

D3 Tips and Tricks



Because Life's too Short to Learn All the Things

www.d3noob.org/

Subscribe to the blog to know when
updates to this document occur.

ver 0.2.c
3 January 2013

d3noob.org

Table of Contents

Introduction.....	4
What do you need to get started?.....	5
HTML.....	5
JavaScript.....	5
Cascading Style Sheets (CSS).....	6
Web Servers.....	6
Other Useful Stuff.....	7
Text Editor.....	7
Getting D3.....	7
Starting with a basic graph.....	9
HTML.....	12
CSS.....	13
D3 JavaScript.....	16
Setting up the margins and the graph area.....	17
Getting the Data.....	18
Formatting the Date / Time.....	21
Setting Scales Domains and Ranges.....	24
Setting up the Axes.....	29
Adding data to the line function.....	33
Adding the SVG Canvas.....	34
Actually Drawing Something!.....	35
Wrap Up.....	37
Things you can do with the basic graph.....	38
Adding Axis Labels.....	38
How to add a title to your graph.....	44
Smoothing out graph lines.....	45
Adding grid lines to a graph.....	51
The grid line CSS.....	51
Define the grid line functions.....	52
Draw the lines.....	54
Make a dashed line.....	55
Filling an area under the graph.....	57
CSS for an area fill.....	58
Define the area function.....	58
Draw the area.....	59
Filling an area above the line.....	60
Adding a drop shadow to allow text to stand out on graphics.....	61
CSS for white shadowy background.....	62
Drawing the white shadowy background.....	63
Adding more than one line to a graph.....	64
Multiple axes for a graph	67
How to rotate the text labels for the x Axis.....	70
Format a date / time axis with specified values.....	72
Change a line chart into a scatter plot.....	75
Adding tooltips.....	77
Transitions.....	78
Events.....	78
Get tipping.....	78

onmouseover.....	80
onmouseout.....	81
What are the predefined, named colours?.....	81
Selecting / filtering a subset of objects.....	86
Select items with an IF statement.....	87
Applying a colour gradient to a line based on value.....	89
Varying an area fill with a gradient.....	92
Using Plunker for development and hosting your D3 creations.....	94
Loading a thumbnail into Gist for bl.ocks.org d3 graphs.....	98
Setting the scene:.....	98
Enough of the scene setting. Let's git going :-).	100
Wrap up.....	102
Fault finding using Chrome.....	103
Another way to do Axis label rotation?.....	103
Threshold encoding to show high / low lines on a graph.....	103
Using a custom time format for an axis.....	103
Adding a hyperlink to an object.....	105
How Sankey Diagrams want their data formatted.....	105
Pattern of Life analysis for Earthquakes.....	106
Saving a d3.js image as an svg for inclusion in a document.....	106
Setting fixed ranges for date / time in the range	107
Hosting data using Google Docs.....	107
Adding a legend.....	107
Accessing a MySQL database as a source of data.....	107
Using Bootstrap with D3.....	107
Add a date-time picker.....	108
Create a multi-series line chart with series for each unique item in 1st column.....	108
How to run a particular function every 5 minutes.....	108
Working with GitHub, Gist and bl.ocks.org (this should be a chapter in itself).....	109
How to set up and use a github account.....	109
Making data available from GitHub.....	109
Using Markup with Gists.....	109
Installing the plug-in for bl.ocks.org for easy block viewing.....	109

Introduction

OK, weird sensation...

I never set out to write treatise on D3.

I am a simple user of this extraordinary framework and when I say simple, I really mean I had no idea how to get it to do anything when I started and I needed to do a lot of searching and trialling by error (emphasis on the errors which were entirely mine). The one thing that I did know was that the example graphics shown by Mike Bostock and others were the sort of graphical goodness that I wanted to play with.

So to get to the point of having no skills whatsoever to the point where I could begin to code up something to display data in a way I wanted, I had to capture the information as I went.

The really cool thing about this sort of process is that it doesn't need to occur all at once. You can start with no knowledge whatsoever (or pretty close) and by standing on the shoulders of others good work, you can add building blocks to improve what you're seeing and then change the blocks to adapt and improve.

For example (and this is pretty much how it started). I wanted to draw a line graph, so I imported an example and then got it running locally on my computer. Then I worked out how to change the example data for my data. Then I worked out how to move the Y axis from the right to the left. Then how to make the axis labels larger, change the tick size, make the lines fatter, change the colour, add a label, fill the area under the graph, put the graph in the centre of the page, add a glow to the text to help it stand out, put it in a framework (bootstrap), add buttons to change data sets, animate the transitions between data sets, update the data automatically when it changed, add a pan and zoom feature, turn parts of the graph into hyper-links to move to other graphs... And then I started on bar graphs :-).

The point to take away from all of this is that any one graph is just a collection of lots of blocks of code. Each block designed to carry out a function. Pick the blocks you want and implement them.

I found it was much simpler to work on one thing (block) at a time and this helped greatly to reduce the uncertainty factor when things didn't work as anticipated.

I'm not going to pretend that everything I've done while trying to build graphs employs the most elegant or efficient mechanism, but in some cases the return on the investment of the training that would require me to do things in a particular (perhaps best practices) way wasn't justified. And in the end, if it all works on the screen, I walk away happy :-). That's not to say I have deliberately ignored any best practices. I just never knew what they were. Likewise, wherever possible I have tried to make things as extensible as possible.

You will find that I have typically eschewed a simple "Do this approach" for more of a story telling exercise. This means that some explanations are longer and flowerier than might be to everyone's liking, but there you go, try to be brave :-)

What do you need to get started?

Let's be frank, my Grandmother will never put together a graphic using D3.

However, that doesn't mean that it's beyond those with a little computer savy and a willingness to have a play. Remember failure is your friend (I am fairly sure that I am also related by blood). Just learn from your mistakes and it'll all work out.

So, here in no particular order is a list of good things to know. None of which are essential, but any one (or more) of which will make your life slightly easier.

- HTML
- JavaScript
- Cascading Style Sheets (CSS)
- Web Servers

First things first **DON'T FREAK OUT!** This isn't rocket science. It's just the inter-webs.

Let's take it gently and I'll be a little more specific.

HTML

This stands for HyperText Markup Language and is the stuff that web pages are made of. Check out the definition and other stuff on Wikipedia¹ for a great overview. Just remember... All you're going to use HTML for is to hold the code that you will use to present your information. This will be as a .html (or .htm) file and they can be pretty simple (we'll look at some in a moment).



JavaScript

JavaScript² is what's called a 'scripting language'. It is the code that will be contained inside the HTML file that will make D3 do all it's fanciness. In fact, D3 is a JavaScript Library, so that's like the native language for using D3.

Knowing a little bit about this would be really good, but to be perfectly honest, I didn't know anything about it before I started. I read a book along the way (JavaScript: The Missing Manual³ from O'Reilly) and that helped with context, but the examples that are available for D3 graphics are understandable, and with a bit of trial and error, you can figure out what's going on.

In fact, most of what this collection of information's about is providing examples and explanations for the JavaScript components of D3.

1 <http://en.wikipedia.org/wiki/HTML>

2 <http://en.wikipedia.org/wiki/JavaScript>

3 JavaScript: The Missing Manual, <http://shop.oreilly.com/product/9780596515898.do>

Cascading Style Sheets (CSS)

Cascading Style Sheets⁴ (everyone appears to call them 'Style Sheets' or 'CSS') is what's known as a 'Markup' language. The job of CSS is to make the presentation of the components you will draw with D3 simpler by assigning specific styles to specific objects. One of the cool things about CSS is that it is an enormously flexible and efficient method for making everything on the screen look more consistent and when you want to change the format of something you can just change the CSS component and the whole look and feel of your graphics will change.



Full disclosure: I know CSS is a ridiculously powerful tool that would make my life easier, but I use it in a very basic (and probably painful) way. Don't judge me, just accept that the way I've learnt was what I needed to get the job done (this probably means that noob's like myself will find it easier, but where possible try and use examples that include what look like logical CSS structures)

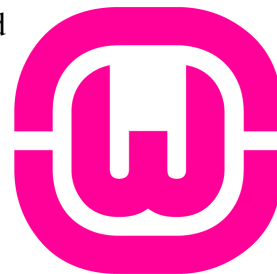
Web Servers

Ok, this can go one of two ways. If you have access to a web server and know where to put the files so that you can access them with your browser, you're on fire. If you're not quite sure, read on...

A web server will allow you to access your HTML files and will provide the structure that allows it to be displayed on a web browser. There are some simple instructions on the main D3 wiki page⁵ for setting up a local server. Or you might have access to a remote one and be able to upload your files. However, for a little more functionality and a whole lot of ease of use, I can thoroughly recommend WampServer as a free and simple way to set up a local web server that includes PHP and a MySQL database (more on those later). Go to the WampServer web page (<http://www.wampserver.com/en/>) and see if it suits you.

Throughout this document I will be describing the files and how they're laid out in a way that has suited my efforts while using WAMP, but they will work equally well on a remote server. I will explain a little more about how I arrange the files later in the 'Getting D3' section.

There are other options of course. You could host code on GitHub⁶ and present the resulting graphics on bl.ocks.org⁷. This is a great way to make sure that your code is available for peer review and sharing with the wider community.



One such alternative option that I have recently started playing with is Plunker (<http://plnkr.co/>) This is a lightweight collaborative online editing tool. It's so cool I wrote a special section for it which you can find later in this document. This is definitely worth trying if you want to use something simple without a great deal of overhead. If you like what you see, perhaps consider an alternative that provides a greater degree of capability if you go on to greater d3.js things.

4 <http://en.wikipedia.org/wiki/Css>

5 <https://github.com/mbostock/d3/wiki>

6 <https://github.com/about>

7 <http://bl.ocks.org/>

Other Useful Stuff

Text Editor

A good text editor for writing up your code will be a real boost. Don't make the fatal mistake of using an office word processor or similar. THEY WILL DOOM YOU TO A LIFE OF MISERY. They add in crazy stuff that you can't even see and never save the files in a way that can be used properly.

Preferably, you should get an editor that will provide some assistance in the form of syntax highlighting which is where the editor knows what language you are writing in (JavaScript for example) and highlights the text in a way that helps you read it.

For example, it will change text that might appear as this;

```
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
  data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
  });
```

Into something like this;

```
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
  data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
  });
```

Infinity easier to use. Trust me.

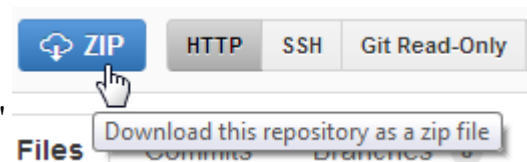
There are plenty of editors that will do the trick. I have a preference for Geany⁸, mainly because it's what I started with and it grew on me :-).



Getting D3

Luckily this is pretty easy.

Just go to the D3 repository on github⁹ and you can download the entire repository just by clicking on the 'ZIP' button.

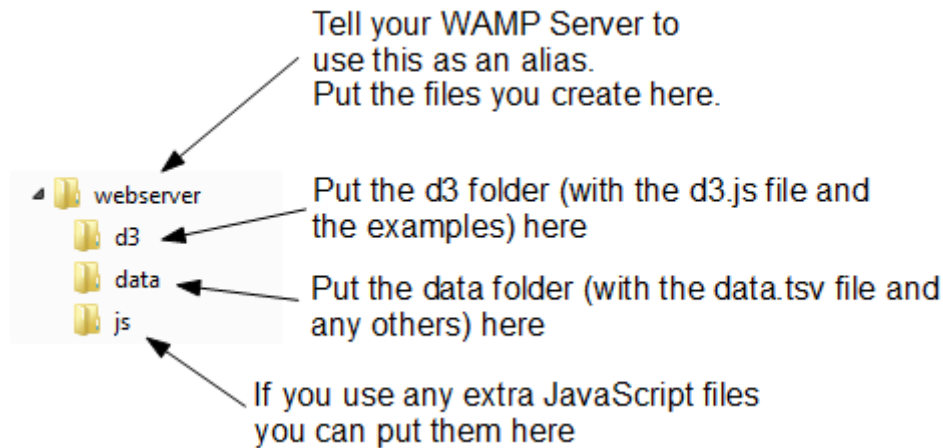


What you do with it from here is dependant on how you're getting your graphs to run. If you're working on them on your local PC, then you will want to have the d3.js file in the path that can be seen by the browser. Again, I would recommend WAMP (a local web server) to access your files locally. If you're using WAMP, then you just have to make sure that it knows to use a directory that will contain the d3 directory and you will be away.

The following image is intended to provide a very crude overview of a way you can set up the directories.

⁸ <http://www.geany.org/>

⁹ <https://github.com/mbostock/d3>



- **webservice:** Use this as your 'base' directory where you put your files that you create. That way when you open your browser you point to this directory and it allows you to access the files like a normal web site.
- **d3:** this would be your unzipped d3 directory. It contains all the examples and more importantly the d3.v3.js file that you need to get things going. You will notice in the code examples that follow there is a line like the following;

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

This tells your browser that from the file it is running (one of the graph.html files) if it goes into the 'd3' folder it will find the 'd3.v3.js' file that it can load.

- **data:** I use this directory to hold any data files that I would use for processing. For example, you will see the following line in the code examples that follow;

```
d3.tsv("data/data.tsv", function(error, data) {
```

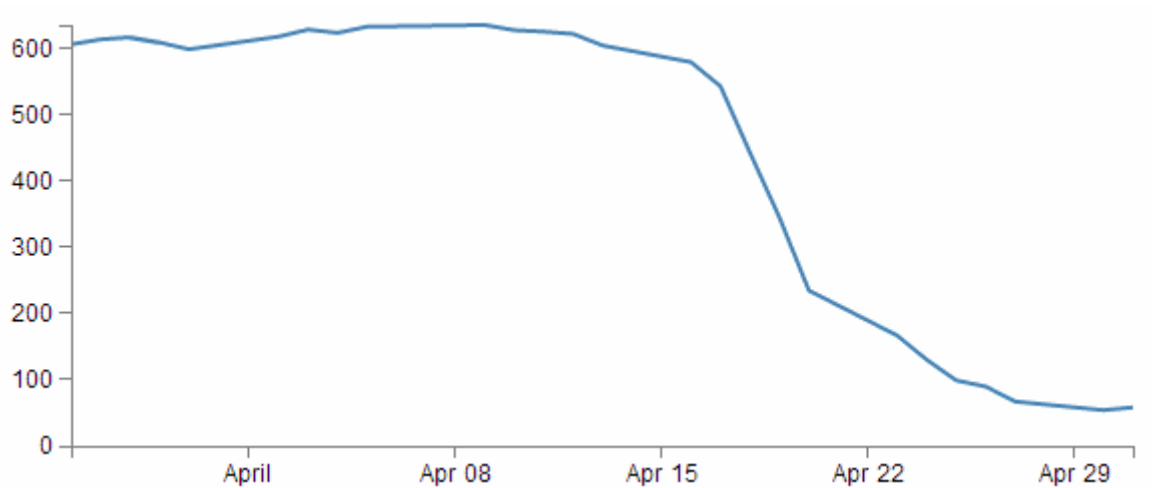
Again, that's telling the browser to go into the 'data' directory and to load the 'data.tsv' file.

- **js:** Often you will find that you will want to include other JavaScript libraries to load. This is a good place to put them.

Starting with a basic graph

How I'll make a start is to provide the full code for a simple graph and then we can explain it piece by piece.

Here's what the basic graph looks like;



And here's the code that make it happen;

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

body { font: 12px Arial;}

path {
  stroke: steelblue;
  stroke-width: 2;
  fill: none;
}

.axis path,
.axis line {
  fill: none;
  stroke: grey;
  stroke-width: 1;
  shape-rendering: crispEdges;
}

</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js"></script>

<script>

var margin = {top: 30, right: 20, bottom: 30, left: 50},
```

```

width = 600 - margin.left - margin.right,
height = 270 - margin.top - margin.bottom;

var   parseDate = d3.time.format("%d-%b-%y").parse;

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

var   xAxis = d3.svg.axis().scale(x)
      .orient("bottom").ticks(5);

var   yAxis = d3.svg.axis().scale(y)
      .orient("left").ticks(5);

var   valueline = d3.svg.line()
      .x(function(d) { return x(d.date); })
      .y(function(d) { return y(d.close); });

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

// Get the data
d3.tsv("data/data.tsv", function(error, data) {
  data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
  });

  // Scale the range of the data
  x.domain(d3.extent(data, function(d) { return d.date; }));
  y.domain([0, d3.max(data, function(d) { return d.close; })]);

  svg.append("path")           // Add the valueline path.
    .attr("d", valueline(data));

  svg.append("g")             // Add the X Axis
    .attr("class", "x axis")
    .attr("transform", "translate(0," + height + ")")
    .call(xAxis);

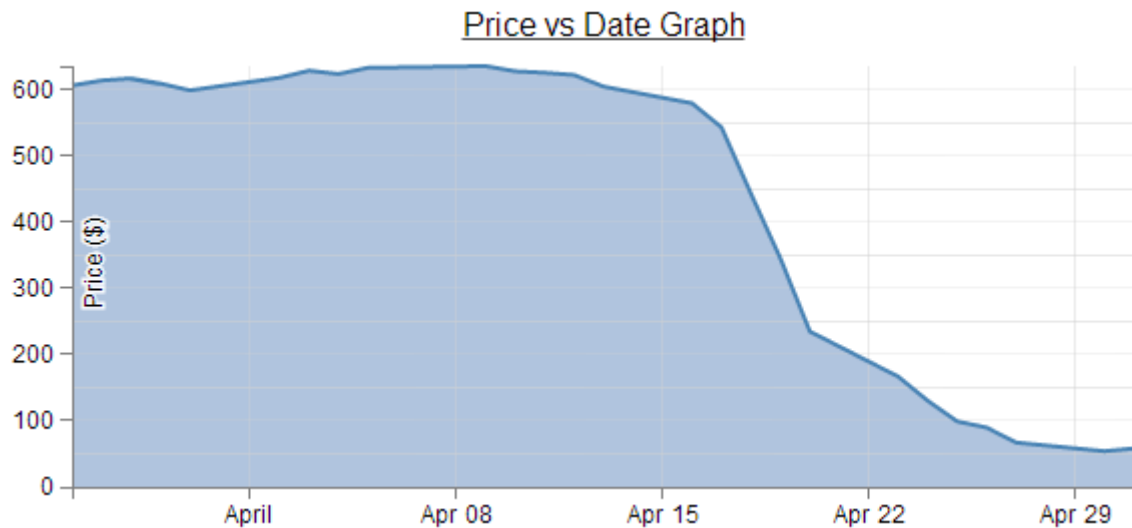
  svg.append("g")             // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);
});

```

```
</script>  
</body>
```

Once we've finished explaining these parts, we'll start looking at what we need to add in and adjust so that we can incorporate other useful functions that are completely reusable in other diagrams as well.

The end point being something hideous like the following;



I say hideous since the graph is not intended to win any beauty prizes, but there are several components to it which some people may find useful (gridlines, area fill, axis label, drop shadow for text, title, text formatting).

So, we can break the file down into component parts. I'm going to play kind of fast and loose here, but never fear, it'll all make sense.

HTML

Here's the HTML portions of the code;

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

The CSS is in here

</style>
<body>
<script type="text/javascript" src="d3/d3.v3.js"></script>

<script>

The D3 JavaScript code is here

</script>
</body>
```

Compare it with the full code. I kind of looks like a wrapping for the CSS and JavaScript. You can see that it really doesn't boil down to much at all (that doesn't mean it's not important).

There are plenty of good options for adding additional HTML stuff into this very basic part for the file, but for what we're going to be doing, we really don't need to bother too much.

One thing probably worth mentioning is the line;

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

That's the line that identifies the file that needs to be loaded to get D3 up and running. In this case the file is stored in a folder called d3 which itself is in the same directory as the main html file. The D3 file is actually called d3.v3.js which may come as a bit of a surprise. That tells us that this is version 3 of the d3.js file (the .v3. part) and while that won't be the release number (you would want to check that in the zip file you downloaded earlier), it is an indication that it is separate from the v2 release, which, as I write is the main release, but the v3 version is being prepared for general use, so I thought it would be better to work with a pre release version of the file.

Later when doing things like implementing integration with bootstrap (a pretty layout framework) we will be doing a great deal more, but for now, that's the basics done.

The two parts that we left out are the CSS and the D3 JavaScript.

CSS

The CSS is as follows

```
body { font: 12px Arial;}

path {
  stroke: steelblue;
  stroke-width: 2;
  fill: none;
}

.axis path,
.axis line {
  fill: none;
  stroke: grey;
  stroke-width: 1;
  shape-rendering: crispEdges;
}
```

So Cascading Style Sheets give you control over the look / feel / presentation of the content. The idea is to define a set of properties to objects in the web page.

They are made up of 'Rules'. Each rule has a 'selector' and a 'declaration' and each 'declaration' has a property and a value (or a group of properties and values).

For instance in the example code for this web page we have the following rule;

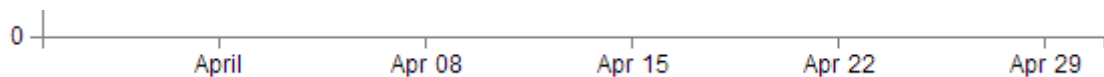
```
body { font: 12px Arial;}
```

'body' is the selector. This tells you that on the web page, this rule will apply to the 'body' of the page. This actually applies to all the portions of the web page that are contained in the 'body' portion of the HTML code (everything between <body> and </body> in the HTML bit).

{ font: 12px Arial;} is the selector portion of the rule. It only has the one declaration which is the bit that is in between the curly braces.

So font: 12px Arial; is the declaration. The property is 'font:' and the value is '12px Arial;'. This tells the web page that the font that appears in the body of the web page will be in 12 px Arial.

Sure enough if we look at the axes of the graph...

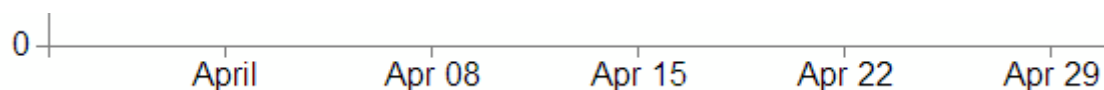


We see that the font might *actually* be 12 px Arial!

Let's try a test. I will change the Rule to the following;

```
body { font: 16px Arial;}
```

and the result is...

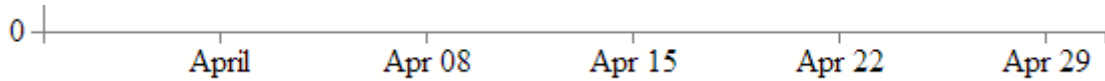


Ahh.... 16px of goodness!

And now we change it to...

```
body { font: 16px times;}
```

and we get...



Hmm... Times font.... I think we can safely say that this has had the desired effect.

So what else is there?

What about the bit that's like;

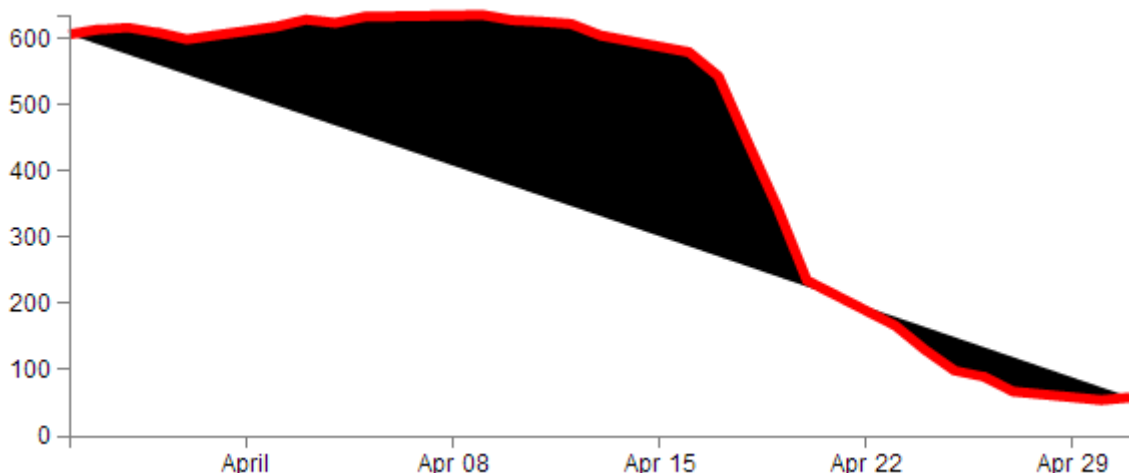
```
path {  
  stroke: steelblue;  
  stroke-width: 2;  
  fill: none;  
}
```

Well, the whole thing is one rule, 'path' is the selector. In this case, 'path' is referring to a line in the D3 drawing nomenclature.

For that selector we have three declarations. They give values for the properties of 'stroke' (in this case colour), 'stroke-width' (the width of the line) and 'fill' (we can fill a path with a block of colour).

So let's change things :-)

```
path {  
  stroke: red;  
  stroke-width: 5;  
  fill: yes;  
}
```

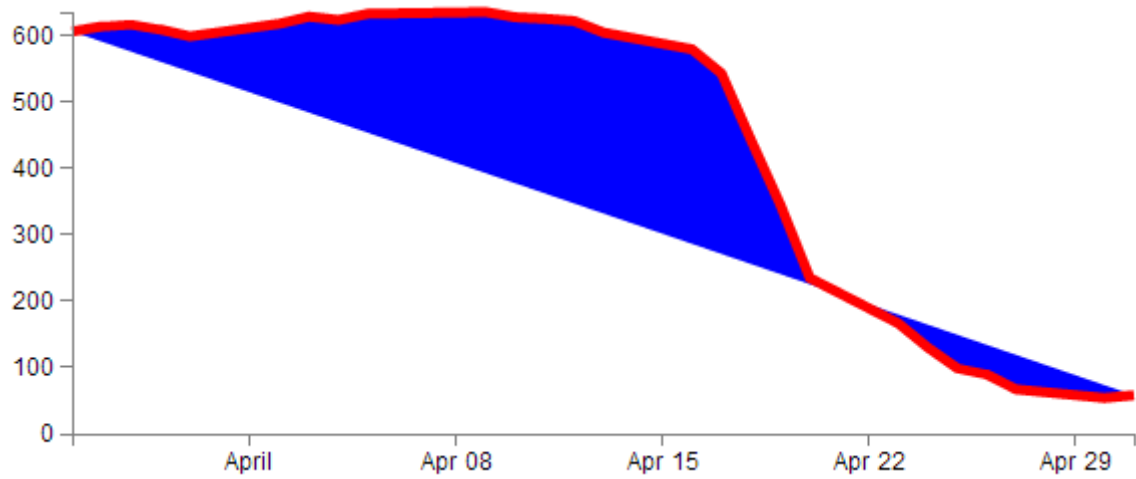


Wow! So the line is now red, it looks about 5 pixels wide and it's tried to fill the areas roughly defined by the curve with a black colour.

