing in visual rhetoric—design decisions that express a point of view by influencing how others interpret our chart. (For great examples of climate change–related visual rhetoric, see Duarte Design's graphics and animations, as used by Al Gore in the Academy Award–winning *An Inconvenient Truth*.)

Having acknowledged that, let's say I want to emphasize the portions of the line that are above the 350 ppm line. I could approach that by simply creating a second data-driven line. I could use `defined()` to include only data points of 350 or greater for the new red line, and to *exclude* such points for the original teal line:

```
//Define line generators
line = d3.line()
            .defined(function(d) { return d.average >= 0 && d.average <= 350; })
            .x(function(d) { return xScale(d.date); })
            .y(function(d) { return yScale(d.average); });

dangerLine = d3.line()
            .defined(function(d) { return d.average >= 350; })
            .x(function(d) { return xScale(d.date); })
            .y(function(d) { return yScale(d.average); });
```

That, plus some styling changes for each line and a new text label, produces what you see in *05_line_chart_labeled.html* and Figure 11-8.

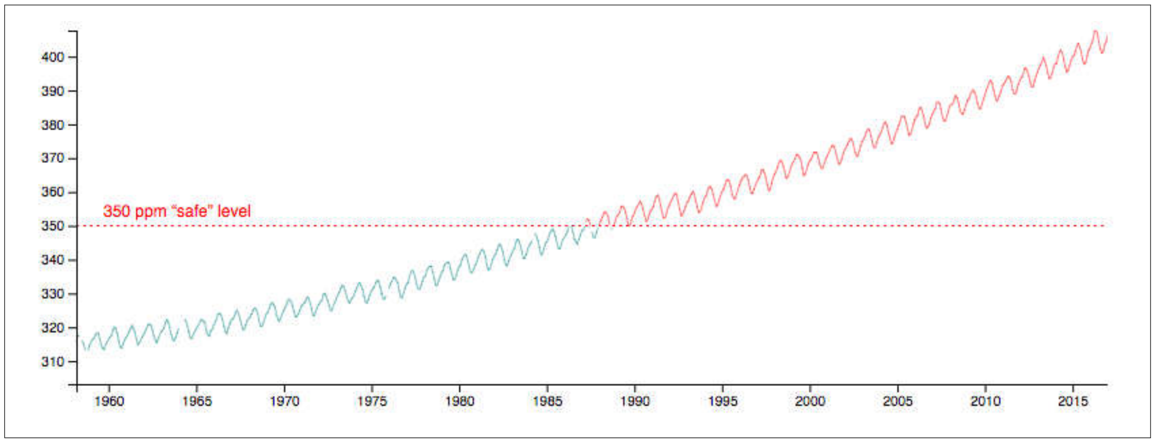


Figure 11-8. Labeled line chart with two lines, teal and red

This is just one of many possible visual solutions. A close look reveals small gaps in the line near the 350 ppm mark. An alternative approach would be to render two full lines in their entirety, but mask them using SVG clipping paths, as described in "Containing visual elements with clipping paths" on page 174 in Chapter 9.

Area Charts

Areas are not too different from lines. If a line is a series of connected x/y points, then an area is just that same line, plus a second such line (usually a flat baseline), with all the space in between filled in.

Figure 11-9 illustrates this and shows what we're working toward.

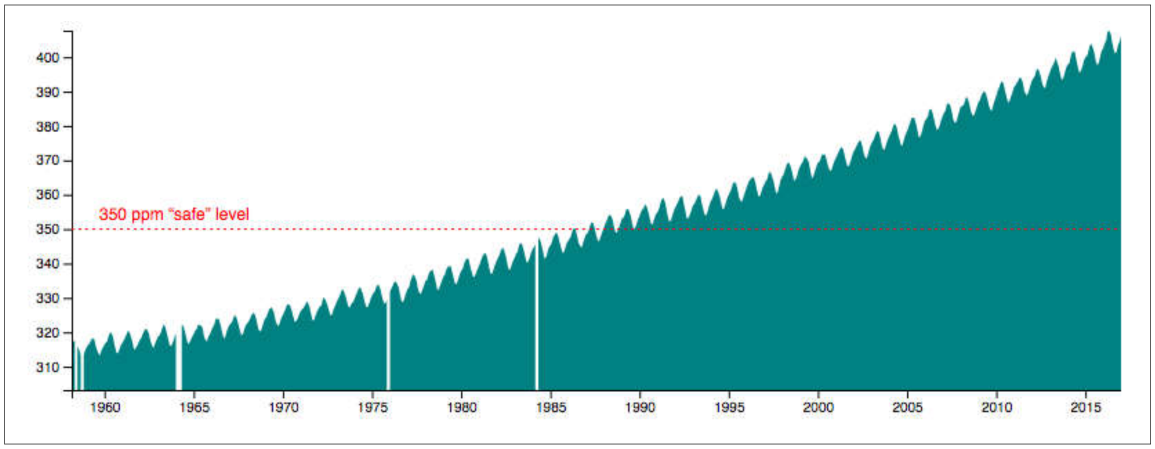


Figure 11-9. Area chart

The main difference is calling `d3.area()` to define an *area generator* instead of a line generator:

```
area = d3.area()
         .defined(function(d) { return d.average >= 0; })
         .x(function(d) { return xScale(d.date); })
         .y0(function() { return yScale.range()[0]; })
         .y1(function(d) { return yScale(d.average); });
```

You'll notice I've specified `x`, `y0`, and `y1` accessors. The `x` accessor is unchanged from earlier. `y0` represents the area's *baseline*, while `y1` represents the top, or data value. You'll notice that `y1` is the same as our earlier `y` accessor. But `y0` is new. For every data value, this accessor returns the first value in `yScale`'s range: that is, the "bottom" edge of the chart.

I don't know about you, but to me this sure *feels* like a whole lot more carbon dioxide. (Suddenly, I hope you are reading this as an ebook powered by renewable energy. If you're reading the dead-tree version, don't feel bad; use what you're learning here for good.)

One potential downside of an area chart is that, if there are missing data points, those absences will be keenly felt, as in Figure 11-9. Those slices punctuating the data don't sit right with me, but they are graphically honest. You *could* go fudge the data to patch those holes, but then how would you sleep at night?

I'll define the second area (the red one) like so:

```
dangerArea = d3.area()
               .defined(function(d) { return d.average >= 350; })
               .x(function(d) { return xScale(d.date); })
               .y0(function() { return yScale(350); })
               .y1(function(d) { return yScale(d.average); });
```

Note that, in this case, I specify a baseline value of `yScale(350)`—that is, the 350 ppm mark—because this area doesn't need to extend all the way down to the bottom of the chart. Nice!

Later in the code, we make sure to call the new area generators instead of the old line generators:

```
//Create areas
svg.append("path")
    .datum(dataset)
    .attr("class", "area")
    .attr("d", area);  // <-- Area!

svg.append("path")
    .datum(dataset)
    .attr("class", "area danger")
    .attr("d", dangerArea);  // <-- Area!
```

I also made some minor adjustments to variable names and CSS properties. But mainly you only need to change `line` to `area` and specify `y0` and `y1` accessors.

See that chart in *06_area_chart.html* and Figure 11-10.

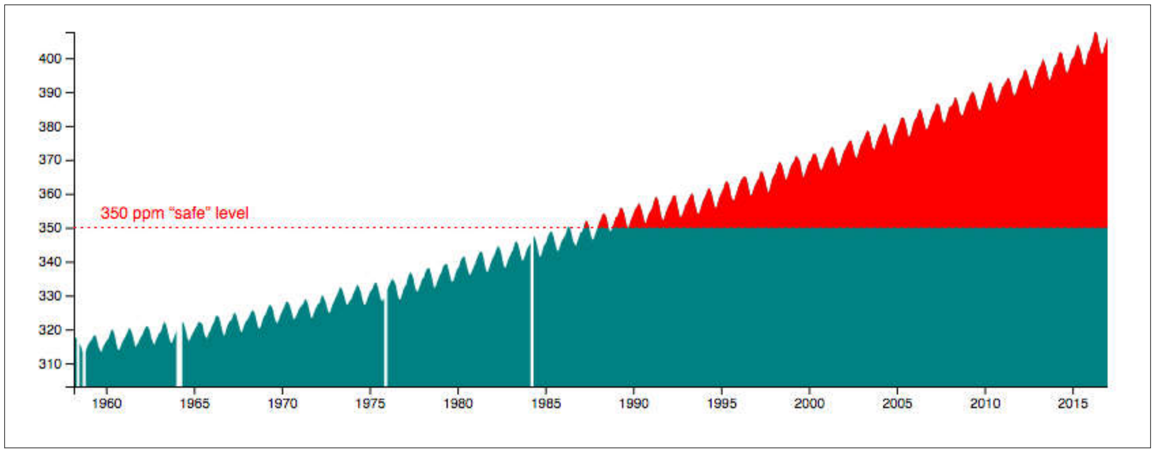


Figure 11-10. Area chart, with both areas

As a designer, I must point out that, yes, you can see the tiniest bit of the teal area bleeding around the edges of the red area. Most people won't notice this, but if you do, I'm sorry for you, but glad to meet someone else with the same affliction. In any case, you could remove that cleanly by using SVG clipping paths (again, see Chapter 9).