It ain't pretty, but it certainly did change. In fact if we go;

```
fill: blue;
```

We'll get...



So the 'fill' property looks pretty flexible. And so does CSS.

D3 JavaScript

Right, the D3 JavaScript part is as follows;

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;

var parseDate = d3.time.format("%d-%b-%y").parse;

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

var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);

var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y(d.close); });

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

// Get the data
d3.tsv("data/data.tsv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });

    // Scale the range of the data
    x.domain(d3.extent(data, function(d) { return d.date; }));
    y.domain([0, d3.max(data, function(d) { return d.close; })]);

    svg.append("path") // Add the valueline path.
        .attr("class", "line")
        .attr("d", valueline(data));

    svg.append("g") // Add the X Axis
        .attr("class", "x axis")
        .attr("transform", "translate(0," + height + ")")
        .call(xAxis);
```



```
svg.append("g")           // Add the Y Axis
    .attr("class", "y axis")
    .call(yAxis);

});
```

Again there's quite a bit of detail in the code, but it's not so long that you can't work out what's doing what.

Let's examine the blocks bit by bit to get a feel for it.

Setting up the margins and the graph area.

The part of the code responsible for defining the canvas (or the area where the graph and associated bits and pieces is placed) is this part.

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
    width = 600 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;
```

This is really (*really*) well explained on Mike Bostock's page on margin conventions here '<http://bl.ocks.org/3019563>', but at the risk of confusing you here's my crude take on it.

The first line defines the four margins that surround the block where the graph proper will be.

```
var margin = {top: 30, right: 20, bottom: 30, left: 50},
```

So there will be a border of 30 pixels at the top, 20 at the right and 30 and 50 at the bottom and left respectively. Now the cool thing about how these are set up is that they use an array to define everything. That means if you want to do calculations in the JavaScript later, you don't need to put the numbers in, you just use the variable that has been set up. In this case `margin.right = 20`!

So when we go to the next line;

```
width = 600 - margin.left - margin.right,
```

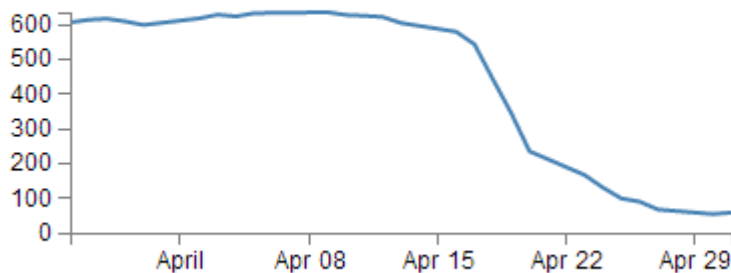
the width of the inner block of the canvas where the graph will be drawn is 600 pixels – `margin.left` – `margin.right` or 600-50-20 or 530 pixels wide. Of course now you have another variable 'width' that we can use later in the code.

Obviously the same treatment is given to height.

Another cool thing about all of this is that just because you appear to have defined separate areas for the graph and the margins, the whole area in there is available for use. It just makes it really useful to have areas set aside for the axis labels and graph labels to go without having to juggle them and the graph proper at the same time.

So, let's have a play and change some values.

```
var margin = {top: 80, right: 20, bottom: 80, left: 50},
    width = 400 - margin.left - margin.right,
    height = 270 - margin.top - margin.bottom;
```



Here we've made the graph narrower (400 pixels) but retained the left / right margins and increased the top bottom margins while maintaining the overall height of the canvas. The really cool thing that you can tell from this is that while we shrank the dimensions of the area that we had to draw the graph in, it was still able to dynamically adapt the axes and line to fit properly. That is the really cool part of this whole business. D3 is running in the background looking after the drawing of the objects, while you get to concentrate on how the data looks without too much maths!

Getting the Data

We're going to jump forward a little bit here to the bit of the JavaScript code that loads the data for the graph.

I'm going to go out of the sequence of the code here, because if you know what the data is that you're using, it will make explaining some of the other functions that are coming up much easier.

So the piece that grabs the data is this bit.

```
d3.tsv("data/data.tsv", function(error, data) {  
  data.forEach(function(d) {  
    d.date = parseDate(d.date);  
    d.close = +d.close;  
  });  
});
```

And in fact it's a combination of a few bits and another bit that isn't shown!, But let's take it one step at a time :-)

Ok... There's lots of different ways that we can get data into our web page to turn into graphics. And the way that you'll want to use will probably depend more on the format that it is in than the mechanism you want to use for importing.

For instance, if it's only a few points of data we could include the information directly in the JavaScript.

That would make it look something like;

```
var data = [  
  {date:"1-May-12",close:"58.13"},  
  {date:"30-Apr-12",close:"53.98"},  
  {date:"27-Apr-12",close:"67.00"},  
  {date:"26-Apr-12",close:"89.70"},  
  {date:"25-Apr-12",close:"99.00"}  
];
```

];

The format of the data shown above is called JSON (JavaScript Object Notation) and it's a great way to include data since it's easy for humans to read what's in there and it's easy for computers to parse the data out.

But if you've got a fair bit of data or if the data you want to include is dynamic and could be changing from one moment to the next, you'll want to load it from an external source. That's when we call on D3's 'Request' functions.

A 'Request' is a function that instructs the browser to reach out and grab some data from somewhere. It could be stored locally (on the web server) or somewhere out in the internet.

There are different types of requests depending on the type of data you want to ingest. Each type of data is formatted with different rules, so the different requests interpret those rules to make sure that the data is returned to the D3 processing in a format that it understands. You could therefore think of the different 'Requests' as translators and the different data formats as being foreign languages.

The different types of data that can be requested by D3 are;

- **text**
A plain old piece of text that can optionally be encoded in a particular way (see the D3 API¹⁰).
- **json**
This is the afore mentioned JavaScript Object Notation.
- **xml**
Extensible Markup Language is a language that is widely used for encoding documents in a human readable form.
- **html**
HyperText Markup Language is the language used for displaying web pages.
- **csv**
Comma Separated Values is a widely used format for storing data where plain text information is separated by (wait for it) commas.
- **tsv**
Tab Separated Values is a widely used format for storing data where plain text information is separated by a tab-stop character.

Details on these ingestion methods and the formats for the requests are explained well on the D3 Wiki page here - <https://github.com/mbostock/d3/wiki/Requests>. In this particular script we will look at the tsv request method.

Now, it is important to note here that this is not an exclusive list of what can be ingested. If you've got some funky data in a weird format, you can still get it in, but you will most likely need to stand up a small amount of code somewhere else in your page to do the conversion (we will look at this process when describing getting data from a MySQL database).

So, back to our request...

```
d3.tsv("data/data.tsv", function(error, data) {  
    data.forEach(function(d) {  
        d.date = parseDate(d.date);
```

¹⁰ <https://github.com/mbostock/d3/wiki/Requests>

```
d.close = +d.close;  
});
```

The first line of that piece of code invokes the d3.tsv request (d3.[tsv](#)) and then the function is pointed to the data file that should be loaded ("[data/data.tsv](#)"). This is referred to as the 'url' (unique resource locator) of the file. In this case the file is stored locally, but the url could just as easily point to a file somewhere on the Internet.

The format of the data in the data.tsv file looks a bit like this;

```
date    close  
1-May-12    58.13  
30-Apr-12    53.98  
27-Apr-12    67.00  
26-Apr-12    89.70  
25-Apr-12    99.00
```

(although the file is longer (about 26 data points)). The 'date' and the 'close' heading labels are separated by a tab as are each subsequent dates and numbers. Hence the 'tab separated values' :-).

The next part is part of the coolness of JavaScript. With the request made and the file requested the script is told to carry out a function on the data (which will now be called 'data').

```
function(error, data) {
```

Now, there are actually more things that get acted on as part of the function call, but the one we will consider here are the following lines;

```
    data.forEach(function(d) {  
        d.date = parseDate(d.date);  
        d.close = +d.close;  
    });
```

This block of code simply ensures that all the numeric values that are pulled out of the tsv file are set and formatted correctly. The first line sets the data variable that is being dealt with (called slightly confusingly 'data') and tells the block of code that for each group within the 'data' array it should carry out a function on them. That function is designated 'd'.

```
    data.forEach(function(d) {
```

The information in the array can be considered as being stored in rows with each row consisting of two values. One value for 'date' and another value for 'close'.

So the function is pulling out values of 'date' and 'close' one row at a time.

Each time it gets a value of 'data' and 'close' it carries out the following operations;

```
        d.date = parseDate(d.date);
```

This the specific value of date being looked at (d.[date](#)) into a date format that D3 can process and do stuff with via a separate function 'parseDate'. Now, the 'parseDate' function is defined in a separate part of the script, and we will examine that later. So for the moment, just be satisfied that it takes the raw date information from the tsv file in a specific row and converts it into a format that D3 can then process. That value is then re-saved in the same variable space.

The next line then sets the 'close' value to a numeric value (if it isn't already) using the '+' operator.

```
        d.close = +d.close;
```

This appears to be good practice when the format of the number being pulled out of the data may not mean that it is automatically recognised as a number. This will ensure that it is.

So, at the end of that section of code, we have gone out and picked up a file with data in it of a particular type (tab separated values) and ensured that it is formatted in a way that the rest of the script can use it correctly.

Now, the astute amongst you will have noticed that in the first line of that block of code (`d3.tsv("data/data.tsv", function(error, data) {`) we opened a normal bracket (`(`) and a curly bracket (`{`), but we never closed them. That's because they stay open until the very end of the file. That means that all those blocks that occur after the `d3.tsv` bit are referenced to the 'data' array. Or put another way, it uses 'data' to draw stuff!

But anyway, let's get back to figuring what the code is doing by jumping back to the end of the margins block.

Formatting the Date / Time.

One of the glorious things about the World is that we all do things a bit differently. One of those things is how we refer to dates and time¹¹.

In my neck of the woods, it's customary to write the date as day - month – year. E.g 23-12-2012. But in the United States the more common format would be 12-23-2012. Likewise, the data may be in formats that name the months or weekdays (E.g. January, Tuesday) or combine dates and time together (E.g. 2012-12-23 15:45:32). So, if we were to attempt to try to load in some data and to try and get D3 to recognise it as date / time information, we really need to tell it what format the date / time is in.

Now, you might be asking yourself what's the point? All you want to do is give it a number and it can sort it out somehow. Well, that is true, but if you want to really bring out the best in your data and to keep maximum flexibility in representing it on the screen, you will want to D3 to play to it's strengths. And one of those is being able to adjust dynamically with variable time values.

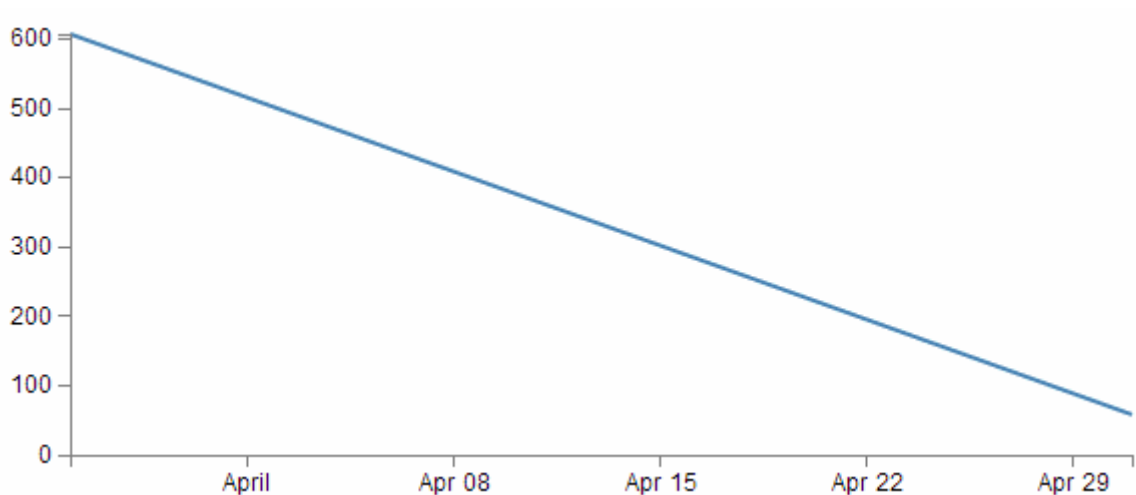
Time for a little demonstration.

I will change our data.tsv file that we use to load data to graph so that it only includes two points. The first one and the last one with a separation of a month and a bit.

date	close
1-May-12	58.13
26-Mar-12	606.98

The graph now looks like this;

¹¹ http://en.wikipedia.org/wiki/Date_format_by_country

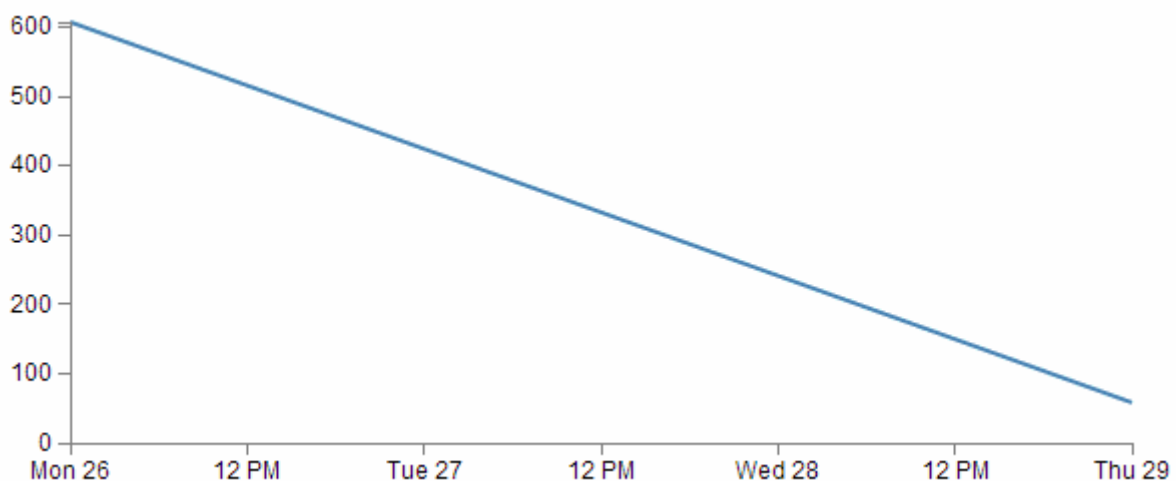


So, nothing too surprising here, a very simple graph (note the time scale on the x axis).

But now I will change the later date in the data.tsv file so that it is a lot closer to the starting date;

```
date    close
29-Mar-12  58.13
26-Mar-12  606.98
```

So, just a three day difference. Let's see what happens.



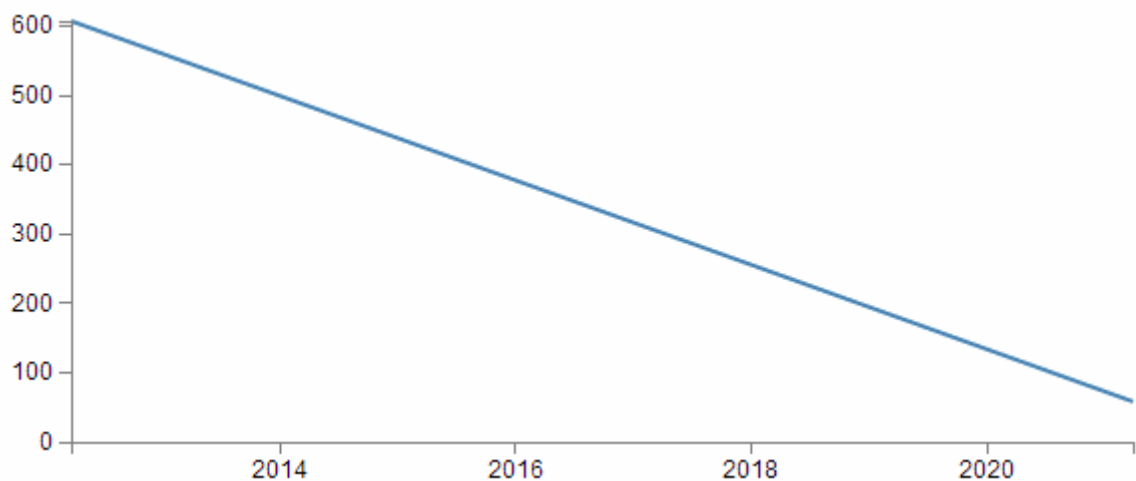
Ahh.... Not only did we not have to make any changes to our JavaScript code, but it was able to recognise the dates were closer and filled in the intervening gaps with appropriate time / day values.

Now, one more time for giggles.

This time we'll stretch the interval out by a few *years*.

```
date    close
29-Mar-21  58.13
26-Mar-12  606.98
```

and the result is...



Hopefully that's enough encouragement to impress upon you that formatting the time is a REALLY good thing to get right and trust me, it will never fail to impress :-).

So, back to formatting.

The line in the JavaScript that parses the time is the following;

```
var parseDate = d3.time.format("%d-%b-%y").parse;
```

This line is used when the `data.forEach(function(d)` portion of the code that we looked at a couple of pages back used `d.date = parseDate(d.date)` as a way to take a date in a specific format and to get it recognised by D3. In effect it said take this value that is supposedly a date and make it into a value I can understand.

The function used is the `d3.time.format(specifier)` function where the specifier in this case is the mysterious combination of characters `'%d-%b-%y'`. The good news is that these are just a combination of directives specific for the type of date we are presenting.

The `'%'` signs are used as prefixes to each separate format type and the `'-'` signs are literals for the actual `'-'` signs that appear in the date to be parsed.

The `'d'` refers to a zero-padded day of the month as a decimal number [01,31].

The `'b'` refers to an abbreviated month name.

And the `'y'` refers to the year with century as a decimal number.

If we look at a subset of the data from the `data.tsv` file we see that indeed, the dates therein are formatted in this way.

1-May-12	58.13
30-Apr-12	53.98
27-Apr-12	67.00
26-Apr-12	89.70
25-Apr-12	99.00

So that's all well and good, but what if your data isn't formatted exactly like that?

Good news. There are a heap of different formatters for different ways of telling time and you get to pick and choose what you want. Check out the Time Formatting page¹² on the D3 Wiki for the authoritative list and some great detail, but the following is the list of currently available formatters (from the d3 wiki);

¹² <https://github.com/mbostock/d3/wiki/Time-Formatting>

- %a - abbreviated weekday name.
- %A - full weekday name.
- %b - abbreviated month name.
- %B - full month name.
- %c - date and time, as "%a %b %e %H:%M:%S %Y".
- %d - zero-padded day of the month as a decimal number [01,31].
- %e - space-padded day of the month as a decimal number [1,31].
- %H - hour (24-hour clock) as a decimal number [00,23].
- %I - hour (12-hour clock) as a decimal number [01,12].
- %j - day of the year as a decimal number [001,366].
- %m - month as a decimal number [01,12].
- %M - minute as a decimal number [00,59].
- %p - either AM or PM.
- %S - second as a decimal number [00,61].
- %U - week number of the year (Sunday as the first day of the week) as a decimal number [00,53].
- %w - weekday as a decimal number [0(Sunday),6].
- %W - week number of the year (Monday as the first day of the week) as a decimal number [00,53].
- %x - date, as "%m/%d/%y".
- %X - time, as "%H:%M:%S".
- %y - year without century as a decimal number [00,99].
- %Y - year with century as a decimal number.
- %Z - time zone offset, such as "-0700".
- %% - a literal "%" character.

As an example, if you wanted to input date / time formatted as a generic MySQL 'YYYY-MM-DD HH:MM:SS' TIMESTAMP format the D3 parse script would look like;

```
parseDate = d3.time.format("%Y-%m-%d %H:%M:%S").parse;
```

Setting Scales Domains and Ranges

This is another example where if you set it up right, D3 will look after you forever.

The “Ah Ha!” moment for me in understanding ranges and scales was after reading Jerome Cukier's great page on 'd3:scales and color'.¹³ I thoroughly recommend you read it (and plenty more of the great work by Jerome) as he really does nail the description in my humble opinion. I will put my own description down here, but if it doesn't seem clear, head on over to Jerome's page.

From our basic web page we have now moved to the section that includes the following lines;

¹³ <http://www.jeromecukier.net/blog/2011/08/11/d3-scales-and-color/>


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

The purpose of these portions of the script is to ensure that the data we ingest fits onto our graph correctly. Since we have two different types of data (date/time and numeric values) they need to be treated separately (but they do essentially the same job). To examine this whole concept of scales, domains and ranges properly, we will also move slightly out of sequence and (in conjunction with the earlier scale statements) take a look at the lines of script that occur later and set the domain. They are as follows;

```
x.domain(d3.extent(data, function(d) { return d.date; }));  
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

So, the idea of scaling is to take the values of data that we have and to fit them into the space we have available.

If we have data that goes from 53.98 to 636.23 (as the data we have for 'close' in our tsv file does), but we have a graph that is 210 pixels high (height = 270 - margin.top – margin.bottom;) we clearly need to make an adjustment.

Not only that. Even though our data goes from 53.98 to 636.23, that would look slightly misleading on the graph and it should really go from 0 to a bit over 636.23.

It sound's really complicated, but let's simple it up a bit.

First we make sure that any quantity we specify on the x axis fits onto our graph.

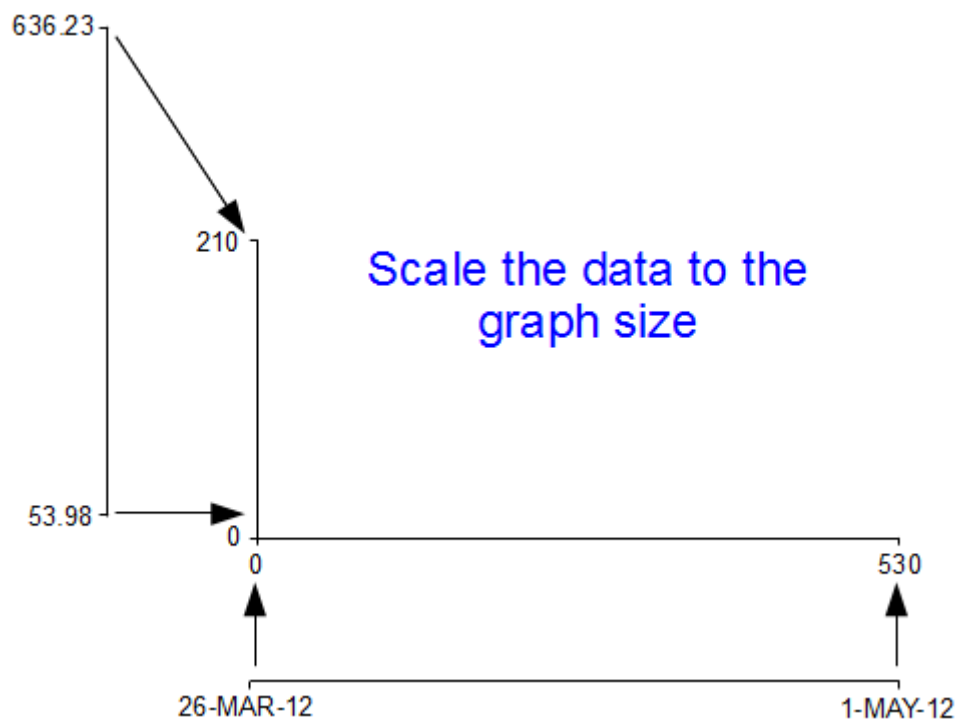
```
var x = d3.time.scale().range([0, width]);
```

Here we set our variable that will tell D3 where to draw something on the x axis. By using the `d3.time.scale()` function we make sure that D3 knows to treat the values as date / time entities (with all their ingrained peculiarities). Then we specify the range that those values will cover (`.range`) and we specify the range as being from 0 to the `width` of our graphing area (See! Setting those variables for margins and widths are starting to pay off now!).

Then we do the same for the Y axis.

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

There's a different function call (`d3.scale.linear()`) but the `.range` setting is still there. In the interests of drawing a (semi) pretty picture to try and explain, hopefully this will assist;

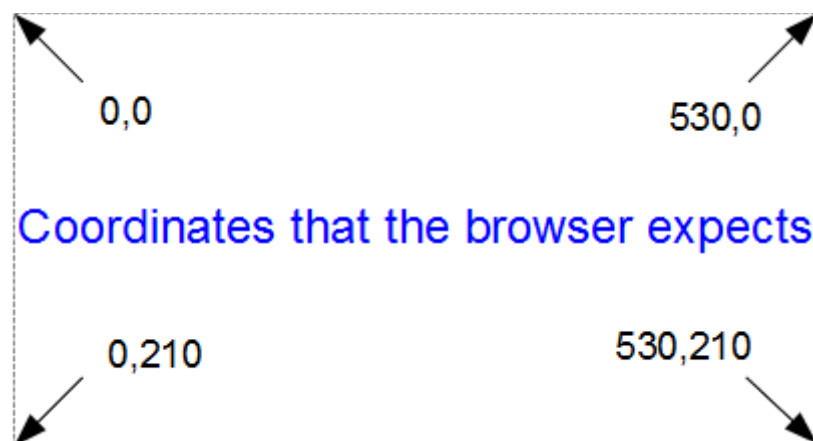


I know, I know, it's a little misleading because nowhere have we actually said to D3 this is our data from 53.98 to 636.23. All we've said is when we get the data, we'll be scaling it into this space.

Now hang on, what's going on with the `[height, 0]` part in y axis scale statement? The astute amongst you will note that for the time scale we set the range as `[0, width]` but for this one (`[height, 0]`) the values look backwards.

Well spotted.

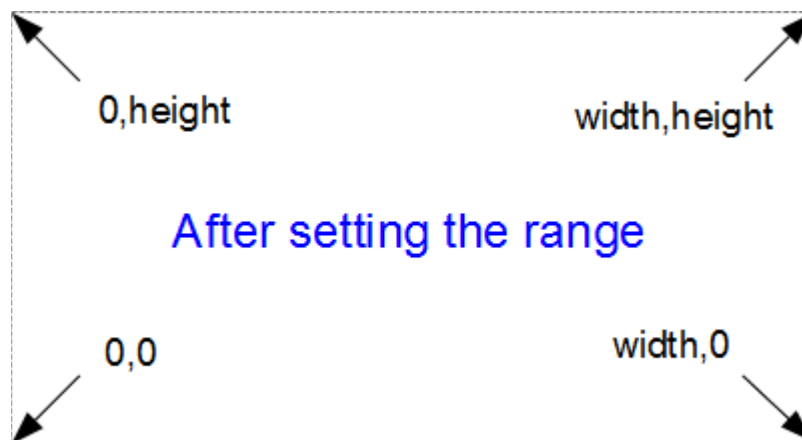
This is all to do with how the screen is laid out and referenced. Take a look at the following diagram showing how the coordinates for drawing on your screen work;



The top left hand of the screen is the origin or 0,0 point and as we go left or down the corresponding x and y values increase to the full values defined by height and width.

So that's all well and good for the time values on the x axis that will start at lower values and increase, but for the values on the y axis we're trying to go against the flow. We want the low values to be at the bottom and the high values to be at the top.

No problem. We just tell D3 via the statement `y = d3.scale.linear().range([height, 0]);` that the larger values (`height`) are at the low end of the screen (at the top) and the low values are at the bottom (as you most probably will have guessed by this stage, the `.range` statement uses the format `.range([closer_to_the_origin, further_from_the_origin])`. So when we put the `height` variable first, that is now associated at the top of the screen.



OK, so we've scaled our data to the graph size and ensured that the range of values is set appropriately, so what's with the domain part that was in the title for this section?

Come on, you remember this little piece of script don't you?

```
x.domain(d3.extent(data, function(d) { return d.date; }));  
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

While it exists in a separate part of the file from the scale / range part, it is certainly linked.

That's because there's something missing from what we have been describing so far with the set up of the data ranges for the graphs. We haven't actually told D3 what the range of the data is. That's also the reason this part of the script occurs where it does. It is within the portion where the `data.tsv` file has been loaded as 'data' and it's therefore ready to act on it.

So, the `.domain` function is designed to let D3 know what the scope of the data will be this is what is then passed to the scale function.

Looking at the first part that is setting up the x axis values, it is saying that the domain for the x axis values will be determined by the `d3.extent` function which in turn is acting on a separate function which looks through all the 'date' values that occur in the 'data' array. In this case the `.extent` function returns the minimum and maximum value in the given array.

- So `function(d) { return d.date; }` returns all the 'date' values in 'data'. This is then passed to...
- The `.extent` function that finds the maximum and minimum values in the array and then...
- The `.domain` function which returns those maximum and minimum values to D3 as the range for the x axis.

Pretty neat really. At first you might think it was slightly overly complex, but breaking the function down into these components, allows additional functionality with differing scales, values and quantities. In short, don't sweat it. It's a good thing.

Now, the x axis values are dates, so the domain for them is basically from the 26th of March 2012 till 1st of May 2012. The y axis is done slightly differently

```
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

Because the range of values desired on the y axis goes from 0 to the maximum in the data range, that's exactly what we tell D3. The '0' in the `.domain` function is the starting point and the finishing point is found by employing a separate function that sorts through all the 'close' values in the 'data' array and returns the largest one. Therefore the domain is from 0 to 636.23.

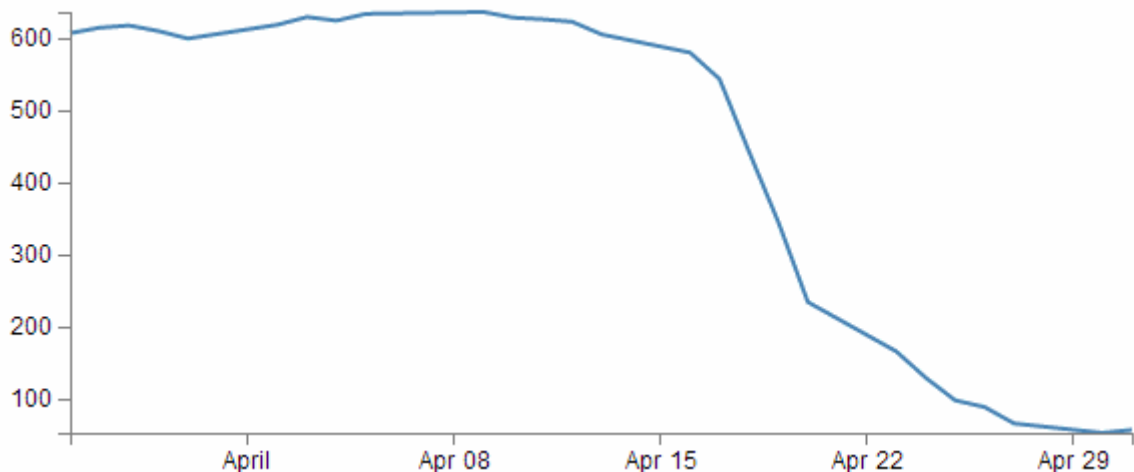
Let's try a small experiment. Let's change the y axis domain to use the `.extent` function (the same way the x axis does) to see what it produces.

So the JavaScript for the y domain will be;

```
y.domain(d3.extent(data, function(d) { return d.close; }));
```

You can see apart from a quick copy paste of the internals, all I had to change was the reference to 'close' rather than 'date'.

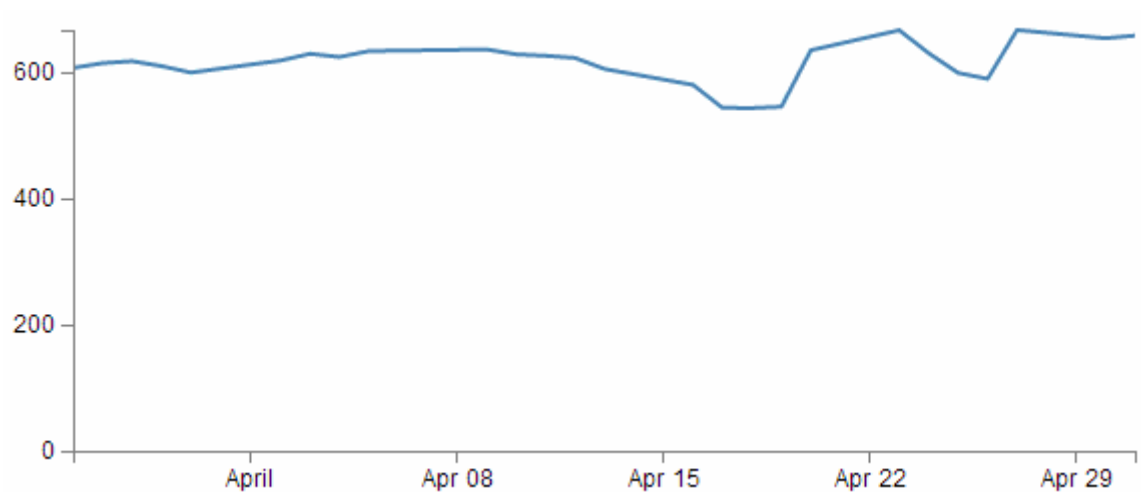
And the result is...



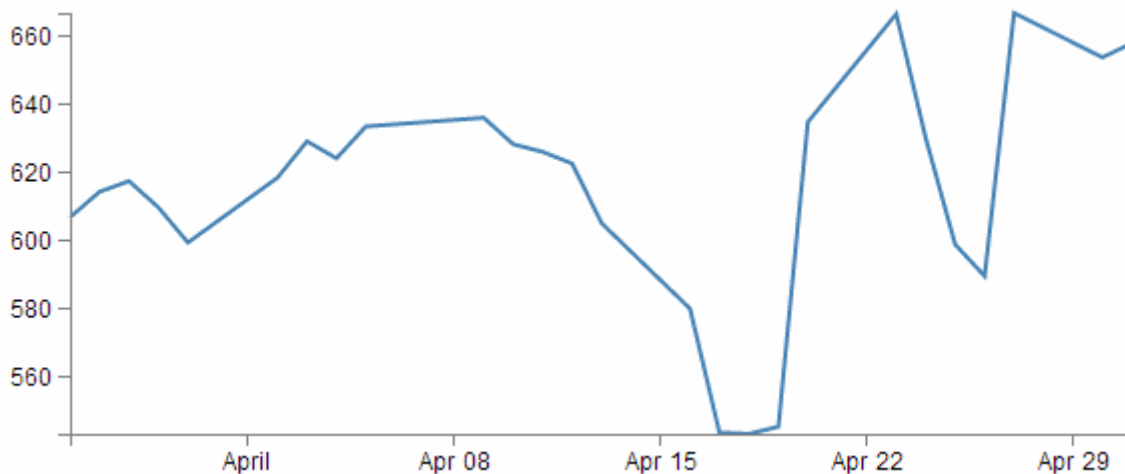
Look at that. The starting point for the y axis looks like it's pretty much on the 53.98 mark and the graph itself certainly touches the x axis where the data would indicate it should.

Now, I'm not really advocating making a graph like this since I think it looks a bit nasty (and a casual observer might be fooled into thinking that the x axis was at 0). However, this would be a useful thing to do if the data was concentrated in a narrow range of values that are quite distant from zero.

For instance, if I change the data.tsv file so that the values are represented like the following;



Then it kind of loses the ability to distinguish between values around the median of the data. But, if I put in our magic `.extent` function for the y axis and redraw the graph...



How about that?

The same data as the previous graph, but with one simple piece of the script changed and D3 takes care of the details.

Setting up the Axes

So we come to our next piece of code;

```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);

var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
```

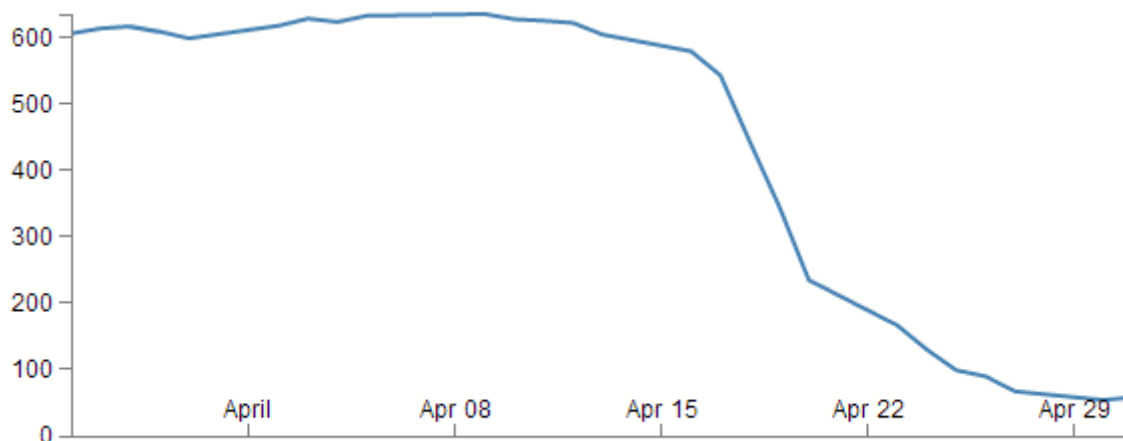
I've included both the x and y axes because they carry out the formatting in very similar ways. And it's worth noting that this is not the point where the axes get drawn. That occurs later in the piece where the `data.tsv` file has been loaded as `'data'`.

D3 has its own axis component that aims to take the fuss out of setting up and displaying the axes. So it includes a number of configurable options.

Looking first at the x axis;

```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(5);
```

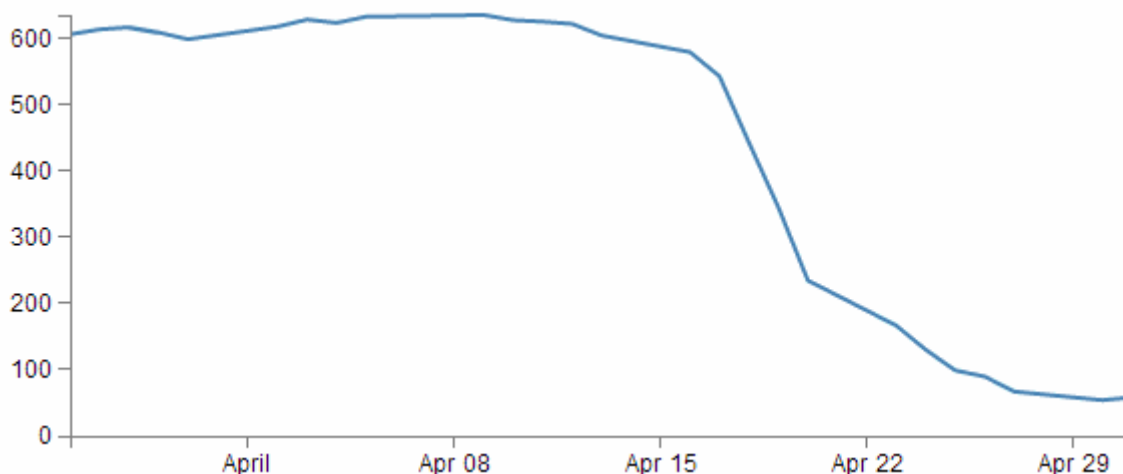
The axis function is called with `d3.svg.axis()` and then the scale is set using the x values that we setup in the scales, ranges and domains section using `.scale(x)`. Then a curious thing happens, we tell the graph to orientate itself to the bottom of the graph `.orient("bottom")`. Now if I tell you that `"bottom"` is the default setting, then you could be forgiven for thinking that technically, we don't need to specify this since it will go there anyway, but it does give us an opportunity to change it to `"top"` to see what happens;



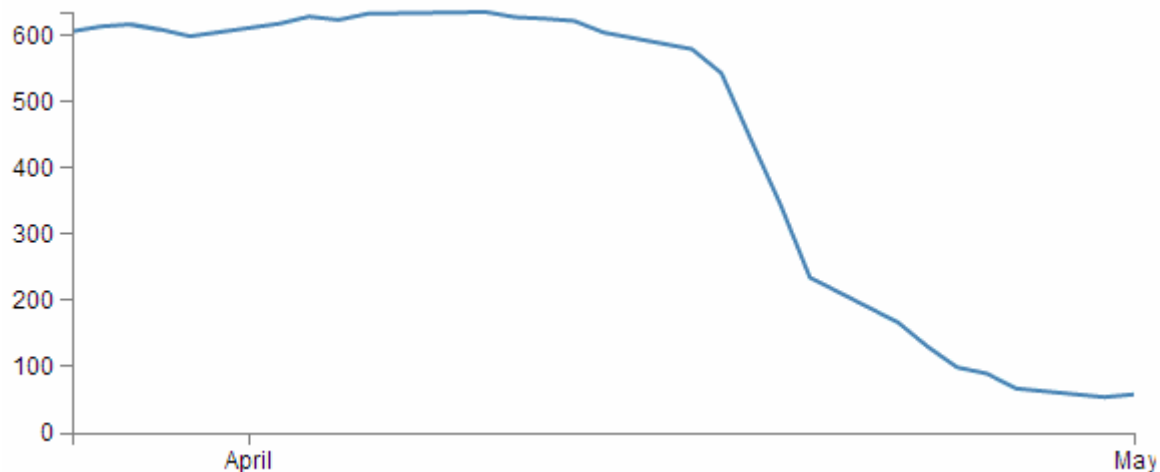
Well, I hope you didn't see that coming, because I didn't. So, it would transpire that what we're talking about there is the orientation of the values and ticks about the axis line itself. Ahh... Ok. So, useful if your x axis is at the top of your graph, but for this one, not so useful.

All right, the next part (`.ticks(5)`) sets the number of ticks on the axis. Hopefully you just did a quick count across the bottom of the previous graph and went "Yep, five ticks. Spot on". Well done if you did, but there's a little bit of a sneaky trick up D3's sleeve with the number of ticks on a graph axis.

For instance, here's what the graph looks like when the `.ticks(5)` value is changed to `.ticks(4)`.



Eh? Hang on. Isn't that some kind of mistake? There's still five ticks. Yep sure is. But wait... we can keep dropping the ticks value till we get to two and it will still be the same. At `.ticks(2)` though, we finally see a change.



How about that? At first glance that just doesn't seem right, then you have a bit of a think about it and you go “Hmm... When there were 5 ticks, they were separated by a week each, and that stayed that way till we got to a point where it could show a separation of a month.”.

So, what's going on here is that D3 is making a command decision for you as to how your ticks should be best displayed. This is great for simple graphs and indeed for the vast majority of graphs. And like all things D3, if you really need to do something bespoke and fancy, it'll let you if you know enough.

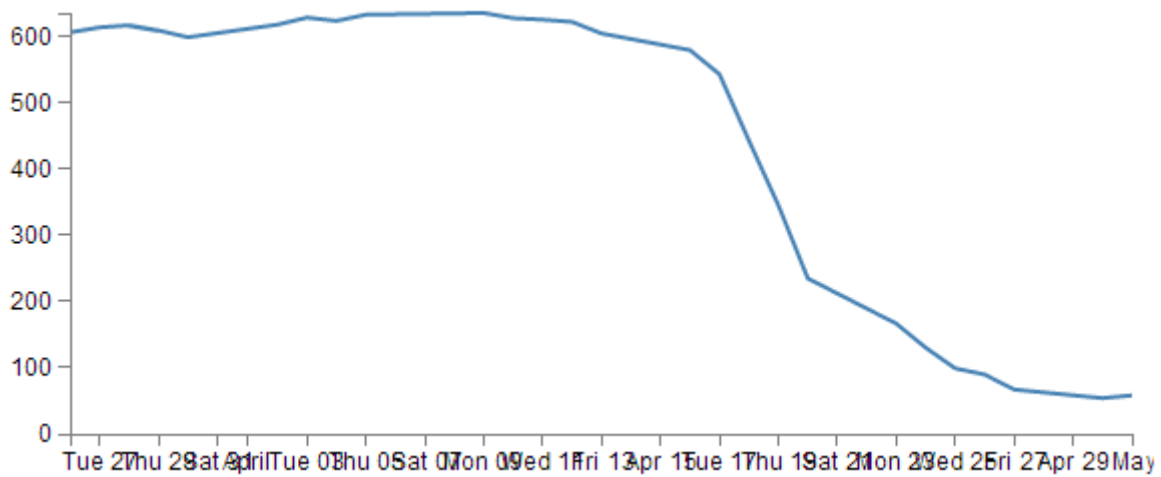
The following is the list¹⁴ of time intervals that D3 will consider when setting automatic ticks on a time based axis;

- 1-, 5-, 15- and 30-second.
- 1-, 5-, 15- and 30-minute.
- 1-, 3-, 6- and 12-hour.
- 1- and 2-day.
- 1-week.
- 1- and 3-month.
- 1-year.

Just for giggles have a think about what value of ticks you will need to increase to until you get D3 to show more than five ticks.

Hopefully you won't sneak a glance at the following graph before you come up with the right answer.

14 <https://github.com/mbostock/d3/wiki/Time-Scales>

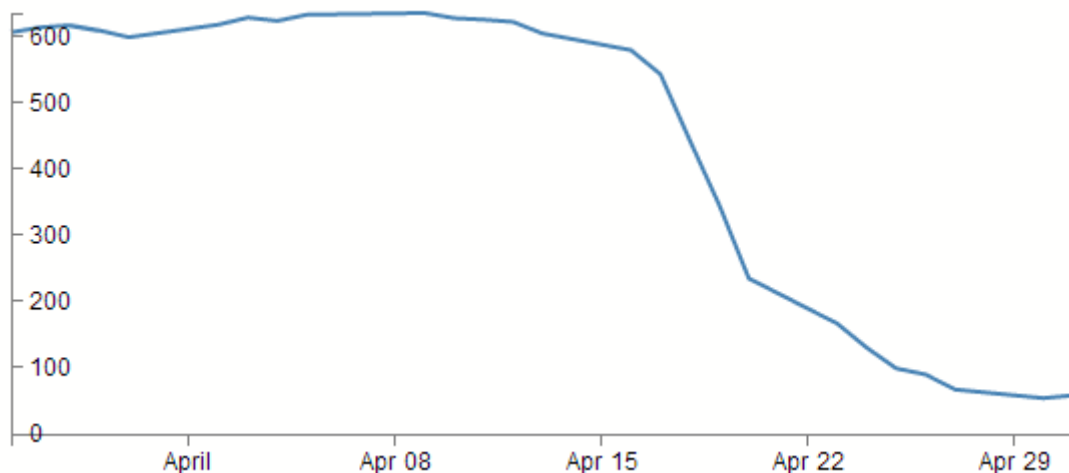


Yikes! The answer is 10! And then when it does, the number of ticks is so great that they jumble all over each other. Not looking to good there. However, you could rotate the text (or perhaps slant it) and it could fit in (that must be the topic of a future how-to). You could also make the graph longer if you wanted, but of course that is probably going to create other layout problems. So think about your data and presentation as a single entity.

The code that formats the y axis is pretty similar;

```
var yAxis = d3.svg.axis().scale(y)
    .orient("left").ticks(5);
```

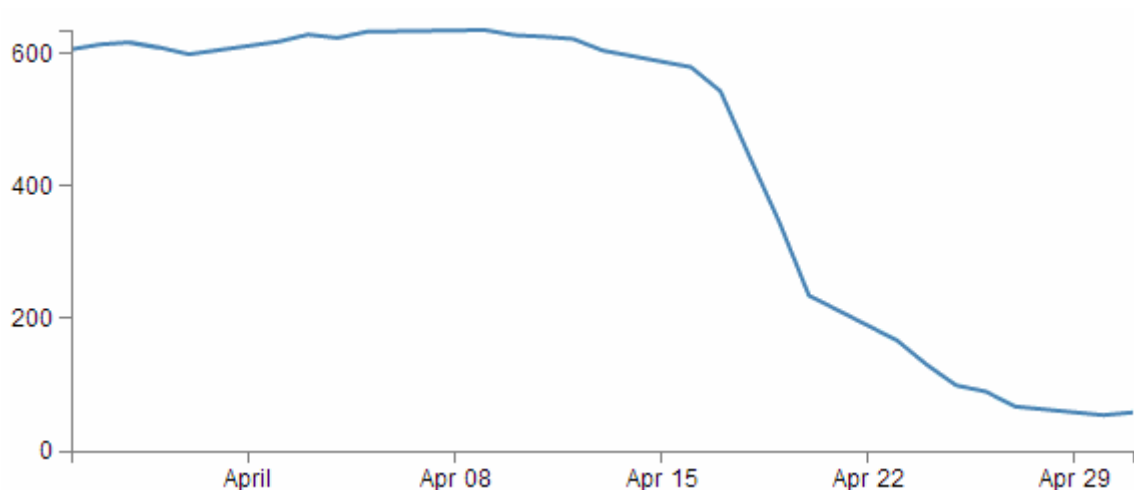
We can change the orientation to "right" if we want, but it won't be winning any beauty prizes.



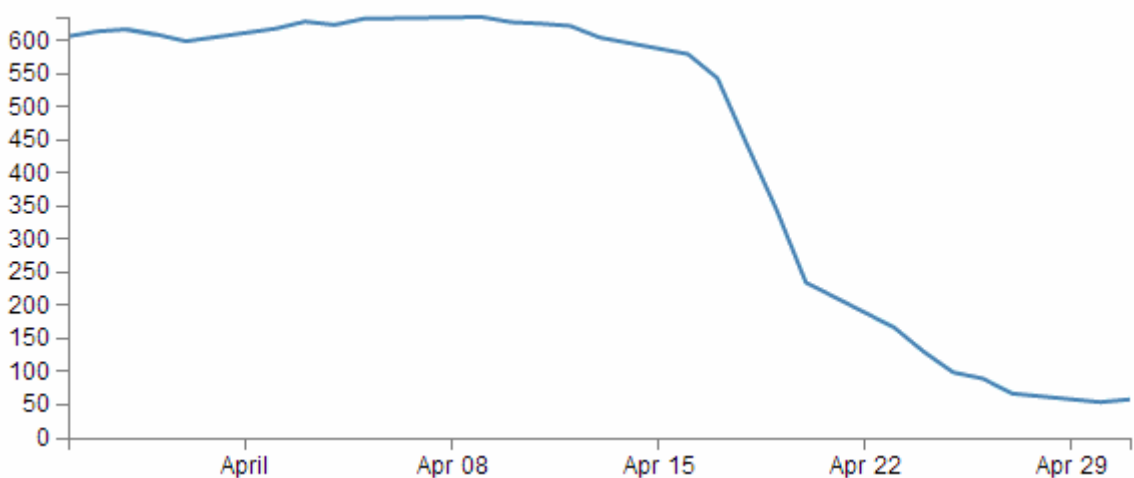
Nope. Not a pretty sight.

So what about the number of ticks? Well this scale is quite different to the x axis. Formatting the dates using logical separators (weeks, months) was tricky, but with standard numbers, it should be a little easier. In fact, there's a better than even chance that you've already had a look at the y axis and seen that there are 6 ticks there already when the script is asking for 5 :-)

And without too much fussing around, we can lower the tick number to 4 and we get a logical result.



But we need to raise the count to 10 before we get more than 6.



Adding data to the line function

We're getting towards the end of our journey through the script now. The next step is to get the information from the array 'data' and to place it in a new array that consists of a set of coordinated that correspond to the points we are going to plot.

```
var    valueline = d3.svg.line()
      .x(function(d) { return x(d.date); })
      .y(function(d) { return y(d.close); });
```

Now I'm aware that the statement above may be somewhat ambiguous. You would be justified in thinking that we already had the data stored and ready to go. But that's not *strictly* correct.

What we have is data in a raw format, we have added pieces of code that will allow the data to be adjusted for scale and range to fit in the area that we want to draw, but we haven't actually taken our raw data and adjusted it for our desired coordinates. That's what the code above does.

The main function that gets used here is the `d3.svg.line()` function¹⁵. This function uses assessor functions to store the appropriate information in the right area and in the case above they use the `x` and `y` assessors (that would be the bits that are `'x'` and `'y'`). The `d3.svg.line()` function is called a 'path generator' and this is an indication that it can carry out some pretty clever things on its own accord. But in essence its job is to assign a set of coordinates in a form that can be used to draw a line.

So each time this line function is called on, it will go through the data it is pointed to and it will assign coordinates to 'date' and 'close' pairs using the `'x'` and `'y'` functions that we set up earlier (which of course are responsible for scaling and setting the correct range / domain).

Of course, it doesn't get the data all by itself, we still need to actually call the `valueline` function with 'data' as the source to act on. But never fear, that's coming up soon.

Adding the SVG Canvas.

As the title states, the next piece of script forms and adds the canvas that D3 will then use to draw on.

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

So what exactly does that all mean?

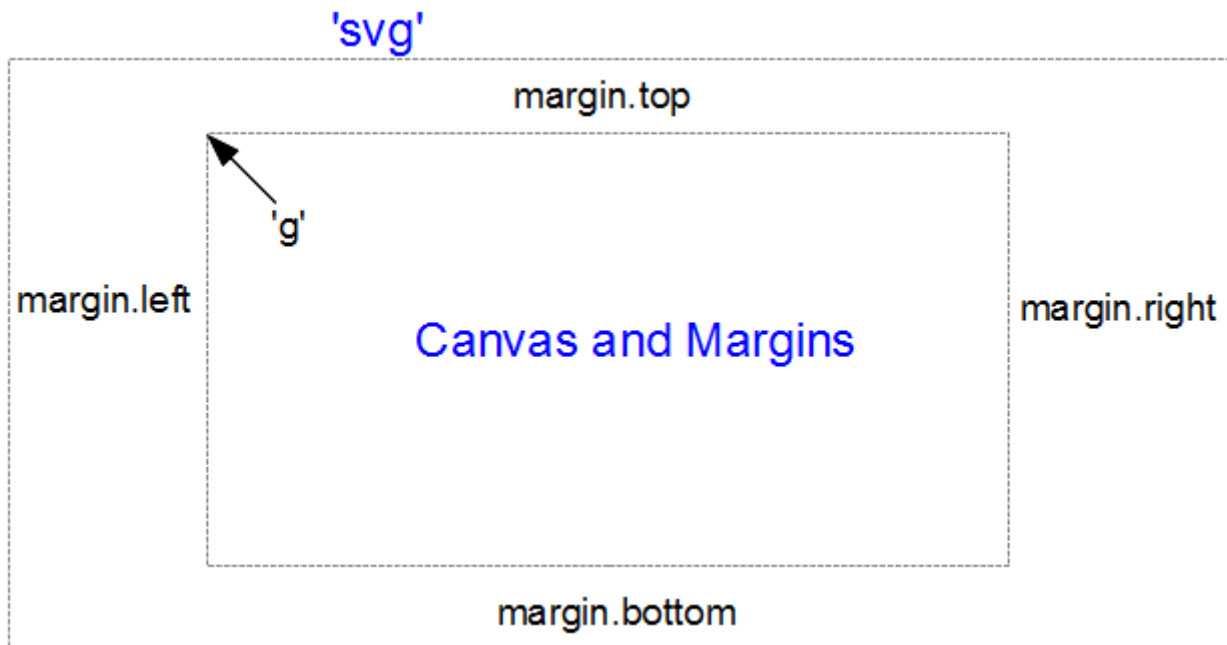
Well D3 needs to be able to have a space defined for it to draw things. And when you define the space it's going to use, you can also give the space you're going to use a name, attributes and positions within that space a designation.

In the example we're using here, we are 'appending' an SVG element (a canvas that we are going to draw things on) to the `<body>` element of the HTML page.

In human talk that means that on the web page and bounded by the `<body>` tag that we saw in the HTML part, we will have an area to draw on. That area will be 'width' plus the left and right margins wide and 'height' plus the top and bottom margins wide.

We also add an element 'g' that is referenced to the top left corner of the actual graph area on the canvas. 'g' is actually a grouping element in the sense that it is normally used for grouping together several related elements. So in this case those grouped elements will have a common reference.

15 <https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line>



(the image above is definitely not to scale, but I hope you get the general idea)

Interesting things to note about the code. The `.attr("stuff in here")` parts are attributes of the appended elements they are part of.

For instance;

```
.append("svg")
  .attr("width", width + margin.left + margin.right)
  .attr("height", height + margin.top + margin.bottom)
```

tells us that the 'svg' element has a "width" of `width + margin.left + margin.right` and the "height" of `height + margin.top + margin.bottom`.

Likewise...

```
.append("g")
  .attr("transform", "translate(" + margin.left + "," + margin.top + ")");
```

tells us that the element "g" has been transformed by moving(translating) to the point margin.left, margin.top. Or to the top left of the graph space proper. This way when we tell something to be drawn on our canvas, we can use the reference point "g" to make sure everything is in the right place.

Actually Drawing Something!

Up until now we have spent a lot of time defining, loading and setting up. Good news! We're about to finally draw something!

We jump lightly over some of the code that we have already explained and land on the part that draws the line.

```
svg.append("path")           // Add the valueline path.
  .attr("d", valueline(data));
```

This area occurs in the part of the code that has the data loaded and ready for action.

The `svg.append("path")` portion adds a new path element. A path element represents a shape that can be manipulated in lots of different ways (see more here:

<http://www.w3.org/TR/SVG/paths.html>). In this case it inherits the 'path' styles from the CSS section and on the following line (`.attr("d", valueline(data));`) we add the attribute "d".

This is an attributer that stands for 'path data' and sure enough the `valueline(data)` portion of the script passes the 'valueline' array (with its x and y coordinates) to the path element. This then creates a `svg` element which is a path going from one set of 'valueline' coordinates to another.

Then we get to draw in the axes;

```
svg.append("g")           // Add the X Axis
  .attr("class", "x axis")
  .attr("transform", "translate(0," + height + ")")
  .call(xAxis);

svg.append("g")           // Add the Y Axis
  .attr("class", "y axis")
  .call(yAxis);
```

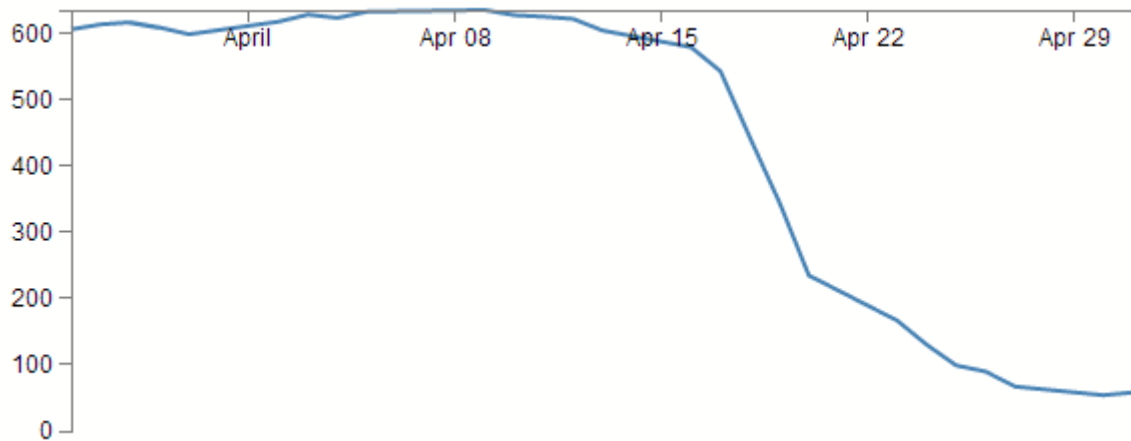
We have covered the formatting of the axis components earlier. So this part is actually just about getting those components drawn onto our canvas.

So both axes start by being appended to the "g" group. Then each has its own classes applied for styling via CSS. If you recall from earlier, they look a little like this;

```
.axis path,
.axis line {
  fill: none;
  stroke: grey;
  stroke-width: 1;
  shape-rendering: crispEdges;
}
```

Feel free to mess about with these to change the appearance of your axes.

On the x axis, we have a transform statement (`.attr("transform", "translate(0," + height + ")")`). If you recall, our point of origin for drawing is in the top left hand corner. Therefore if we want our x axis to be on the bottom of the graph, we need to move (transform) it to the bottom by a set amount. The set amount in this case is the height of the graph proper (height). So, for the point of demonstration we will remove the transform line and see what happens;



Yep, pretty much as anticipated.

The last part of the two sections of script (`.call(xAxis);` and `.call(yAxis);`) call the x and y axis functions and initiate the drawing action.

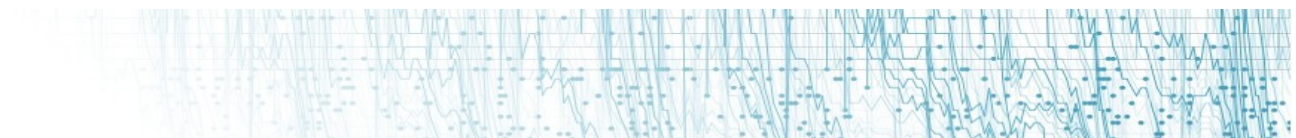
Wrap Up

Well that's it. In theory, you should now be a complete D3 ninja.

OK, perhaps a slight exaggeration. In fact there is a strong possibility that the information I have laid out here is at best borderline useful and at worst laden with evil practices and gross inaccuracies.

But look on the bright side. Irrespective of the nastiness of the way that any of it was accomplished or the inelegance of the code, if the picture drawn on the screen is pretty, you can walk away with a smile. :-)

I've said it before and I'll say it again. This is not a how-to for learning D3. This is how I have managed to muddle through in a bumbling way to try and achieve what I wanted to do. If some small part of it helps you. All good. Those with a smattering of knowledge of any of the topics I have butchered above (or below) are fully justified in feeling a large degree of righteous indignation. To those I say, please feel free to amend where practical and possible, but please bear in mind this was written from the point of view of someone with no experience in the topic and therefore try to keep any instructions at a level where a new entrant can step in.



Things you can do with the basic graph

The following headings in this section are intended to be a list of relatively simple 'block' type improvements that you can do to your graph to add functionality. The idea is to be able to use the simple graph that was used for the explanation of how D3 worked and just slot in code to add functionality (let's hope it works for you :-).

Adding Axis Labels

What's the first thing you get told at school when drawing a graph?

“Always label your axes!”

So, time to add a couple of labels!

First things first (because they're done slightly differently), the x axis. If we begin by describing what we want to achieve, it may make the process of implementing a solution a little more logical

What we want to do is to add a simple piece of text under the x axis and in the centre of the total span. Wow, that does sound easy.

And it is, but there are different ways of accomplishing it, and I think I should take an opportunity to demonstrate them. Especially since one of those ways is a BAD idea.

Lets start with the bad idea first :-).

This is the code we're going to add to the simple line graph script;

```
svg.append("text")           // text label for the x axis
  .attr("x", 265 )
  .attr("y", 240 )
  .style("text-anchor", "middle")
  .text("Date");
```

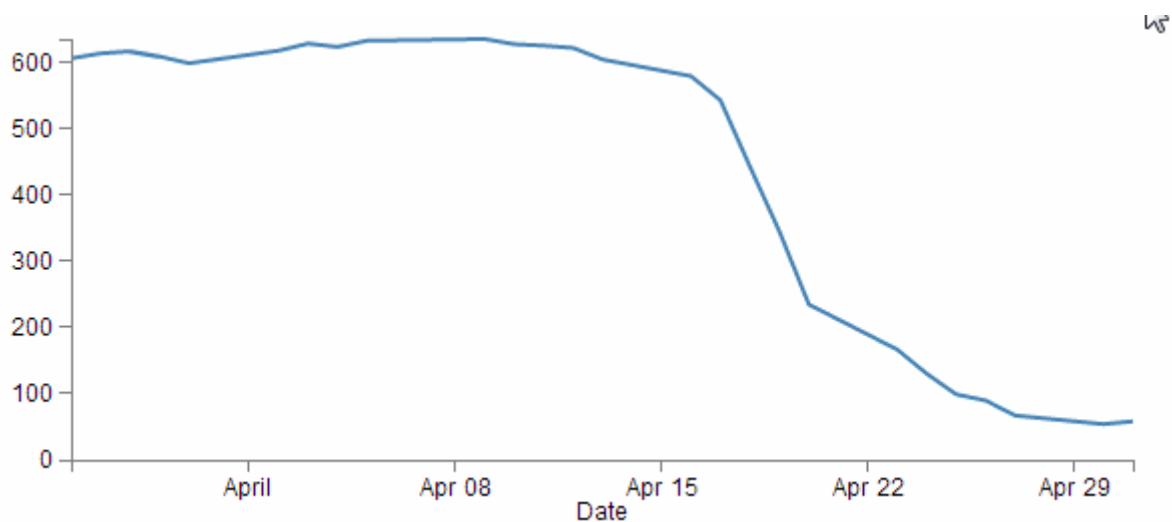
We will put it in between the blocks of script that add the x axis and the y axis.

```
svg.append("g")              // Add the X Axis
  .attr("class", "x axis")
  .attr("transform", "translate(0," + height + ")")
  .call(xAxis);

//      PUT THE NEW CODE HERE!

svg.append("g")              // Add the Y Axis
  .attr("class", "y axis")
  .call(yAxis);
```

Before we describe what's happening, let's take a look at the result;



Well, it certainly did what it was asked to do. There's a 'Date' label as advertised! (Yes, I know it's not pretty.) Let's describe the code and then work out why there's a better way to do it.

```
svg.append("text")           // text label for the x axis
    .attr("x", 265 )
    .attr("y", 240 )
    .style("text-anchor", "middle")
    .text("Date");
```

The first line appends a "text" element to our canvas. There is a lot more to learn about "text" elements here; <http://www.w3.org/TR/SVG/text.html#TextElement>.

The next two lines (`.attr("x", 265)` and `.attr("y", 240)`) set the attributes for the x and y coordinates to position the text on the canvas.

The second last line (`.style("text-anchor", "middle")`) ensures that the text 'style' is such that the text is centre aligned and therefore remains nicely centred on the x,y coordinates that we send it to.

The final line (`.text("Date");`) adds the actual text that we are going to place.

That seems really simple and effective and it is. However, the bad part about it is that we have hard coded the location for the date into the code. This means if we change any of the physical aspects of the graph, we will end up having to re-calculate and edit our code. And we don't want to do that.

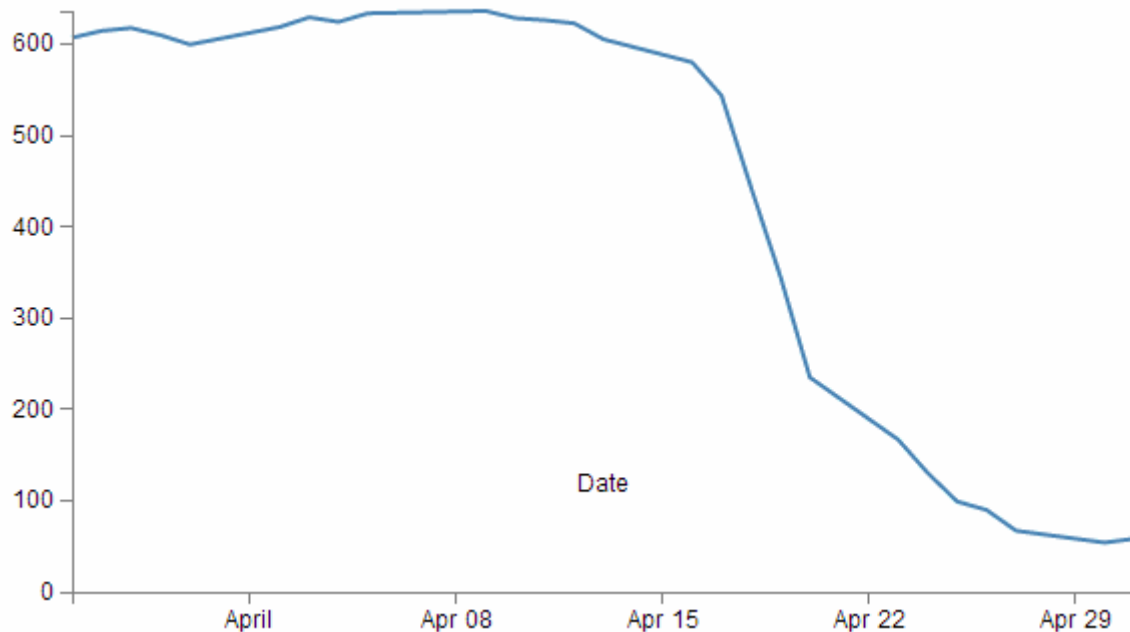
Here's an example. If I decide that I would prefer to increase the height of the graph by editing the line here;

```
height = 270 - margin.top - margin.bottom;
```

and making the height 350 pixels;

```
height = 350 - margin.top - margin.bottom;
```

The result is as follows;



EVERYTHING about the graph has adjusted itself, except our nasty, hard coded 'Date' label. This is far from ideal and can be easily fixed by using the variables that we set up ever so carefully earlier.

So, instead of;

```
.attr("x", 265 )
.attr("y", 240 )
```

lets let our variables do the walking and use;

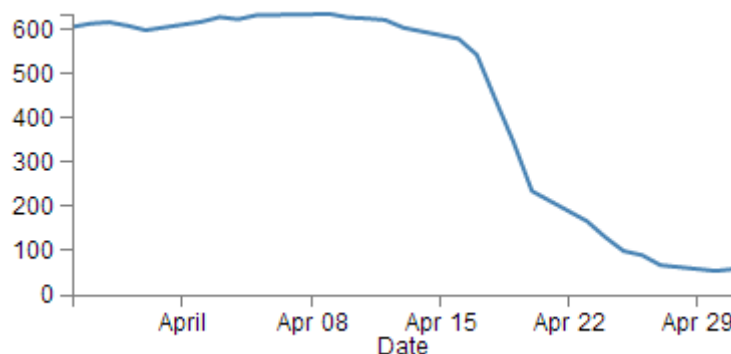
```
.attr("x", width / 2 )
.attr("y", height + margin.bottom)
```

So with this code we tell the script that the 'Date' label will always be halfway across the width of the graph (no matter how wide it is) and at the bottom of the graph with respect to it's height and the bottom margin (remember it uses a coordinates system that increases from the top down).

The end result of using variables is that if I go to an extreme of changing the height and width of my graph to;

```
width = 400 - margin.left - margin.right,
height = 200 - margin.top - margin.bottom;
```

We still get an acceptable result;



Well, for the label position at least :-).

So the changes to using variables is just a useful lesson that variables rock and mean that you don't have to worry about your graph staying in relative shape while you change the dimensions. The astute readers amongst you will have learned this lesson very early on in your programming careers, but it's never a bad idea to make sure that users that are unfamiliar with the concept have an indicator of why it's a good idea.

Now the third method that I mentioned at the start of our x axis odyssey. This is not mentioned because its any better or worse way to implement your script (The reason that I say this is because I'm not *sure* if it's better or worse.) but because it's sufficiently different to make it look confusing if you didn't think of it in the first place.

So, we'll take our marvellous coordinates code;

```
.attr("x", width / 2 )  
.attr("y", height + margin.bottom)
```

And replace it with a single (longer) line;

```
.attr("transform", "translate(" + (width / 2) + " , " + (height + margin.bottom) + ")")
```

This uses the "transform" attribute to move (translate) the point to place the 'Date' label to exactly the same spot that we've been using for the other two examples (using variables of course).

Things to note about this piece of code;

The "translate" function is done in a 'translate(x,y)' style but it is put on the page in such a way that the verbatim pieces that get passed back are in speech marks and the variables are in the clear (in a manner of speaking). That's why the comma is in speech marks.

Additionally, the variables are contained within plus signs. I make the assumption that this is a designator for 'areas where there is variable action going on'. The end result is that if you try to do some maths in that area with a plus sign, it does not appear to work (or at least it didn't for me). That's why I put the variable for (+ (height + margin.bottom) +) in parenthesis (then I thought I should make the + (width / 2) + part look the same, but actually you can get away without them there).

So, that's the x axis label. Time to do the y axis. The code we're going to use looks like this;

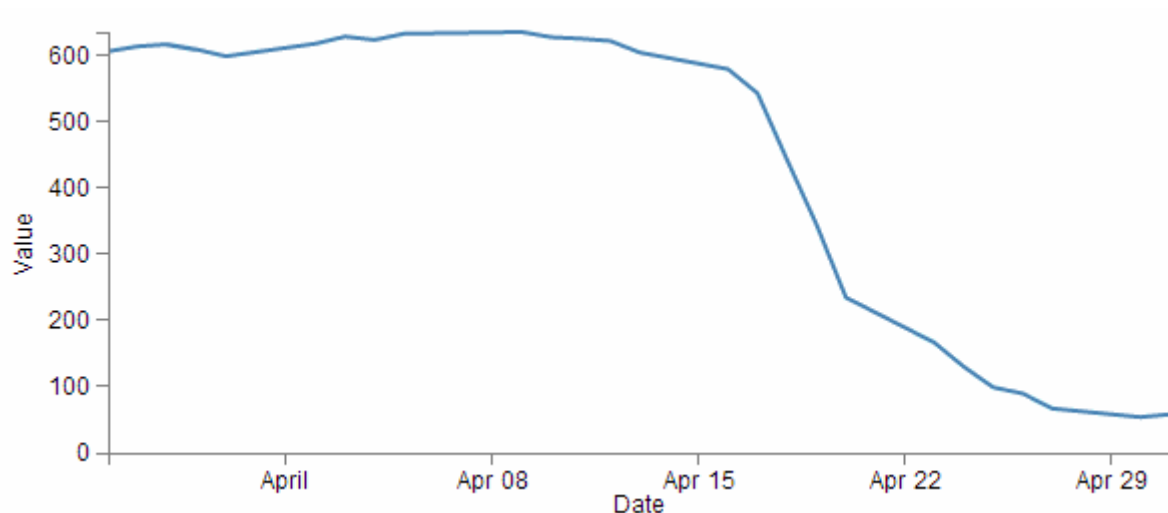
```
svg.append("text")  
  .attr("transform", "rotate(-90)")  
  .attr("y", 0 - margin.left)  
  .attr("x", 0 - (height / 2))  
  .attr("dy", "1em")  
  .style("text-anchor", "middle")  
  .text("Value");
```

For the sake of neatness we will put the piece of code in a nice logical spot and this would be following the block of code that added the y axis (but before the closing curly bracket)

```
svg.append("g") // Add the Y Axis  
  .attr("class", "y axis")  
  .call(yAxis);  
  
// PUT THE NEW CODE HERE!
```

```
});
```

And the result looks like this;



There we go, a label for the y axis that is nicely centred and (gasp!) rotated by 90 degrees! Woah, does the leetness never end! (No. No it does not.)

So, how do we get to this incredible result?

The first thing we do is the same as for the x axis and append a test element to our canvas (`svg.append("text")`).

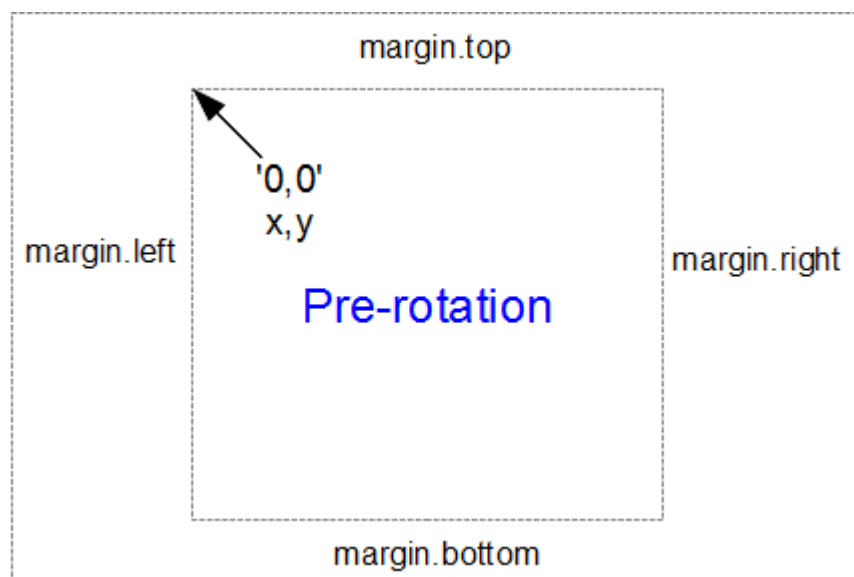
Then things get interesting.

```
.attr("transform", "rotate(-90)")
```

Because that line rotates everything by -90 degrees. While it's obvious that the text label 'Value' has been rotated by -90 degrees (from the picture), the following lines of code show that we also rotated our reference point (which can be a little confusing).

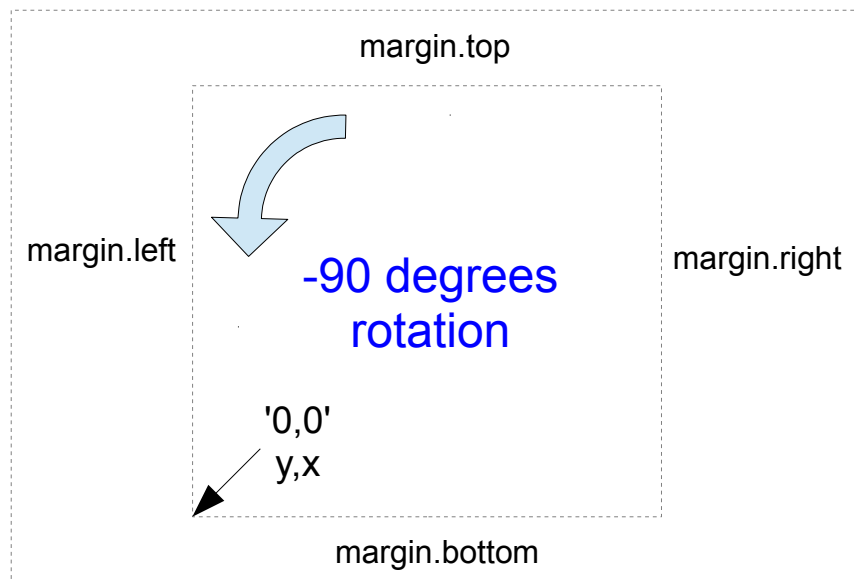
```
.attr("y", 0 - margin.left)
.attr("x", 0 - (height / 2))
```

Let's get graphical to illustrate how this works;



Here's our starting position, with `x,y` in the 0,0 coordinate of the graph drawing area surrounded by the margins.

When we apply a -90 degrees transform we get the equivalent of this;



Here the 0,0 coordinate has been shifted by -90 degrees and the x,y designations are flipped so that we now need to tell the script that we're moving a 'y' coordinate when we would have otherwise been moving 'x'.

Hence, when the script runs...

```
.attr("y", 0 - margin.left)
```

... we can see that this is moving the x position to the left from the new 0 coordinate by the margin.left value.

Likewise when the script runs...

```
.attr("x", 0 - (height / 2))
```

... this is actually moving the y position from the new 0 coordinate halfway up the height of the graph area.

Now, I will be the first to admit that this does seem a little confusing, but here's the good part. You really don't need to understand it completely. Simply do what I did when I saw the code. Play with it a bit till you get the result you were looking for. If that means putting in some hard coded numbers and incrementing them to see which way is the new 'up'. Good! Once you work it out, then work out how to get the right variable expression in there and you're set.

In the worst case scenario, simply use the code blocks as shown here and leave well enough alone :-)

Right, we're not quite done yet. The following line has the effect of shifting the text slightly to the right.

```
.attr("dy", "1em")
```

Firstly the reason we do this is that our previous translation of coordinates means that when we place our text label it sits exactly on the line of 0 - margin.left. But in this case that takes the text to the other side of the line, so it actually sits just outside the boundary of the overall canvas.

The "dy" attribute is another coordinate adjustment move, but this time a relative adjustment and the "1em" is a unit of measure that equals exactly one unit of the currently specified text point size¹⁶. So what ends up happening is that the 'Value' label gets shifted to the right by exactly the height of the text, which neatly places it exactly on the edge of the canvas.

The two final lines of this part of the script are the same as for the x axis and they make sure reference point is aligned to the centre of the text (`.style("text-anchor", "middle")`) and then it prints the text (`.text("Value");`). There, that wasn't too painful.

How to add a title to your graph

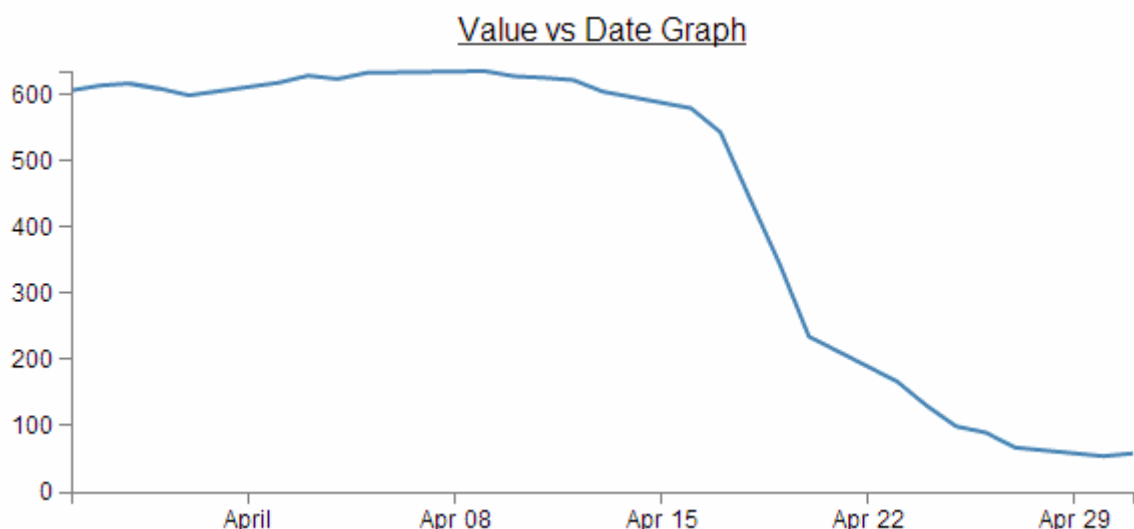
If you've read through the adding the axis labels section most of this will come as no surprise.

What we want to do to add a title to the graph is to add a text element (just a few words) that will appear above the graph and centred left to right.

The code block we will use will look like this;

```
svg.append("text")
  .attr("x", (width / 2))
  .attr("y", 0 - (margin.top / 2))
  .attr("text-anchor", "middle")
  .style("font-size", "16px")
  .style("text-decoration", "underline")
  .text("Value vs Date Graph");
```

And the end result will look like this;



A nice logical place to put the block of code would be towards the end of the JavaScript. In fact I would put it as the last element we add. So here;

```
svg.append("g") // Add the Y Axis
  .attr("class", "y axis")
  .call(yAxis);

// PUT THE NEW CODE HERE!

});
```

¹⁶ [http://en.wikipedia.org/wiki/Em_\(typography\)](http://en.wikipedia.org/wiki/Em_(typography))

Now since the vast majority of the code for this block is a regurgitation of the axis labels code, I don't want to revisit that and bloat up this document even more, so I will direct you back to that section if you need to refresh yourself on any particular line. But..... There are a couple of new ones in there which could benefit from a little explanation.

Both of them are style descriptors and as such their job is to apply a very specific style to this element.

```
.style("font-size", "16px")  
.style("text-decoration", "underline")
```

What they do is pretty self explanatory. Make the text a specific size and underline it. But what is perhaps slightly more interesting is that we have this declaration in the JavaScript code and not in the CSS portion of the file.

Because strictly speaking, this is the sort of thing that would be placed in the `<style>` section of the HTML code, but in this case since it is only going to be used the once, we shouldn't feel too bad putting it here.

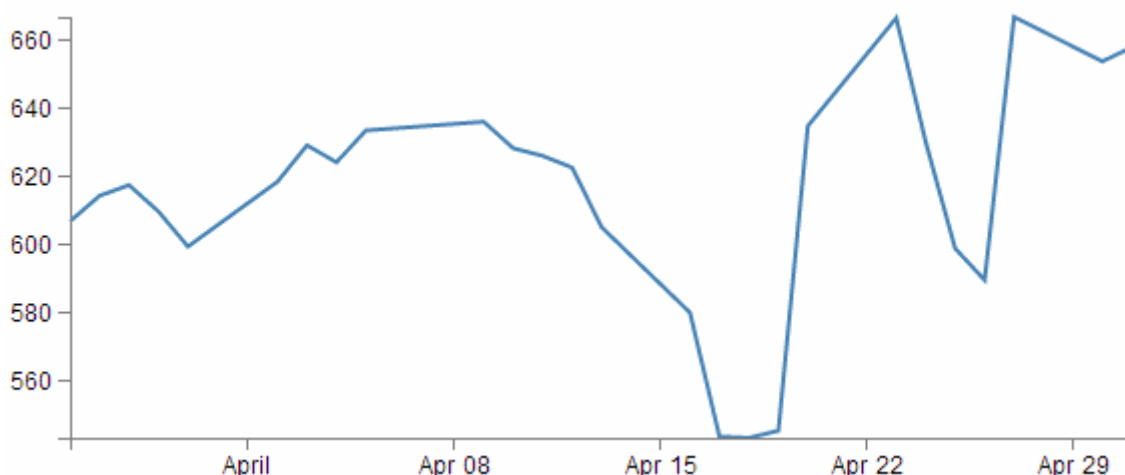
Smoothing out graph lines

When you draw a line graph, what you're doing is taking two (or more) sets of coordinates and connecting them with a line (or lines). I know that sounds simplistic, but bear with me. When you connect these points, you're telling the viewer of the graph that in between the individual points, you expect the value to vary in keeping with the points that the line passes through. So in a way, you're trying to interpret the change in values that are not shown.

Now this is not strictly true for all graph types, but it does hold for a lot of line graphs.

So... when connecting these known coordinated together, you want to make the best estimate of how the values would be represented. In this respect, sometimes a straight line between points is not the best representation.

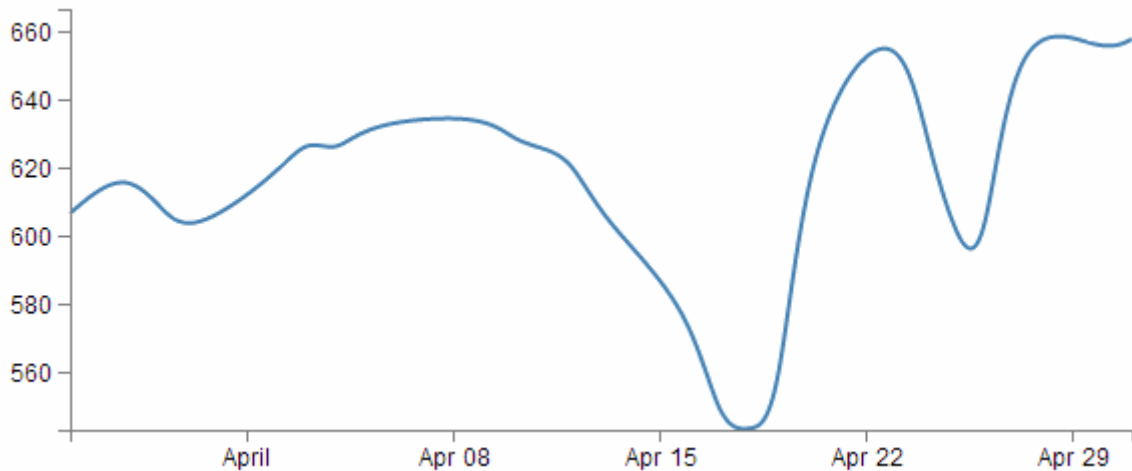
For instance. Earlier, when demonstrating the extent function for graphing we showed a graph of the varying values with the y axis showing a narrow range.



The resulting variation of the graph shows a fair amount of extremes and you could be forgiven for thinking that if this represented a smoothly flowing analog system of some kind then some of those sharp peaks and troughs would not be a true representation of how the system or figures varied.

So how should it look? Ahh... The \$64,000 question. I don't know :-). You will have a better idea since you are the person who will know your data best. However, what I do know is that D3 has some tricks up its sleeve to help.

We can easily change what we see above into;



How about that? And the massive amount of code required to carry out what must be a ridiculously difficult set of calculations?

```
.interpolate("basis")
```

Now, that is slightly unfair because that's the code that YOU need to put in your script, but Mike Bostock probably had to do the mental equivalent of walking across hot coals to get it to work so nicely.

So where does this neat piece of code go? Here;

```
var valueline = d3.svg.line()
  .interpolate("basis")           // <=== THERE IT IS!
  .x(function(d) { return x(d.date); })
  .y(function(d) { return y(d.close); });
```

So is that it? Noooooo..... There's more! This is one form of interpolation effect that can be applied to your data, but there is a range and depending on your data you can select the one that is appropriate.

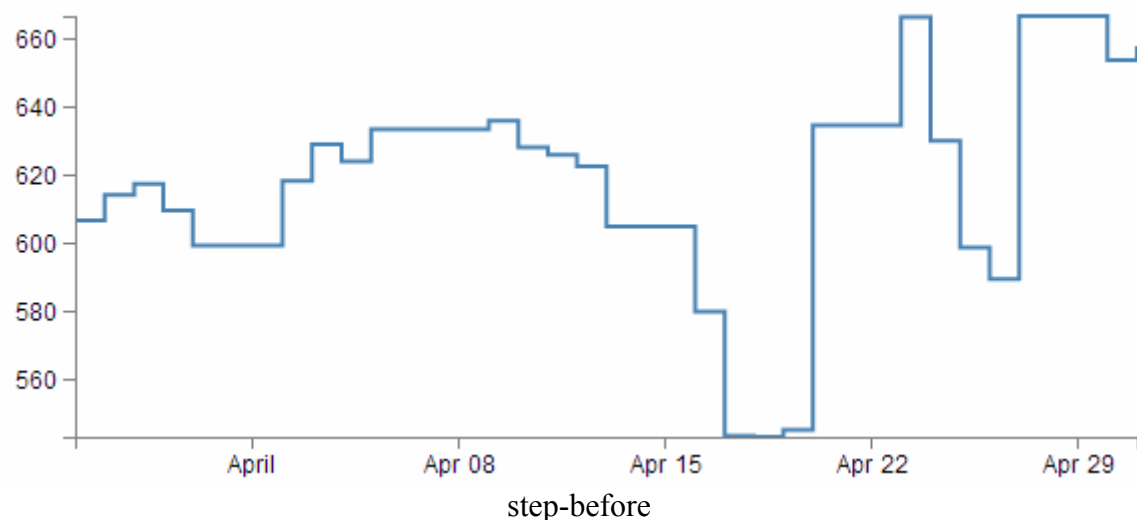
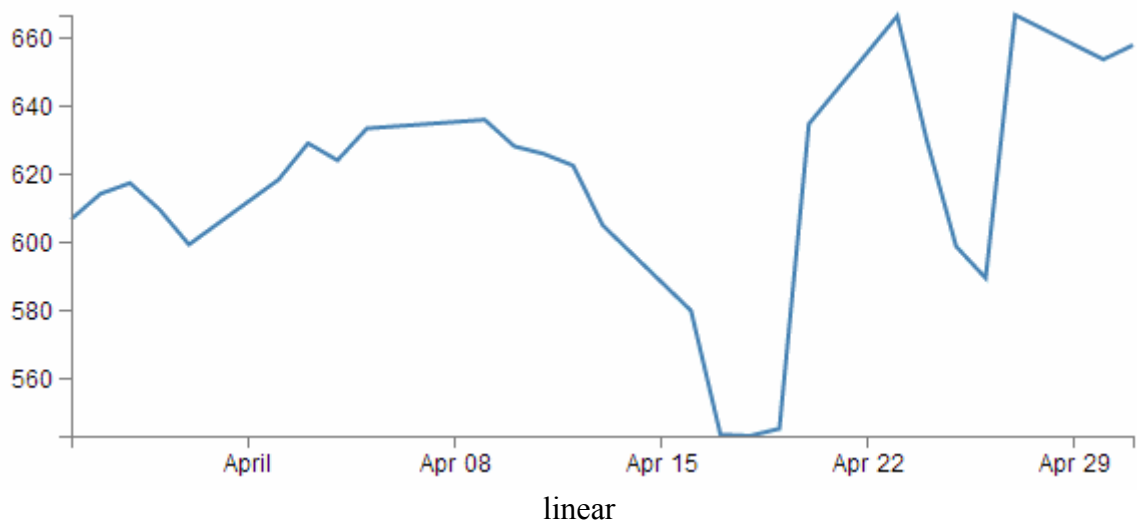
Here's the list of available options and for more about them head on over to the D3 wiki¹⁷ and look for 'line.interpolate'.

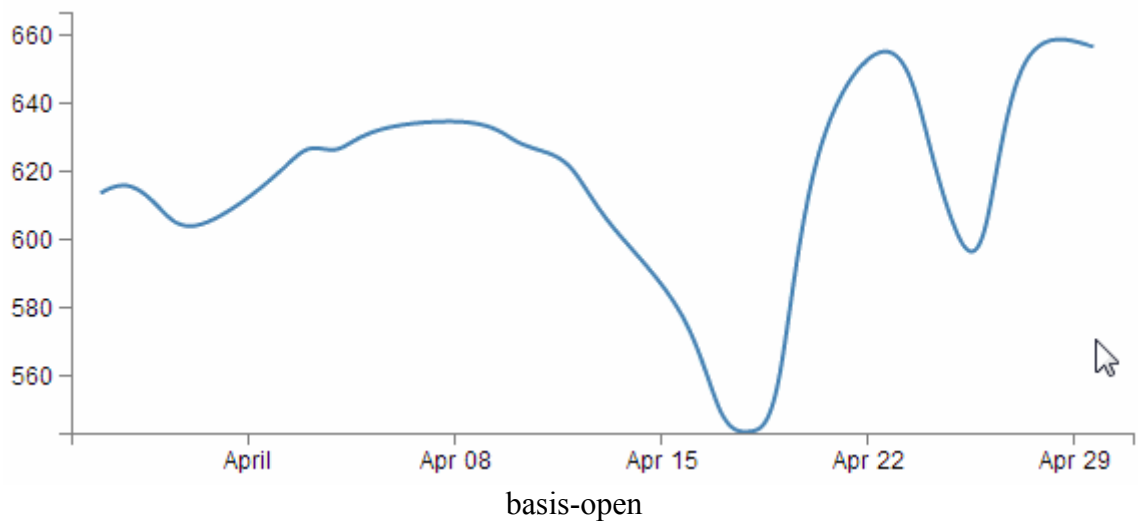
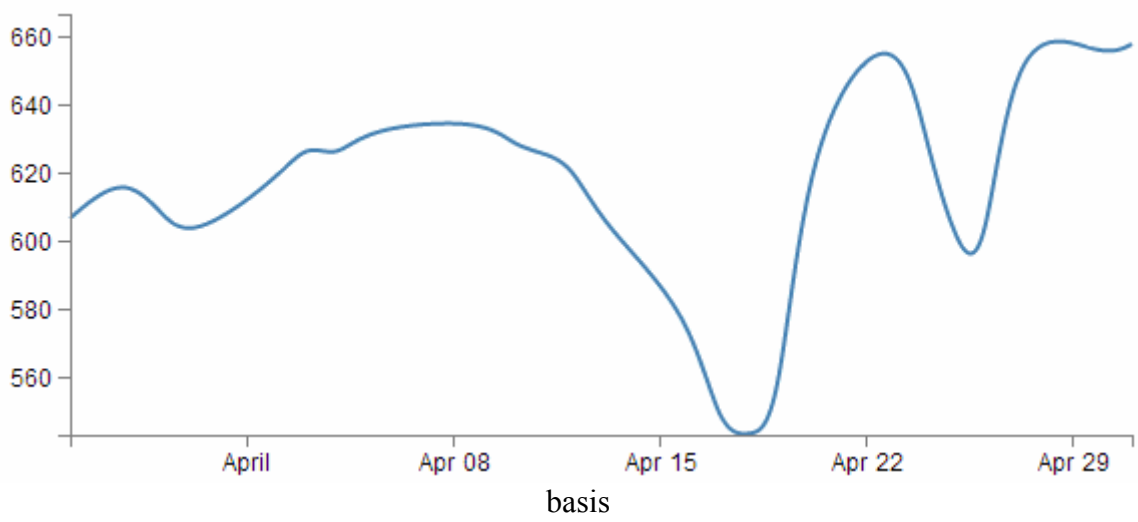
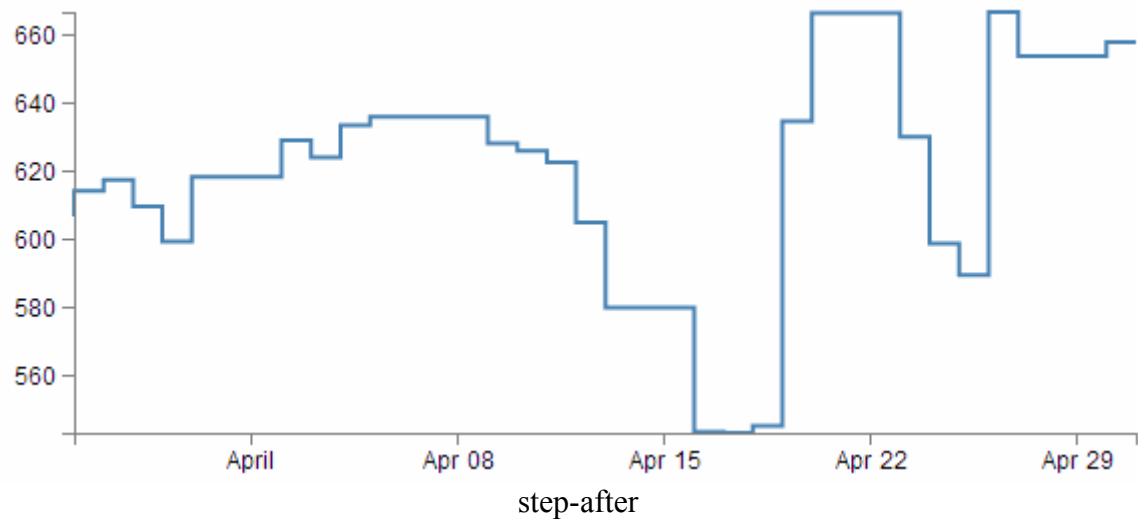
- linear – Normal line (jagged).
- step-before – a stepping graph alternating between vertical and horizontal segments.
- step-after – a stepping graph alternating between horizontal and vertical segments.
- basis – a B-spline, with control point duplication on the ends (that's the one above).
- basis-open – an open B-spline; may not intersect the start or end.
- basis-closed – a closed B-spline, with the start and the end closed in a loop.

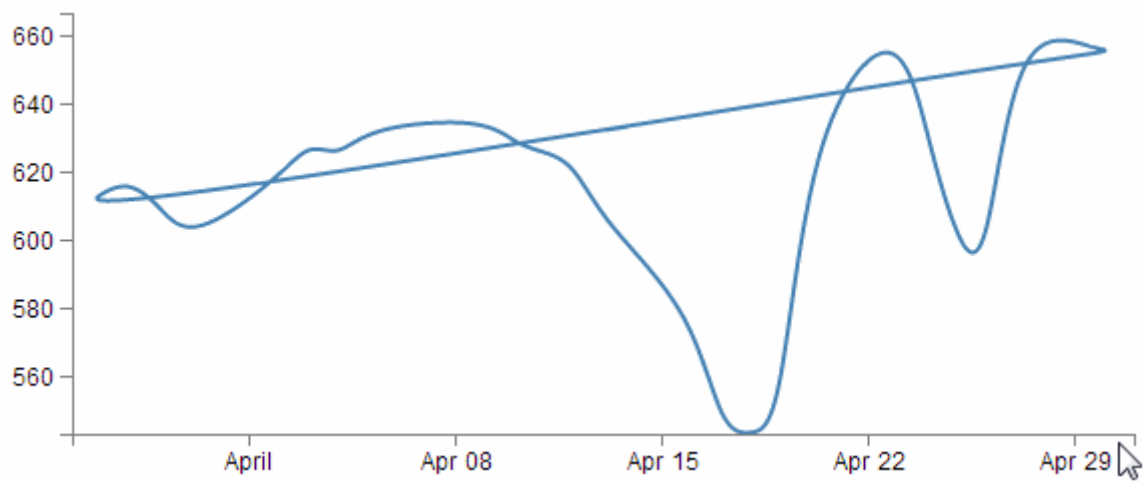
¹⁷ https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-line_interpolate

- bundle - equivalent to basis, except a separate tension parameter is used to straighten the spline. This could be really cool with varying tension.
- cardinal - a Cardinal spline, with control point duplication on the ends. It looks slightly more 'jagged' than basis.
- cardinal-open - an open Cardinal spline; may not intersect the start or end, but will intersect other control points. So kind of shorter than 'cardinal'.
- cardinal-closed - a closed Cardinal spline, looped back on itself.
- monotone - cubic interpolation that makes the graph only slightly smoother.

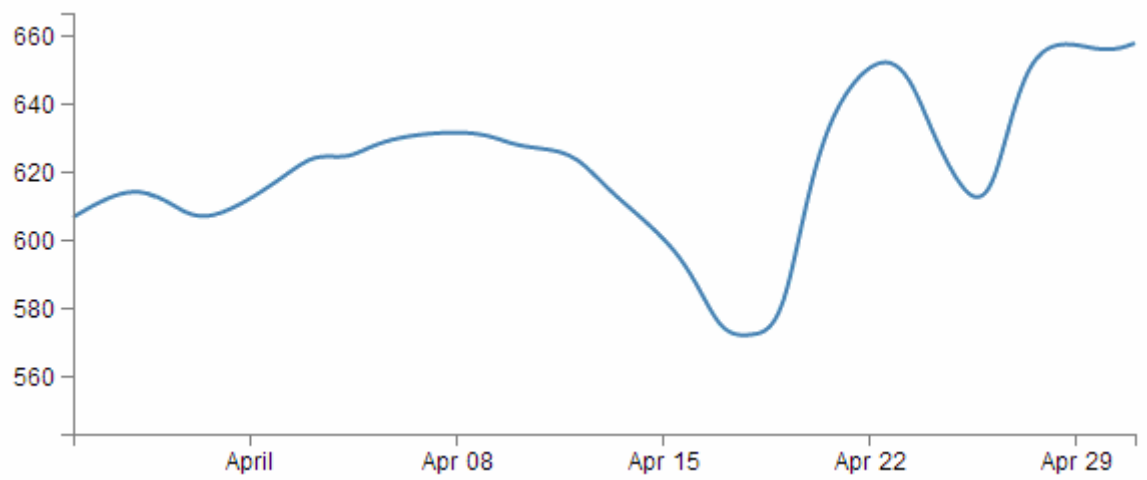
Because in the course of writing this I took an opportunity to play with each of them, I was pleasantly surprised to see some of the effects and it seems like a shame to deprive the reader of the same joy :-). So at the risk of deforesting the planet (so I hope you are reading this in electronic format) here is each of the above interpolation types applied to the same data.



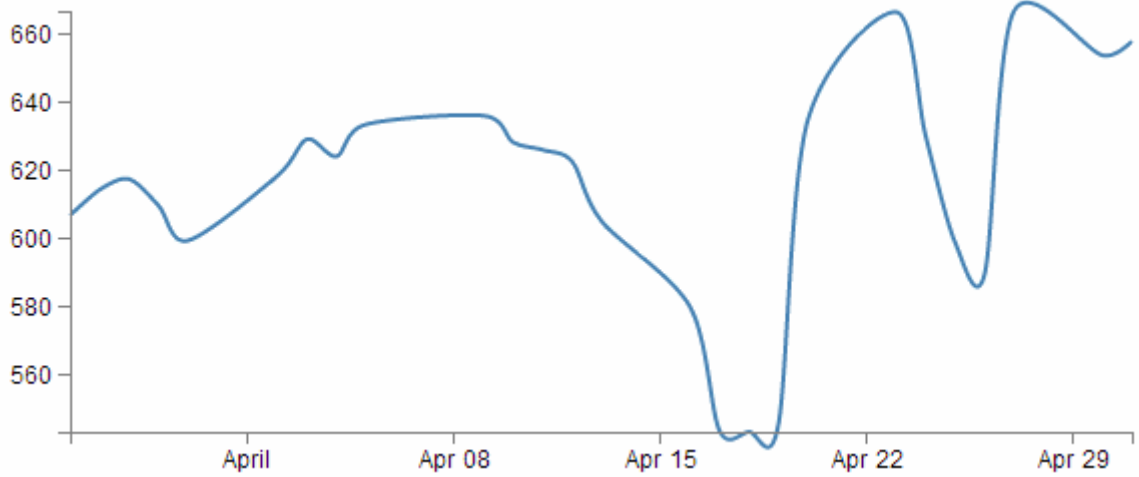




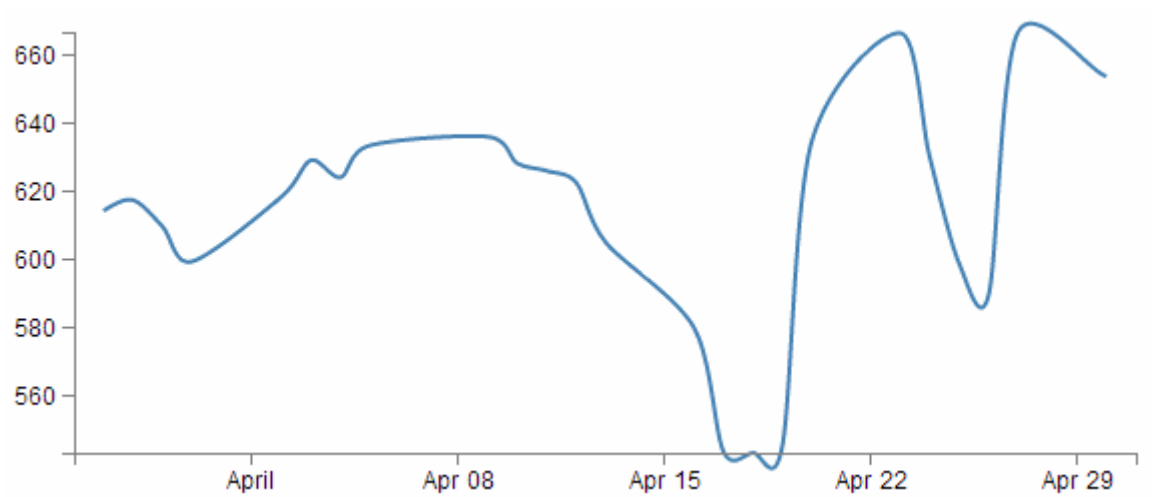
basis-closed



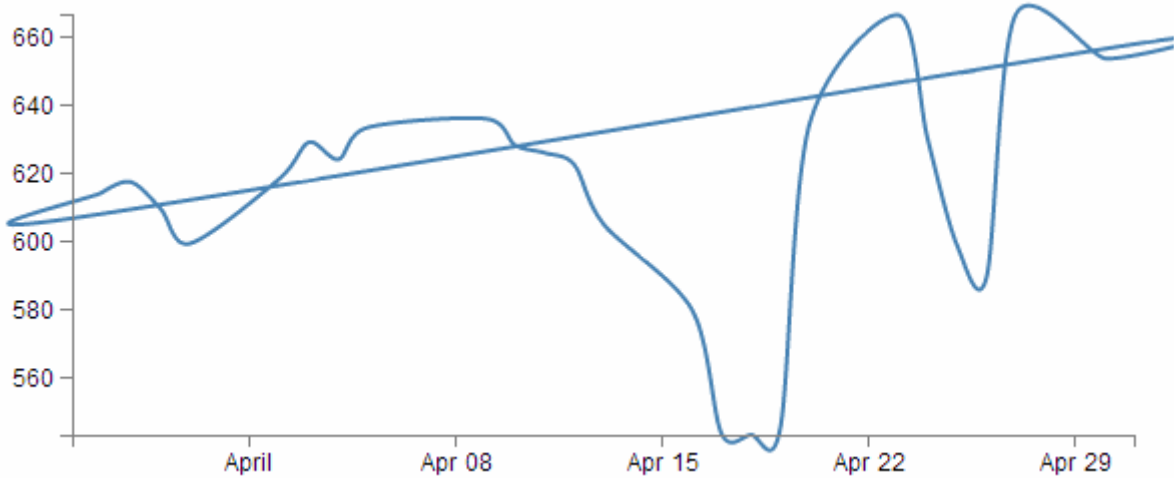
bundle



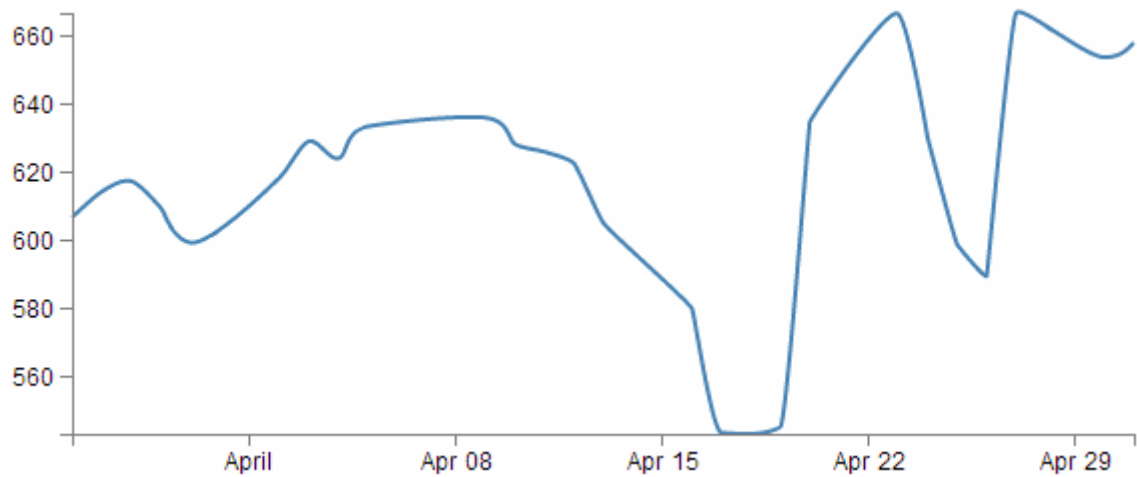
cardinal



cardinal-open



cardinal-closed



monotone

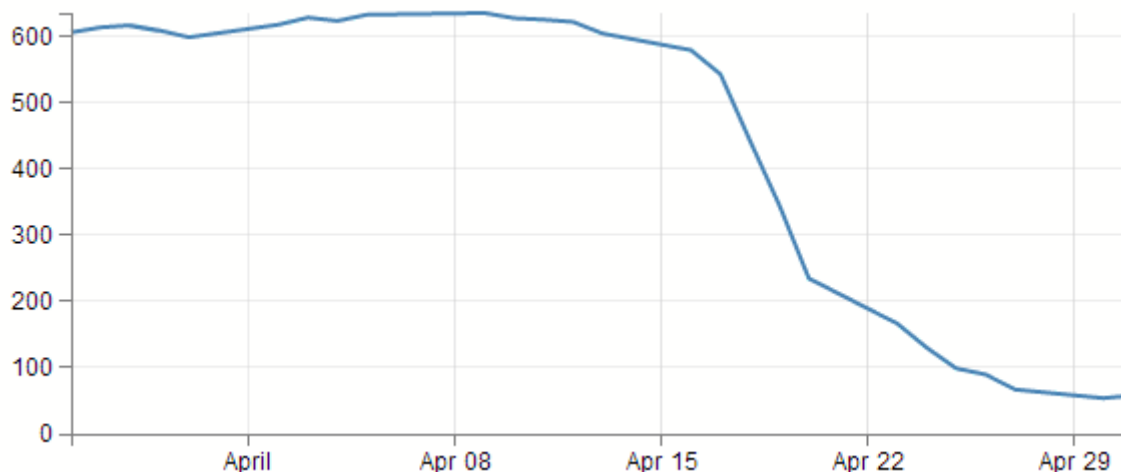
So, over to you to decide which format of interpolation is going to suit your data best:-).

Adding grid lines to a graph

Grid lines are an important feature for some graphs as they allow the eye to associate three analogue scales (the x and y axis and the displayed line).

There is currently a tendency to use graphs without grid lines online as it gives the appearance of a 'cleaner' interface, but they are still widely used and a necessary component for graphing.

This is what we're going to draw;



Like pretty much everything in this document, the clever parts of this are not my work. I've simply used other peoples cleverness to solve my problems. In this case I think the source of this solution came from the good work of Justin Palmer in his excellent description of the design of a line graph here <http://dealloc.me/2011/06/24/d3-is-not-a-graphing-library.html>. However, in retrospect when I've looked back, I'm not sure if I got this right (as I did this quite a while ago when I was less fastidious about noting my sources). In any case, Justin's work is excellent and I heartily recommend it, and here is my implementation of what I think is his work :-)

So, how to build grid lines?

Well what we're going to do is to use the axis function to generate two more axis elements (one for x and one for y) but for these ones instead of drawing the main lines and the labels, we're just going to draw the tick lines. Really long ticklines.

To create then we have to add in 3 separate blocks of code.

1. One in the CSS section to define what style the grid lines will have.
2. One to define the functions that generate the grid lines. And...
3. One to draw the lines.

The grid line CSS

This is the total styling that we need to add for the tick lines;

```
.grid .tick {  
  stroke: lightgrey;  
  opacity: 0.7;  
}  
.grid path {  
  stroke-width: 0;  
}
```

Just add this block of code at the end of the current CSS that is in the simple graph template (just before the `</style>` tag).

The CSS here is done in two parts.

The first portion sets the line colour (stroke) and the opacity (transparency) of the lines.

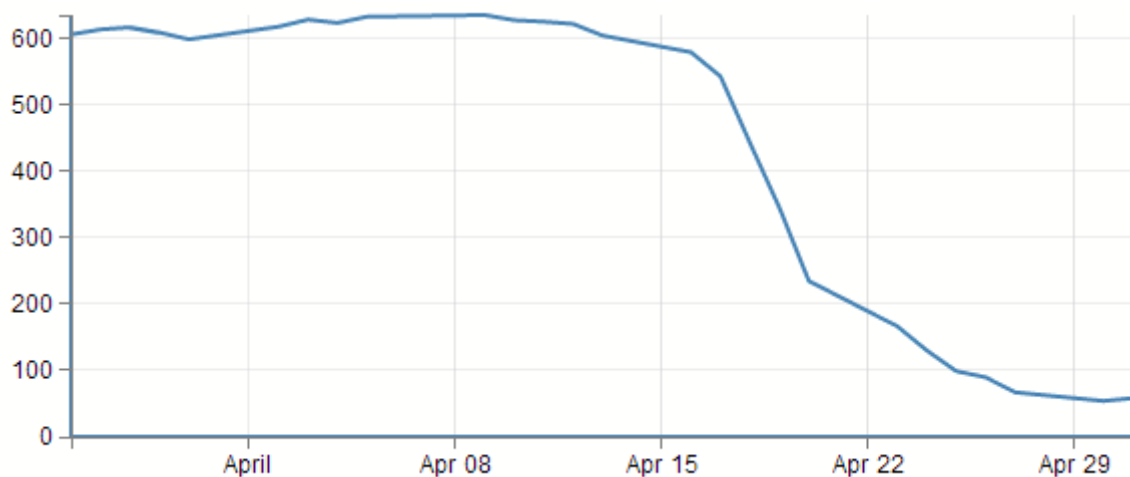
```
stroke: lightgrey;  
opacity: 0.7;
```

The colour is pretty standard, but in using the opacity style we give ourselves the opportunity to use a good shade of colour (if grey actually *is* a colour) and to juggle the degree to which it stands out a little better.

The second part is the stroke width.

```
stroke-width: 0;
```

Now it might seem a little weird to be setting the stroke width to zero, but if you don't (and we remove the style) this is what happens;



If you look closely (compare with the previous picture if necessary) the main lines for the axis have turned thicker. The stroke width style is obviously adding in new (thicker) axis lines and we're not interested in them at the moment. Therefore, if we set the stroke width to zero, we get rid of the problem.

Define the grid line functions

We will need to define two functions to generate the grid lines and they look a little like this;

```
function make_x_axis() {
    return d3.svg.axis()
        .scale(x)
        .orient("bottom")
        .ticks(5)
}

function make_y_axis() {
    return d3.svg.axis()
        .scale(y)
        .orient("left")
        .ticks(5)
}
```

Each function will carry out it's configuration when called from the later part of the script (the drawing part).

A good spot to place the code is just before we load the data with the d3.tsv

```
//      <== Put the functions here!

// Get the data
d3.tsv("data/data.tsv", function(error, data) {
    data.forEach(function(d) {
        d.date = parseDate(d.date);
        d.close = +d.close;
    });
```

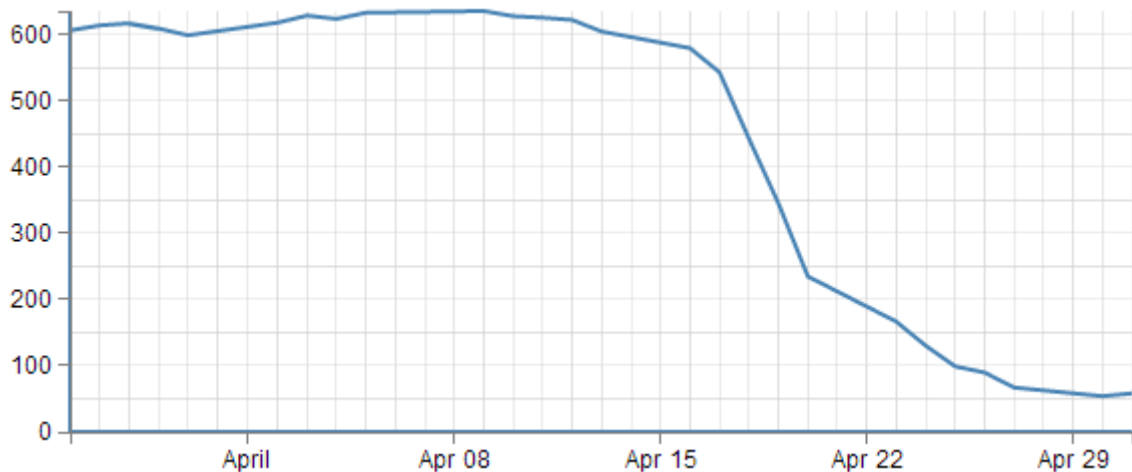
Both functions are almost identical. They give the function a name (make_x_axis and make_y_axis) which will be used later when the piece of code that draws the lines calls out to them.

Both functions also show which parameters each will be fed back to the drawing process when called. Both make sure that it uses the d3.svg.axis function and then they set individual attributes which make sense.

They make sure they've got the right axis (.scale(x) and .scale(y)).

They set the orientation of the axes to match the incumbent axes (.orient("bottom") and .orient("left")).

And they set the number of ticks to match the number of ticks in the main axis (.ticks(5) and .ticks(5)). You have the opportunity here to do something slightly different if you want. For instance, if we think back to when we were setting up the axis for the basic graph and we messed about with seeing how many we could get to appear. If we increase the number of ticks that appear in the grid (lets say to .ticks(30) and .ticks(10))) we get the following;



So the grid lines can now show divisions of 50 on the y axis and per day on the x axis :-)

Draw the lines

The final block of code we need is the bit that draws the lines.

```
svg.append("g")
  .attr("class", "grid")
  .attr("transform", "translate(0," + height + ")")
  .call(make_x_axis()
    .tickSize(-height, 0, 0)
    .tickFormat("")
  )

svg.append("g")
  .attr("class", "grid")
  .call(make_y_axis()
    .tickSize(-width, 0, 0)
    .tickFormat("")
  )
```

The first two lines of both the x and y axis grid lines code above should be pretty familiar by now. The first one appends the element to be drawn to the group "g". the second line (`.attr("class", "grid")`) makes sure that the style information set out in the CSS is applied.

The x axis grid lines portion makes a slight deviation from conformity here to adjust its positioning to take into account the coordinates system `.attr("transform", "translate(0," + height + ")")`.

Then both portions call their respective make axis functions (`.call(make_x_axis())` and `.call(make_y_axis())`).

Now comes the really interesting bit.

What you will see if you go to the D3 API wiki¹⁸ is that for the `.tickSize` function, the following is the format.

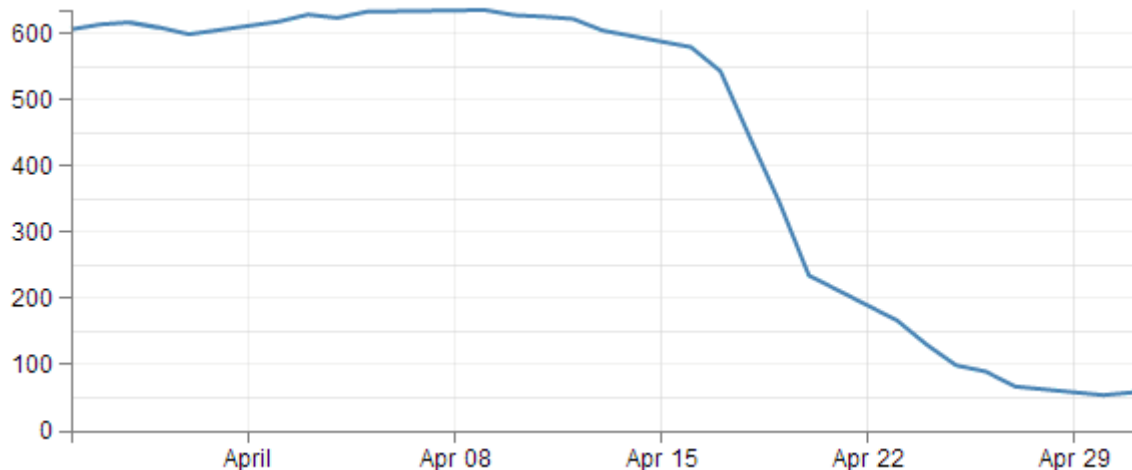
- `axis.tickSize([major[[, minor], end]])`

¹⁸ <https://github.com/mbostock/d3/wiki/SVG-Axes#wiki-tickSize>

That tells us that you get to specify the size of the ticks on the axes, by the major ticks, the minor ticks and the end ticks (that is to say the lines on the very end of the graph which in the case of the example we are looking at aren't there!).

So in our example we are setting our major ticks to a length that corresponds to the full height or width of the graph. Which of course means that they extend across the graph and have the appearance of grid lines! What a neat trick.

Something I haven't done before is to see what would happen if I included the tick lines for the minor and end ticks. So here we go :-)



Darn! Disappointment. We can see a minor tick line for the y axis, but nothing for the x axis and nothing on the ends. Clearly I will have to run some experiments to see what's going on there (later).

The last thing that is included in the code to draw the grid lines is the instruction to suppress printing any label for the ticks;

```
.tickFormat("")
```

After all, that would become a bit confusing to have two sets of labels. Even if one was on top of the other. They do tend to become obvious if that occurs (they kind of bulk out a bit like bold text).

And that's it. Grid lines!

Make a dashed line

Dashed lines totally rock!

OK, there may be an element of exaggeration there, but I certainly found it interesting that there didn't seem to be a lot of explanation for a simple bloke like myself to make a dashed line in D3. So for me they rocked :-)

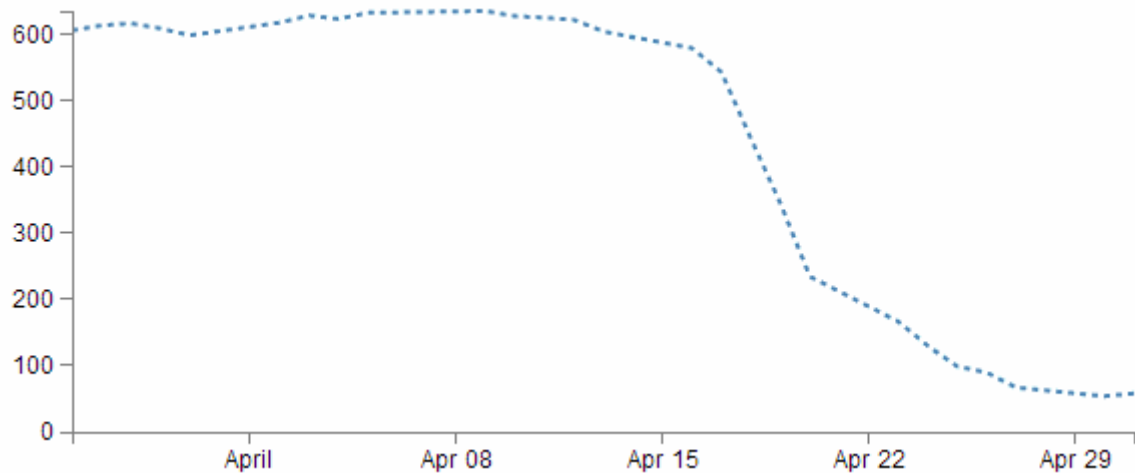
One of the best parts about it is that they're so simple to do!

Literally one line!!!!

So lets imagine that we want to make the line on our simple graph dashed. All we have to do is insert the following line in our JavaScript code here;

```
svg.append("path")
  .attr("class", "line")
  .style("stroke-dasharray", ("3, 3")) // <== This line here!!
  .attr("d", valueline(data));
```

And our graph ends up like this;



Hey! It's dashtastic!

So how does it work?

Well, obviously "[stroke-dasharray](#)" is a style for the path element, but the magic is in the numbers.

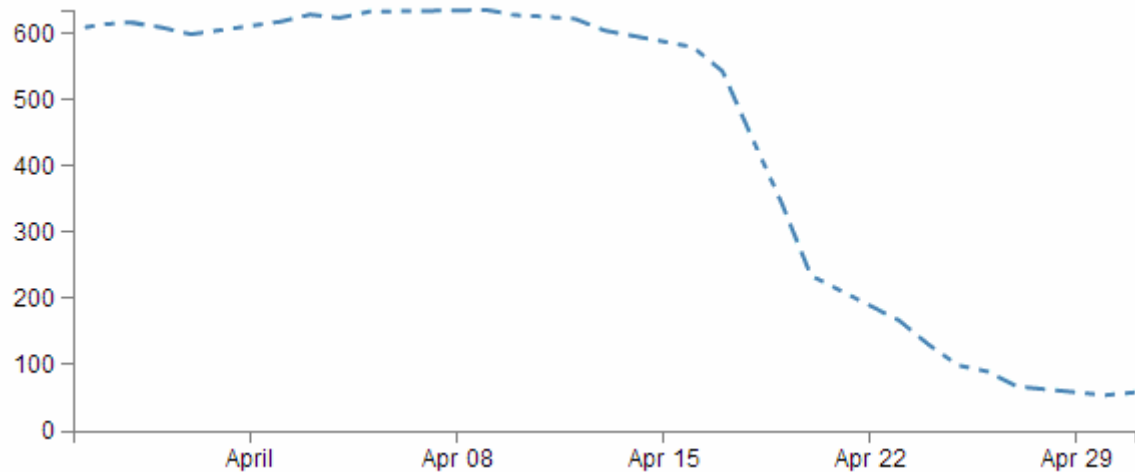
Essentially they describe the on length and off length of the line. So "[3, 3](#)" translates to 3 pixels (or whatever they are) on and 3 pixels off. Then it repeats. Simple eh?

So, experiment time :-)

What would the following represent?

[“5, 5, 5, 5, 5, 5, 10, 5, 10, 5, 10, 5”](#)

Try not to cheat...



Ahh yes, Mr Morse would be proud.

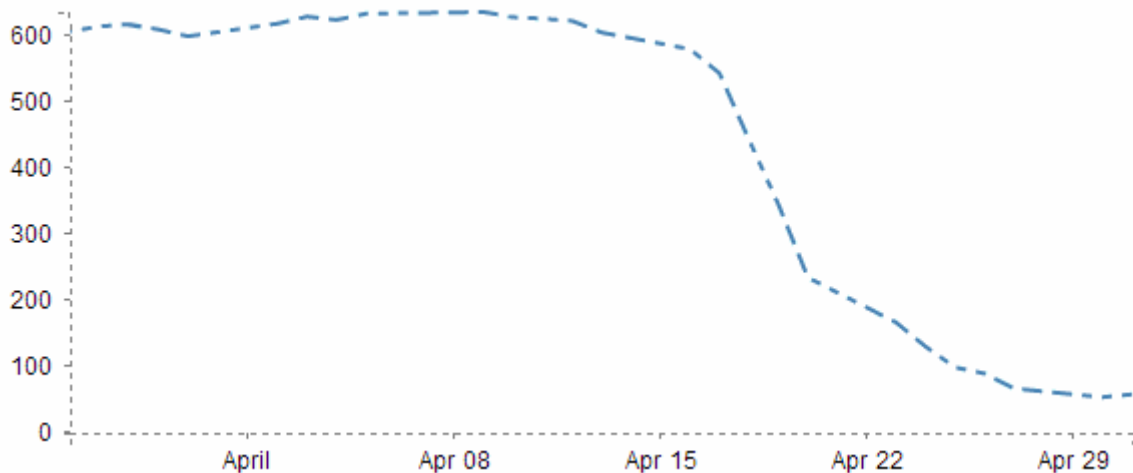
And you can put them anywhere. Here's our axes perverted with dashes;


```

svg.append("g")
  .attr("class", "x axis")
  .attr("transform", "translate(0," + height + ")")
  .style("stroke-dasharray", ("3, 3"))
  .call(xAxis);

svg.append("g")
  .attr("class", "y axis")
  .style("stroke-dasharray", ("3, 3"))
  .call(yAxis);

```



Well... I suppose you can have too much of a good thing. With great power comes great responsibility. Use your dash skills wisely and only for good.

Filling an area under the graph.

Lines are all very well and good, but that's not the whole story for graphs. Some times you've just got to go with a fill.

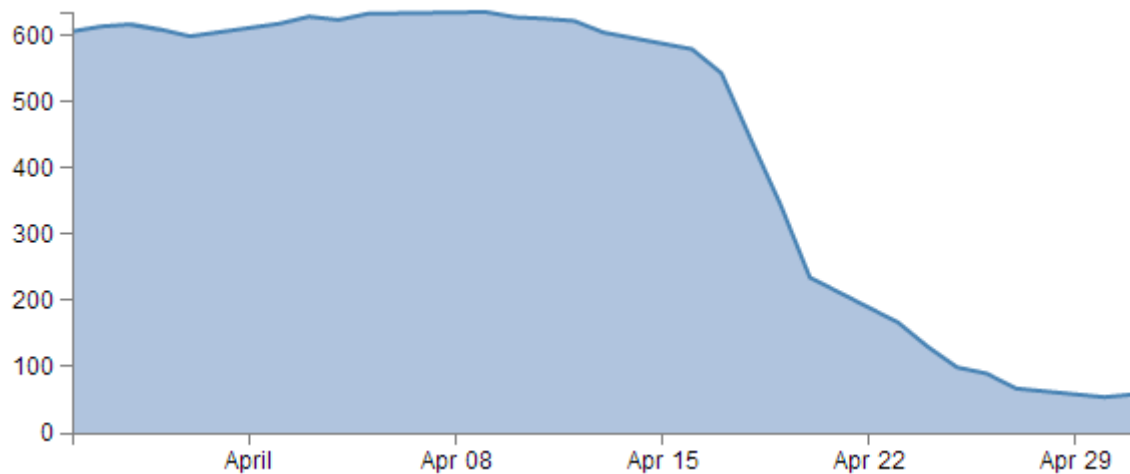
Filling an area with a solid colour isn't too hard. I mean we did it by mistake back a few pages when we were trying to draw a line.

But to do it in a nice coherent way is fairly straight forward.

It takes three sections of code in much the same way that we drew our grid lines earlier it is done in three sections;

1. One in the CSS section to define what style the area will have.
2. One to define the functions that generate the area. And...
3. One to draw the area.

The end result will look a bit like this;



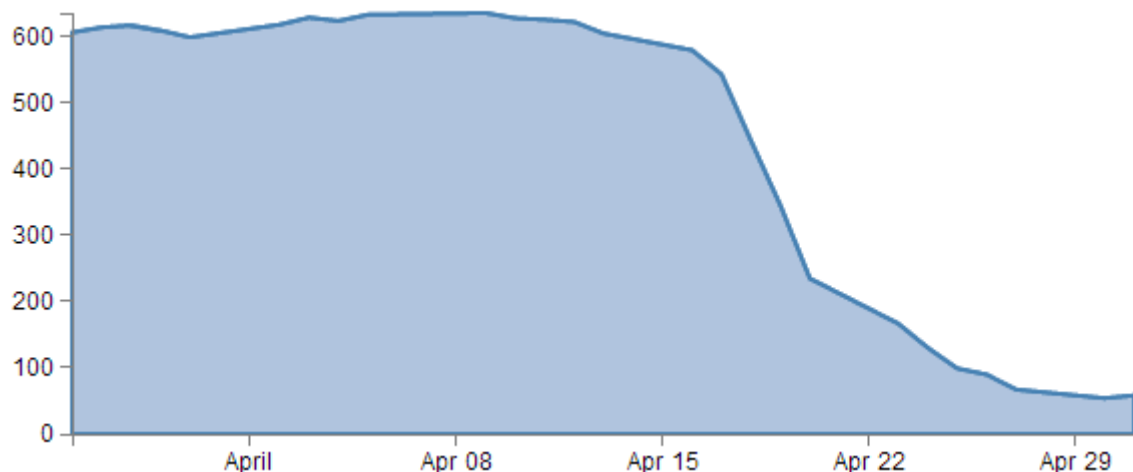
CSS for an area fill

This is pretty straight forward and only consists of two rules;

```
.area {
  fill: lightsteelblue;
  stroke-width: 0;
}
```

Put them at the bottom of your <style> section.

The first one (fill: lightsteelblue;) sets the colour of our fill (and in this case we have chosen a lighter shade of the same colour as our line to match it) and the second one (stroke-width: 0;) sets the width of the line that surrounds the area to zero. This last rule is kind of important in making a filled area work well. The whole idea is that the graph is made up of separate elements that will compliment each other. There's the axes, the line and the fill. If we don't tell the code that there is no line surrounding the filled area, it will assume that there is one and add it in like this.



So what has happened here is that the area element has inherited the line property from the path element and surrounding the area is a 2px wide steelblue line. Not too pretty. Let's not go there.

Define the area function

We need a function that will tell the area what space to fill. This is accessed from the d3.svg.area function¹⁹.

¹⁹ <https://github.com/mbostock/d3/wiki/SVG-Shapes#wiki-area>

The code that we will use is as follows;

```
var area = d3.svg.area()  
  .x(function(d) { return x(d.date); })  
  .y0(height)  
  .y1(function(d) { return y(d.close); });
```

I have placed it in between the axis variable definitions and the line definitions here;

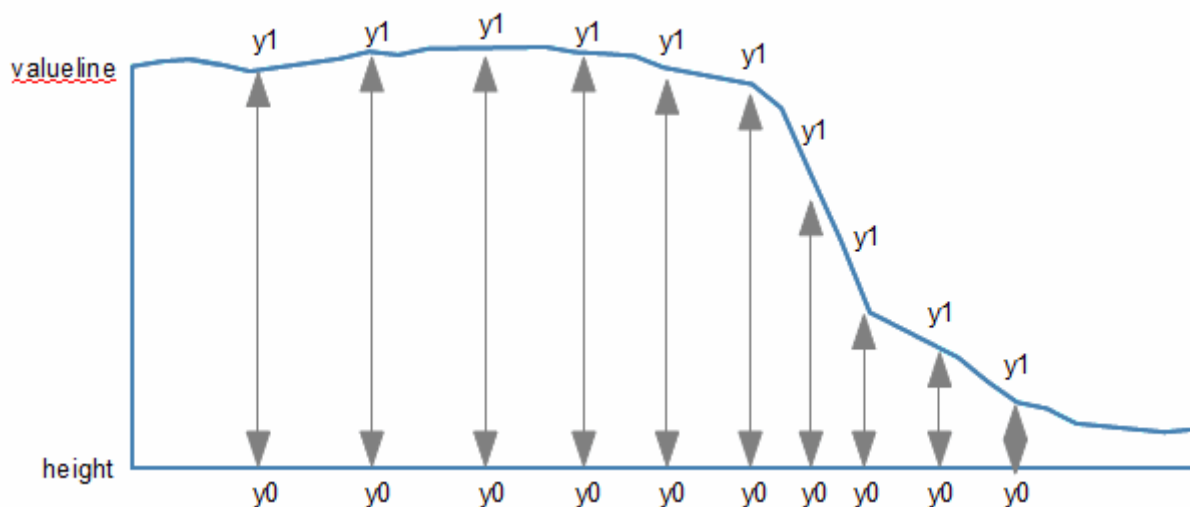
```
var yAxis = d3.svg.axis().scale(y)  
  .orient("left").ticks(5);  
  
var valueline = d3.svg.line()  
  .x(function(d) { return x(d.date); })  
  .y(function(d) { return y(d.close); });
```

<===== Put the new code here!

You will notice it looks *INCREDIBLY* similar to the valueline function definition. That's because while the line definition describes drawing a line that connects a set of coordinates, I imagine the area definition describes drawing two lines that share the same x coordinates, but simultaneously draws two y coordinates, y0 and y1. Then when it's finished drawing the resultant shape, it fills it with the colour of your choosing.

So the only changes to the code are the addition of the 'y0' line and the renaming of the 'y' line 'y1'.

Here's a picture that might help explain;



As should be apparent, the top line (y1) follows the valueline line and the bottom line is at the constant 'height' value. Everything in between these lines is what gets filled. The function in this section describes the area.

Draw the area

Now to the money maker.

The final section of code in the area filling odyssey is as follows;

```
svg.append("path")  
  .datum(data)  
  .attr("class", "area")  
  .attr("d", area);
```

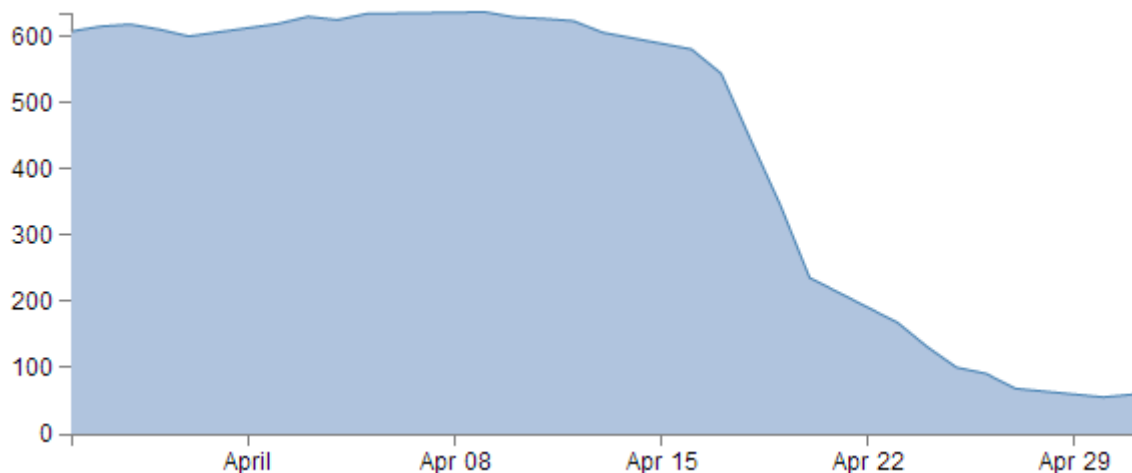
We should place this block directly after the domain functions but before the drawing of the valueline path;

```
x.domain(d3.extent(data, function(d) { return d.date; }));
y.domain([0, d3.max(data, function(d) { return d.close; })]);
                                                                    // <== Area drawing code here!

svg.append("path")
    .attr("class", "line")
    .attr("d", valueline(data));
```

This is actually a pretty good idea to put it there since the various bits and pieces that are drawn in the graph are done so one after the other. This means that the filled area comes first, then the valueline is layered on top and then the axes come last. This is a pretty good sequence since if there are areas where two or more elements overlap, it might cause the graph to look 'wrong'.

For instance, here is the graph drawn with the area added last.



You should be able to notice that part of the valueline line has been obscured and the line for the y axis where it coincides with the area is obscured also.

Looking at the code we are adding here, the first line appends a path element (`svg.append("path")`) much like the script that draws the line.

The second line (`.datum(data)`) declares the data we will be utilising for describing the area and the third line (`.attr("class", "area")`) makes sure that the style we apply to it is as defined in the CSS section (under 'area').

The final line (`.attr("d", area);`) declares "d" as the attributer for path data and calls the 'area' function to do the drawing.

And that's it!

Filling an area above the line

Pop Quiz:

How would you go about filling the area *ABOVE* the graph?

Now it might sound a little trite, but believe it or not, this could come in handy. For instance, what if you want to highlight an area that was too high and an area that was too low for a line of data on a graph with an area in the centre where a projected 'normal' set of values should be present?

In this instance, you could fill the lower area as has been demonstrated here, and with a small change you can fill another area with a solid colour above another line.

How is this incredible feat achieved?

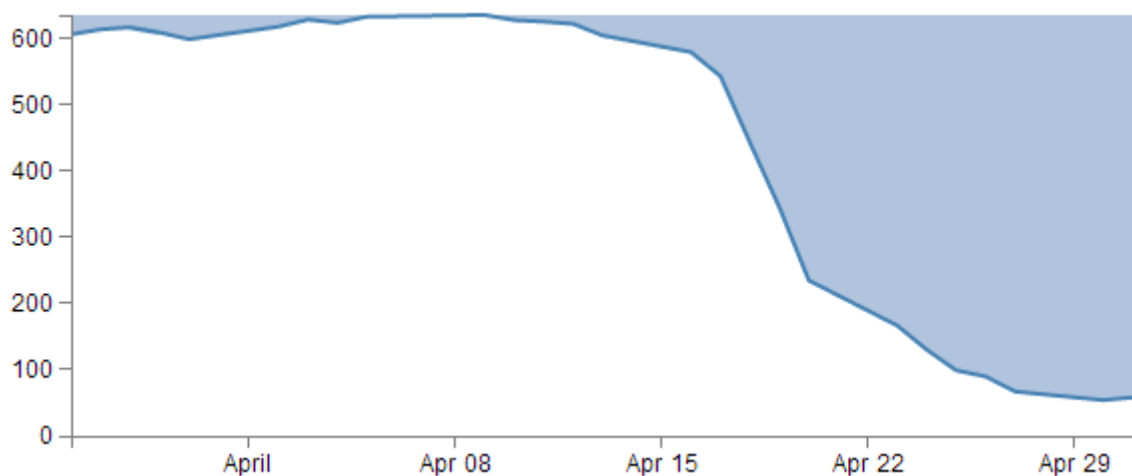
Well, remember the code that defined the area?

```
var area = d3.svg.area()  
  .x(function(d) { return x(d.date); })  
  .y0(height)  
  .y1(function(d) { return y(d.close); });
```

All we have to do is tell it that instead of setting the y0 constant value to the height of the graph (remember, this is the bottom of the graph) we will set it to the constant value that is at the top of the graph. In other words zero (0).

```
.y0(0)
```

That's it.



Now, I'm not going to go over the process of drawing two lines and filling each in different directions to demonstrate the example I described, but this provides a germ of an idea that you might be able to flesh out :-)

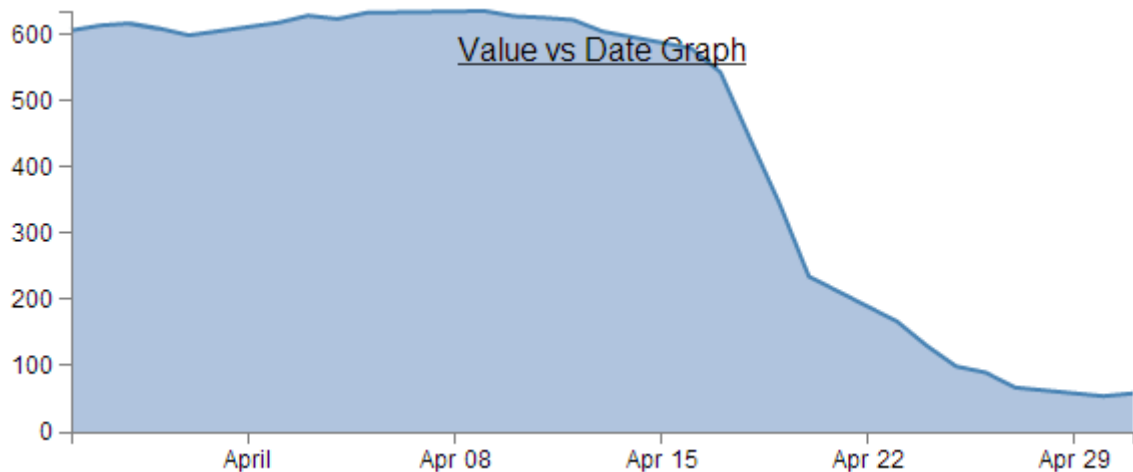
Adding a drop shadow to allow text to stand out on graphics.

I've deliberately positioned this particular tip to follow the 'filling an area' description because it provides an opportunity to demonstrate the principle to slightly better effect.

There have been several opportunities where I have wanted to place text overlaid on graphs for convenience sake only to have it look overly messy as the text interferes with the graph.

Now, I'll be the first to say that the principle of overlaying text on a graph is probably not best practice, but sometimes you've got to do what you've got to do. Besides. Sometimes it's a valid idea. If I remember rightly, the first time I came across this idea, it was being used to highlight text when positioned on bars of a bar graph. So it's not always an evil practice :-).

Anyway, what we'll do is leave the fill in place and place the title back on the graph, but position the title so that it lays on top of the fill like so;

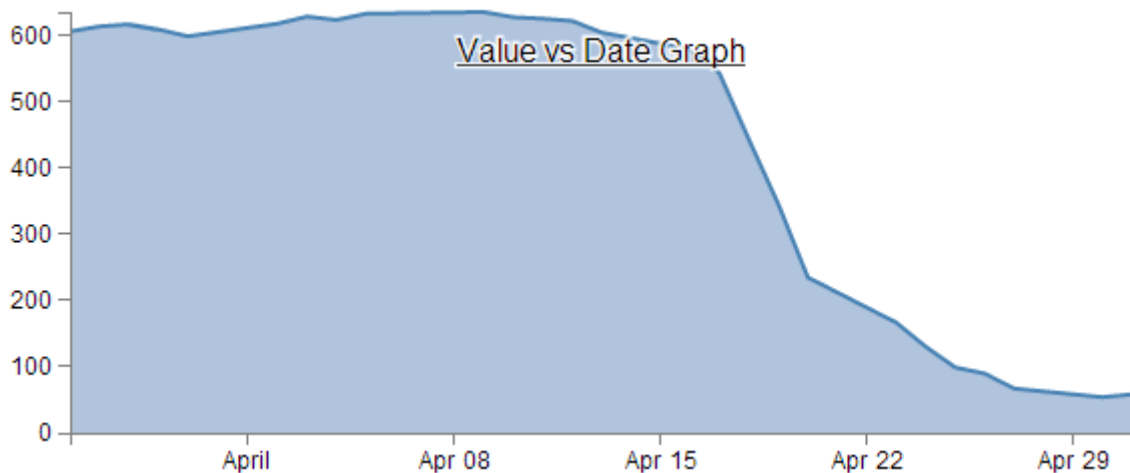


The additional code for the title is the following and appears just after the drawing of the axes.

```
svg.append("text")
  .attr("x", (width / 2))
  .attr("y", 25 )
  .attr("text-anchor", "middle")
  .style("font-size", "16px")
  .style("text-decoration", "underline")
  .text("Value vs Date Graph");
```

(the only change from the previous title example is the 'y' attribute which has been hard coded to 25 to place it inconveniently on the filled area.)

So, what we want to end up with is something like the following...



In my humble opinion, it's just enough to make the text acceptable :-).

The method that I'll describe to carry this out is designed so that the drop shadow effect can be applied to any of the text in a graph, not the isolated example that we will use here. In order to implement this marvel of utility we will need to make changes in two areas. One in the CSS where we will define a style for white shadowy backgrounds and the second to draw it.

CSS for white shadowy background

The code to add to the CSS section is as follows;

```
text.shadow {
  stroke: white;
  stroke-width: 2.5px;
  opacity: 0.9;
}
```

The first line designates that the style applies to text with a 'shadow' label. The stroke is set to white, the width of the line is set to 2.5px and it is made to be slightly see-through. So by setting the line that surrounds the text to be thick, white and see-through gives it a slightly 'cloudy' effect. If we remove the black text from over the top we get a slightly better look;



Of course if you want to have a play with any of these settings, you should have a go and see what works best for your graph.

Drawing the white shadowy background.

Now that we've set the style for our background, we need to draw it in.

The code for this should be extremely familiar;

```
svg.append("text")
  .attr("x", (width / 2))
  .attr("y", 25 )
  .attr("text-anchor", "middle")
  .style("font-size", "16px")
  .style("text-decoration", "underline")
  .attr("class", "shadow") // <=== Here's the different line
  .text("Value vs Date Graph");
```

That's because it's identical to the piece of code that was used to draw the title except for the one line that is indicated above. The reason that it's identical is that what we are doing is placing a white shadow on the graph and then the text on top of it, if it deviated by a significant amount it will just look silly. Of course a slight amount could look effective, in which case adjust the 'x' or 'y' attributes.

One of the things I pointed out in the previous paragraph was extremely important. That's the bit that tells you that we needed to place the shadow before we placed the black text. For the same reason that we placed the area fill on first in the area fill example, If black text goes on before the shadow, it will look pretty silly. So place this block of code just before the block that draws the title.

So the line that has been added in is the one that tells D3 that the text that is being drawn will have the white cloudy effect. And at the risk of repeating myself, if you have several text elements that could benefit from this effect, once you have the CSS code in place, all you need to do is duplicate the block that adds the text and add in that single line and viola!

Adding more than one line to a graph

All right, we're starting to get serious now. Two lines on a graph is a bit of a step into a different world in one respect. I mean that in the sense that there's more than one way to carry out the task, and I tend to do it one way and not the other mainly because I don't fully understand the other way :-).

I should stress that that's not because it's more complex, or that it's a bad way, it's just that once I started doing things one way, I haven't come across a need to do things another way. There's a good chance I will have to revisit this decision in the future, but for now I'll keep moving.

So, how are we going to do this? I think that the best way will be to make the executive decision that we have suddenly come across more data and that it is also in our data.tsv file. In fact it looks a little like this (apologies in advance for the big ugly block of data);

date	close	open
1-May-12	58.13	34.12
30-Apr-12	53.98	45.56
27-Apr-12	67.00	67.89
26-Apr-12	89.70	78.54
25-Apr-12	99.00	89.23
24-Apr-12	130.28	99.23
23-Apr-12	166.70	101.34
20-Apr-12	234.98	122.34
19-Apr-12	345.44	134.56
18-Apr-12	443.34	160.45
17-Apr-12	543.70	180.34
16-Apr-12	580.13	210.23
13-Apr-12	605.23	223.45
12-Apr-12	622.77	201.56
11-Apr-12	626.20	212.67
10-Apr-12	628.44	310.45
9-Apr-12	636.23	350.45
5-Apr-12	633.68	410.23
4-Apr-12	624.31	430.56
3-Apr-12	629.32	460.34
2-Apr-12	618.63	510.34
30-Mar-12	599.55	534.23
29-Mar-12	609.86	578.23
28-Mar-12	617.62	590.12
27-Mar-12	614.48	560.34
26-Mar-12	606.98	580.12

Three columns, date open and close. The first two are exactly what we have been dealing with all along and the last (open) is our new made up data. Each column is separated by a tab (hence .tsv (Tab Separated Values)), which is the format we're currently using to import data.

We should save this as a new file so we don't mess up our previous data, so let's call it data2.tsv.

We will be using our simple graph template to start with, so the immediate consequence of this is that we need to edit the line that was looking for 'data.tsv'. To reflect the new name.

```
d3.tsv("data/data2.tsv", function(error, data) {
```


So when you browse to our new graph html file, we don't see any changes. It still happily loads the new data, but because it hasn't been told to do anything with it, nothing new happens.

What we need to do now is to essentially duplicate the code blocks that drew the first line for the second line.

The good news is that in the simplest way possible that's just two code blocks.

The first sets up the function that defines the new line;

```
var    valueline2 = d3.svg.line()
      .x(function(d) { return x(d.date); })
      .y(function(d) { return y(d.open); });
```

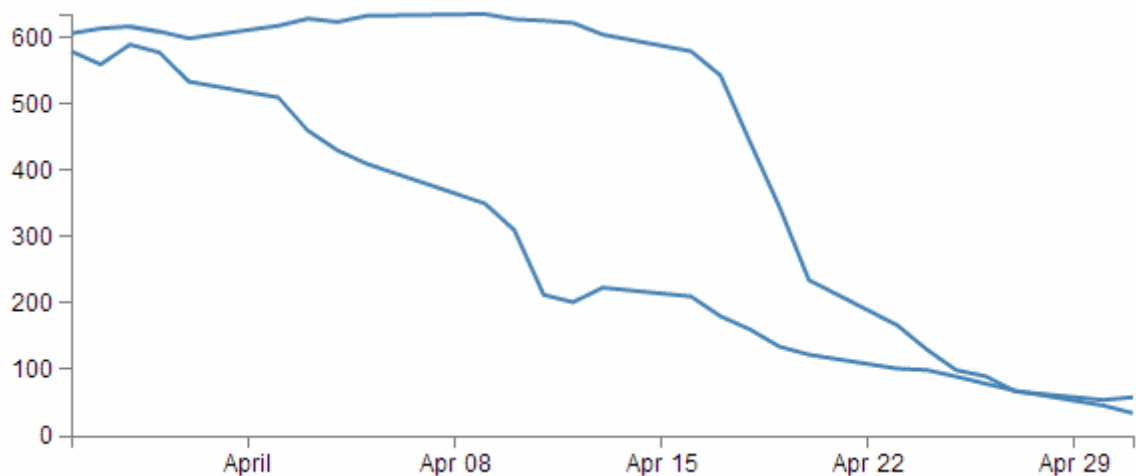
You should notice that this block is identical to the block that sets up the function for the first line, except this one is called (imaginatively) `valueline2`. We should put it directly after the block that sets up the function for `valueline`.

The second block draws our new line;

```
svg.append("path")           // Add the valueline2 path.
   .attr("class", "line")
   .attr("d", valueline2(data));
```

Again, this is identical to the block that draws the first line, except this one is called `valueline2`. We should put it directly after the block that draws `valueline`.

After those three small changes, check out your new graph;

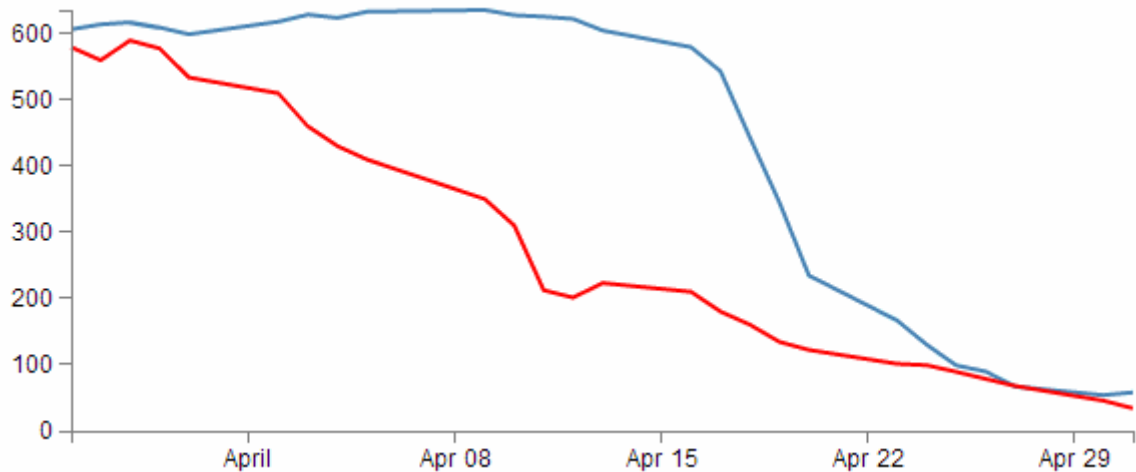


Hey! Two lines! Hmm.... Both being the same colour is a bit confusing. Good news. We can change the colour of the second line by inserting a line that adjusts its stroke (colour) very simply.

So here's what our new drawing block looks like;

```
svg.append("path")           // Add the valueline2 path.
   .attr("class", "line")
   .style("stroke", "red")
   .attr("d", valueline2(data));
```

And as if by magic, here's our new graph;



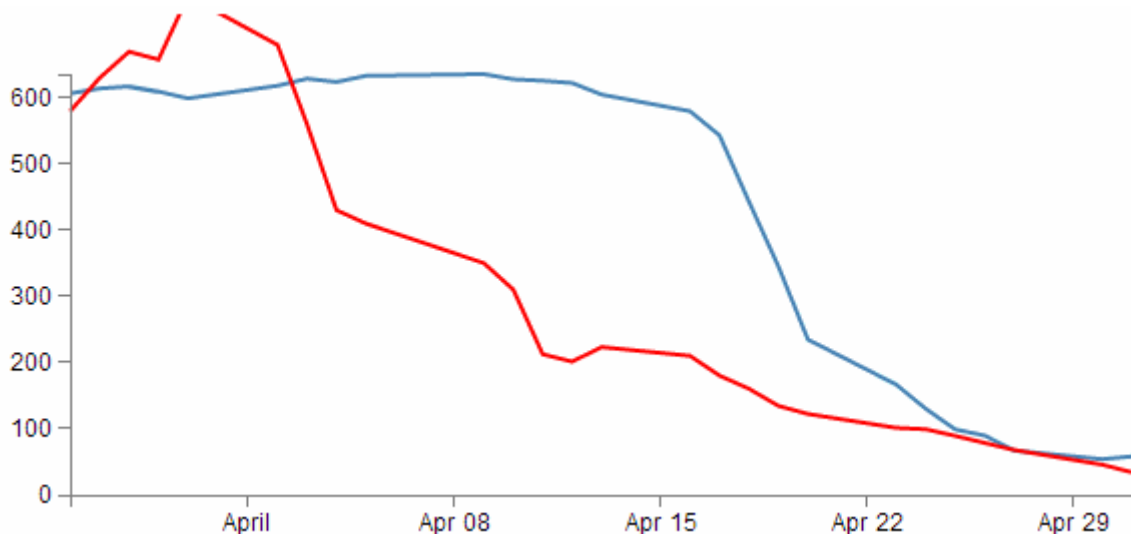
Wow. Right about now, we're thinking ourselves pretty clever. But there's two places where we're not doing things right. We took a simple way, but we took some short cuts that might bite us in the posterior.

The first mistake we made was not ensuring that our variable “d.open” is being treated as a number or a string. We're fortunate in this case that it is, but this can't always be assumed. So, this is an easy fix and we just need to put the following (indicated line) in our code;

```
// Get the data
d3.tsv("data/data.tsv", function(error, data) {
  data.forEach(function(d) {
    d.date = parseDate(d.date);
    d.close = +d.close;
    d.open = +d.open;           // <=== Add this line in!
  });
});
```

The second and potentially more fatal flaw is that nowhere in our code do we make allowance for our second set of data (the second line's values) exceeding our first lines values.

Now that might not sound too normal straight away, but consider this. What if when we made up our data earlier, some of the new data exceeded our maximum value in our original data? As a means of demonstration, here's what happens when our second line of data has values higher than the first lines;



Ahh.... We're not too clever now.

Good news though, we can fix it!

The problem comes about because when we set the domain for the y axis this is what we put in the code;

```
y.domain([0, d3.max(data, function(d) { return d.close; })]);
```

So that only considers d.close when establishing the domain. With d.open exceeding our domain, it just keeps drawing off the graph!

The good news is that 'Bill' has provided a solution for just this problem here;

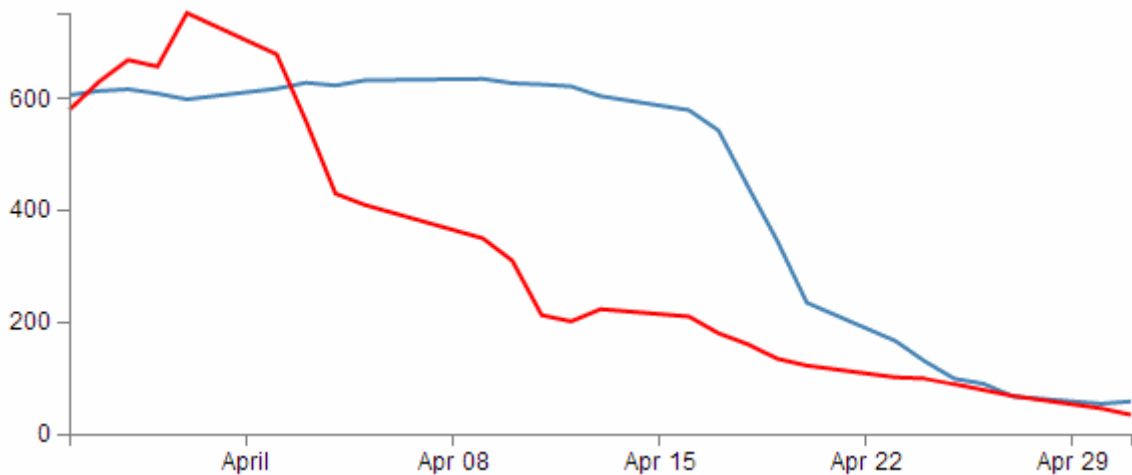
<http://stackoverflow.com/questions/12732487/d3-js-dataset-array-w-multiple-y-axis-values>

So all you need to replace the y.domain line with is this;

```
y.domain([0, d3.max(data, function(d) { return Math.max(d.close, d.open); })]);
```

It does much the same thing, but this time it returns the maximum of d.close and d.open (whichever is largest). Good work Bill.

If we put that code into the graph with the higher values for our second line we are now presented with this;



And it doesn't matter which of the two sets of data is largest, the graph will always adjust :-)

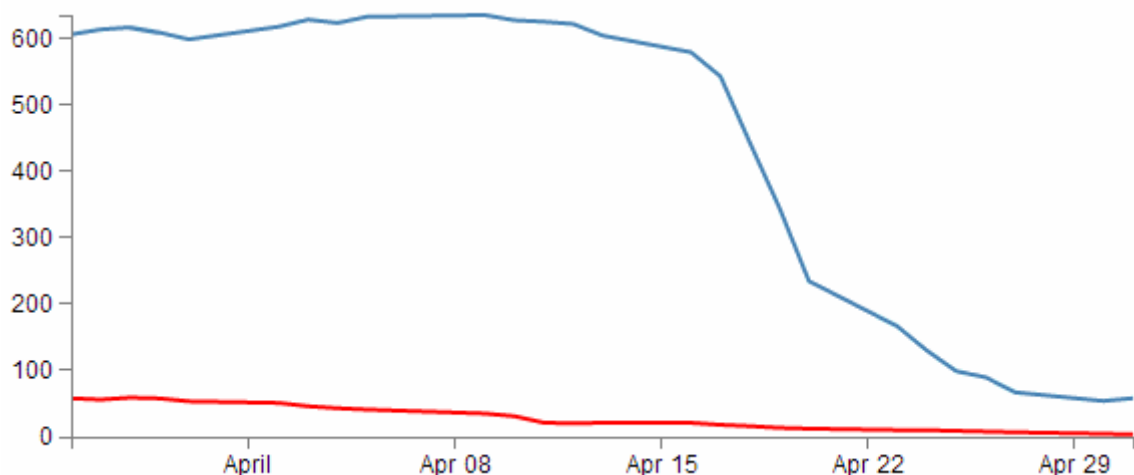
You will also have noticed that our y axis has auto adjusted again to cope. Clever eh?

Multiple axes for a graph

Alrighty... Let's imagine that we want to show our wonderful graph with two lines on it much like we already have, but that the data that the lines is made from is significantly different in magnitude from the original data (in the example below, the data for the second line has been reduced by approximately a factor of 10 from our original data).

date	close	open
1-May-12	58.13	3.41
30-Apr-12	53.98	4.55
27-Apr-12	67.00	6.78
26-Apr-12	89.70	7.85
25-Apr-12	99.00	8.92
24-Apr-12	130.28	9.92
23-Apr-12	166.70	10.13
20-Apr-12	234.98	12.23
19-Apr-12	345.44	13.45
18-Apr-12	443.34	16.04
17-Apr-12	543.70	18.03
16-Apr-12	580.13	21.02
13-Apr-12	605.23	22.34
12-Apr-12	622.77	20.15
11-Apr-12	626.20	21.26
10-Apr-12	628.44	31.04
9-Apr-12	636.23	35.04
5-Apr-12	633.68	41.02
4-Apr-12	624.31	43.05
3-Apr-12	629.32	46.03
2-Apr-12	618.63	51.03
30-Mar-12	599.55	53.42
29-Mar-12	609.86	57.82
28-Mar-12	617.62	59.01
27-Mar-12	614.48	56.03
26-Mar-12	606.98	58.01

Now this isn't a problem in itself. D3 will still make a reasonable graph of the data, but because of the difference in range, the detail of the second line will be lost.



So what I'm proposing is that we have a second y axis on the right hand side of the graph that relates to the red line.

The adoption I've done here is based on the great examples put forward by Ben Christensen here: <http://benchristensen.com/2012/05/02/line-graphs-using-d3-js/>.

Now... You'll want to concentrate a bit here since there are quite a few different bits to change and adapt, but don't despair, they're all quite logical and make sense.

First things first, there won't be space on the right hand side of our graph to show the extra axis, so we should make our right hand margin a little larger.

```
var margin = {top: 30, right: 40, bottom: 30, left: 50},
```

I went for 40 and it seems to fit pretty well.

Then (and here's where the main point of difference for this graph comes in) you want to amend the code to separate out the two scales for the two lines in the graph. This is actually a lot easier than it sounds, since it consists mainly of finding anywhere that mentions 'y' and replacing it with 'y0' and then adding in a reciprocal piece of code for 'y1'.

The idea here is that we will be creating two references for the y axis. One for each column of data. Then when we draw the lines the scales will automatically scale the data correctly (and separately) to our canvas and we will draw two different y axes with the different scales. Believe it or not, it's sounds a lot harder than it is.

Let's get started.

Firstly, change the variable declaration for 'y' to 'y0' and add in 'y1'.

```
var x = d3.time.scale().range([0, width]);
var y0 = d3.scale.linear().range([height, 0]);
var y1 = d3.scale.linear().range([height, 0]);
```

Then change our yAxis declaration to be specific for 'y0' and specifically 'left'. And add in a declaration for the right hand axis;

```
var yAxisLeft = d3.svg.axis().scale(y0) // <== Add in 'Left' and 'y0'
    .orient("left").ticks(5);

var yAxisRight = d3.svg.axis().scale(y1) // This is the new declaration for the 'Right', 'y1'
    .orient("right").ticks(5); // and includes orientation of the axis to the right.
```

Note the orientation change for the right hand axis.

Now change our valueline declarations so that they refer to the 'y0' and 'y1' scales.

```
var valueline = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y0(d.close); }); // <== y0

var valueline2 = d3.svg.line()
    .x(function(d) { return x(d.date); })
    .y(function(d) { return y1(d.open); }); // <== y1
```

There are a few different ways for the scaling to work, but we'll stick with the fancy max method we used in the dual line example (although technically it's not required).

```
y0.domain([0, d3.max(data, function(d) { return Math.max(d.close); })]);
y1.domain([0, d3.max(data, function(d) { return Math.max(d.open); })]);
```

Again, here's the 'y0' and 'y1' changed and added and the maximums for 'd.close' and 'd.open' are separated out).

The final piece of the puzzle is to draw the new axis, but we also want to make a slight change to the original y axis. Since we have two lines and two axes, we need to know which belongs to which, so we can colour code the text in the axes to match the lines;

```

svg.append("g")
  .attr("class", "y axis")
  .style("fill", "steelblue")
  .call(yAxisLeft);

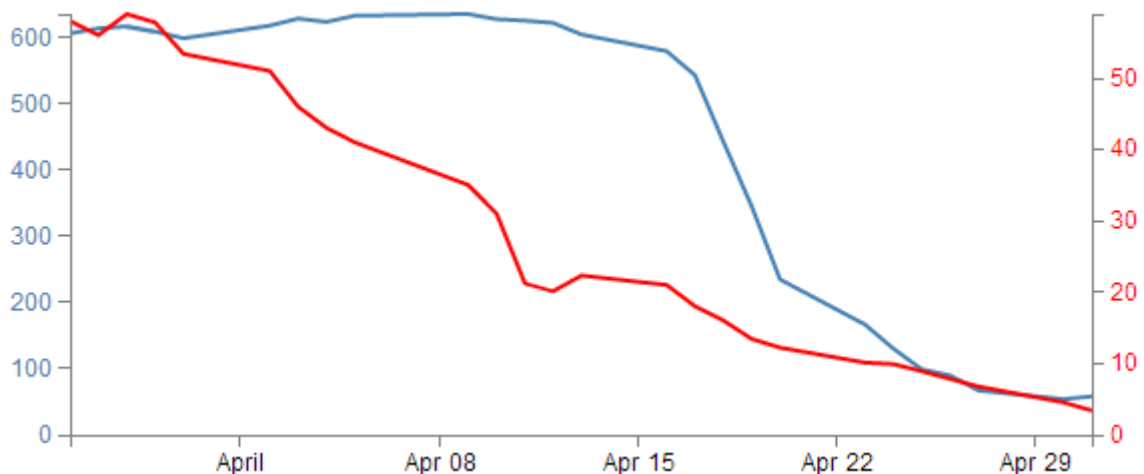
svg.append("g")
  .attr("class", "y axis")
  .attr("transform", "translate(" + width + ",0)")
  .style("fill", "red")
  .call(yAxisRight);

```

In the above code you can see where we have added in a 'style' change for the yAxisLeft to make it 'steelblue' and a complementary change in the new section for yAxisRight to make that text red.

The yAxisRight section obviously needs to be added in, but the only significant difference is the transform / translate attribute that moves the axis to the right hand side of the graph.

And after all that, here's the result...



Now, let's not kid ourselves that it's a thing of beauty, but we should console our aesthetic concerns with the warm glow of understanding how the function works :-).

How to rotate the text labels for the x Axis.

The observant reader will recall the problem we had observed earlier when increasing the number of ticks on our x axis to 10. The effect had been to produce a large number of x axis ticks (actually 19) but they had run together and become unreadable.

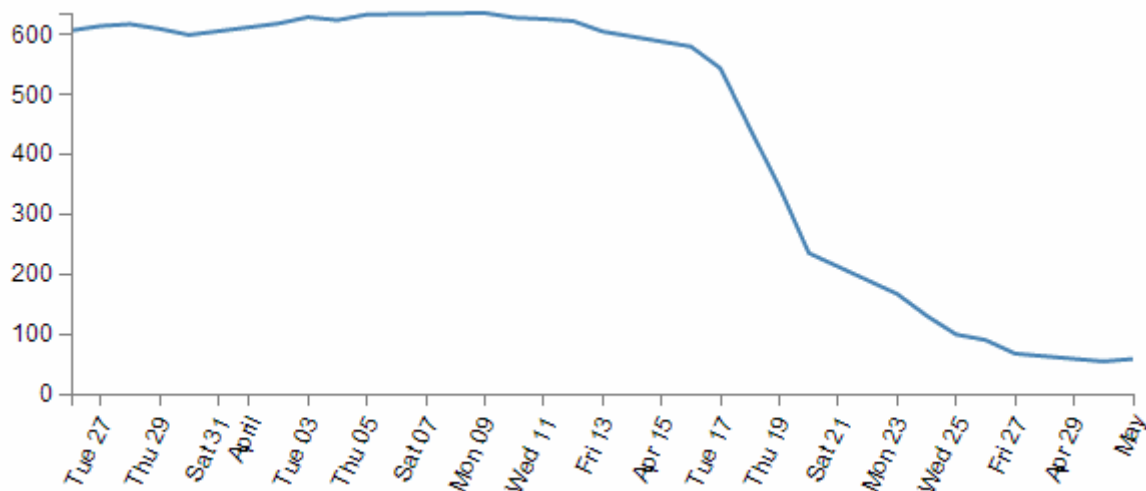
The `.style("text-anchor", "end")` line ensures that the text label has the end of the label 'attached' to the axis tick. This has the effect of making sure that the text rotates about the end of the date. This makes sure that the text all ends up a uniform distance from the axis ticks.

The 'dx' and 'dy' attribute lines move the end of the text just far enough away from the axis tick so that they don't crowd it and not too far away so that it appears disassociated. This took a little bit of fiddling to get right and you will notice that I've used the 'em' units to get an adjustment if the size of the font differs.

The final action is kind of the money shot.

The transform attribute applies itself to each text label and rotates each line by -65 degrees. I selected -65 degrees just because it looked OK. There was no deeper reason.

The end result then looks like the following;



This was a surprisingly difficult problem to find a solution to that I could easily understand (well done Aaron). So that makes me think that there are some far deeper mysteries to it that I don't fully appreciate that could trip this solution up. But in lieu of that, enjoy!

Format a date / time axis with specified values

OK then. We've been very clever in rotating our text, but you will notice that D3 has used it's own good judgement as to what format the days / date will be represented as.

Not that there's anything wrong with it, but what if we want to put a specific format of date / time nomenclature as axis labels?

No problem. D3 has your back.

This is actually a pretty easy thing to do, but there are plenty of options for the formatting, so the only really tricky part is deciding what to put where.

But before we start doing anything we are going to have to expand our bottom margin even more than we did with the rotate the axis labels feature.

```
var margin = {top: 30, right: 40, bottom: 70, left: 50},
```

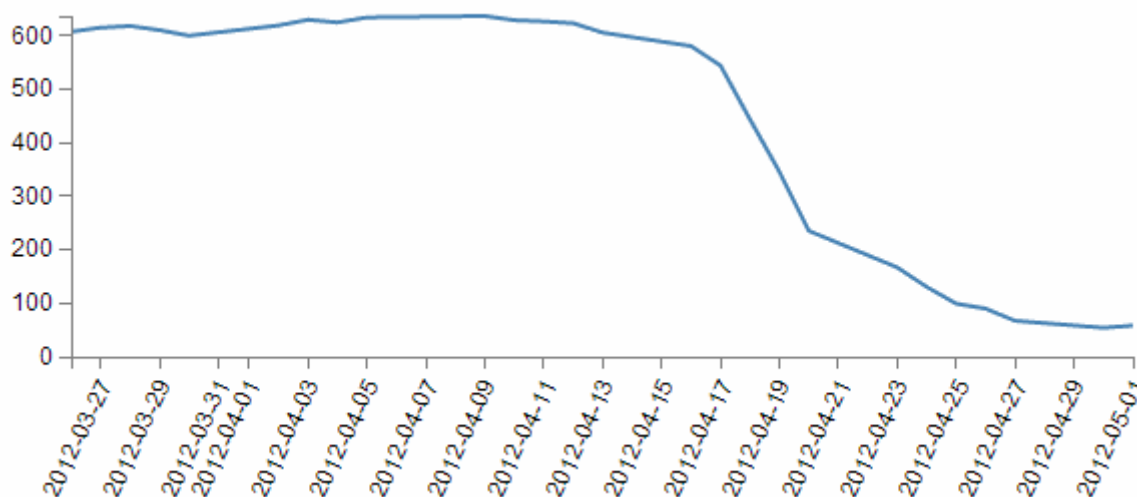
That should see us right.

Right, now the simple part :-). changing the format of the label is as simple as inserting the tickFormat command into the xAxis declaration a little like this;


```
var xAxis = d3.svg.axis().scale(x)
    .orient("bottom").ticks(10)
    .tickFormat(d3.time.format("%Y-%m-%d")); // <== insert the tickFormat function
```

So, what the tickFormat is allowing the setting of formatting for the tick labels. The `d3.time.format` portion of the code is specifying the exact format of those ticks. This formatting is described using the same arguments that were explained in the section on formatting date time values starting on page 21 (or of course the source here; <https://github.com/mbostock/d3/wiki/Time-Formatting>). That means that the examples we see here (`%Y-%m-%d`) should display the year as a four digit number then a hyphen then the month as a two digit number, then another hyphen, then a two digit number corresponding to the day.

Let's take a look at the result;



There we go! You should be able to see this file in the downloads section on d3noob.org with the general examples as `formatted-date-time-axis-labels.html`.

So how about we try something a little out of the ordinary (extreme)?

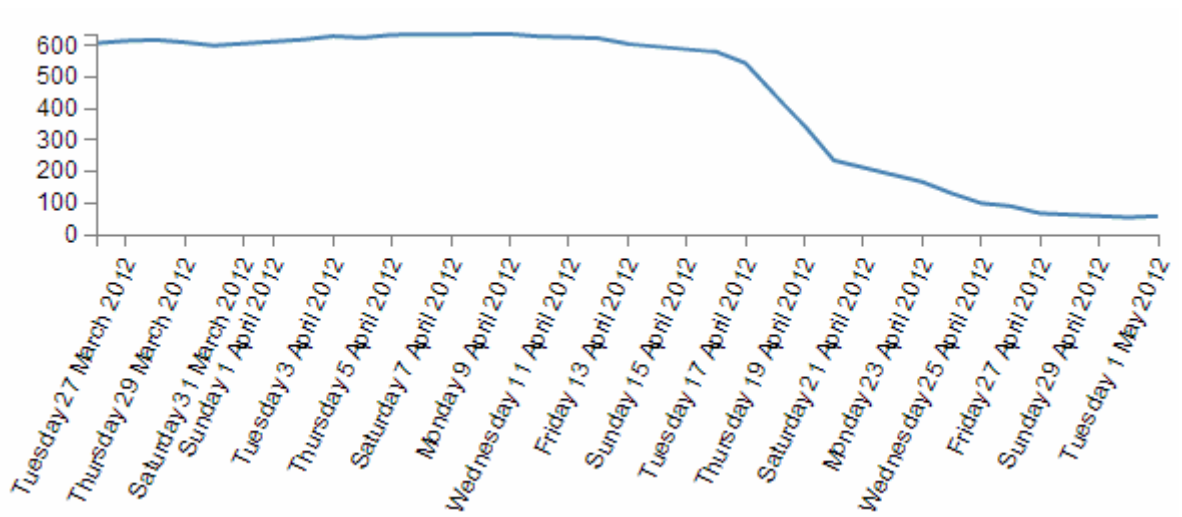
How about the full weekday name (`%A`), the day (`%d`), the full month name (`%B`) and the year (`%Y`) as a four digit number?

```
.tickFormat(d3.time.format("%A %d %B %Y"));
```

We will also need some extra space for the bottom margin, so how about 140?

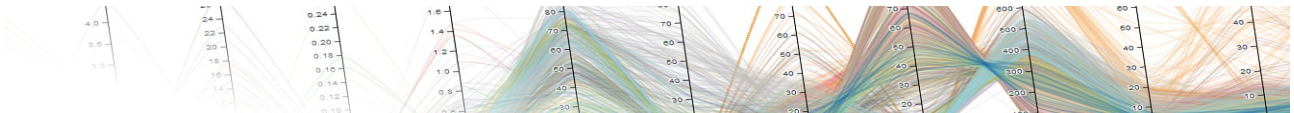
```
var margin = {top: 30, right: 40, bottom: 140, left: 50},
```

and....



Oh yeah... When axis ticks go bad...

But seriously, that does work as a pretty good example of the flexibility available.



Change a line chart into a scatter plot

Confession time.

I didn't actually intend to add in a section with a scatter plot in it for its own sake because I thought it would be;

- a) tricky
- b) not useful
- c) all of the above

I was wrong on all counts.

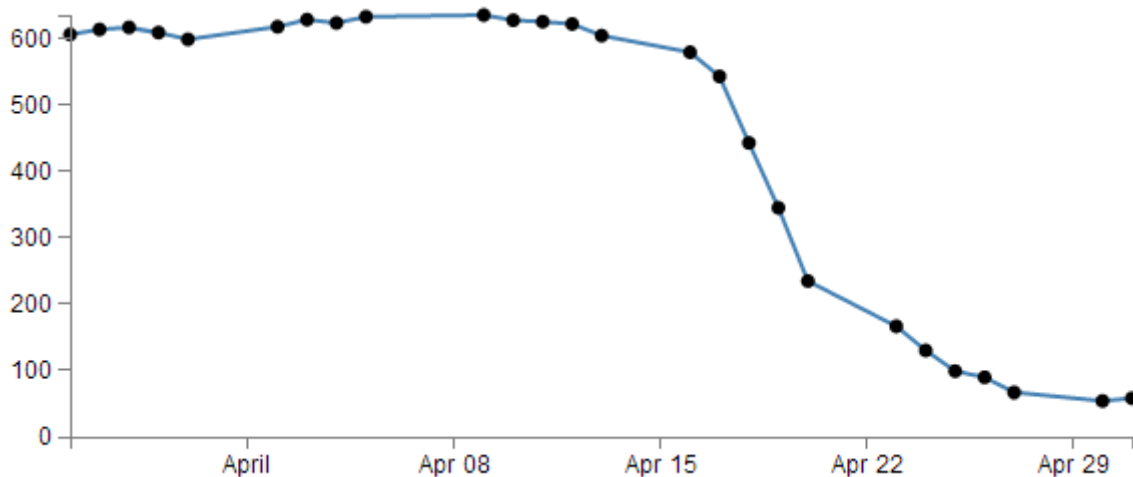
I *did* want to have a scatter plot, because I wanted to display tool tips, but this is too neat to ignore.

It was literally a 5 minute job, 3 minutes of which was taken up by going to the d3 gallery on the wiki (<https://github.com/mbostock/d3/wiki/Gallery>) and ogling at the cool stuff there before snapping out of it and going to the scatter plot example (<http://bl.ocks.org/3887118>).

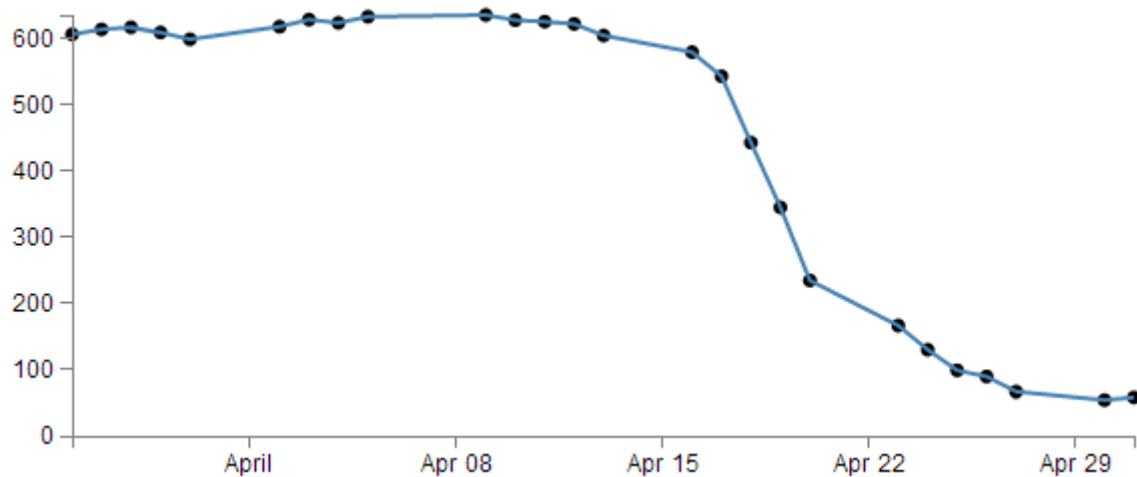
All you need to so is take the simple graph example file and slot the following block in between the 'Add the valueline path' and the 'add the x axis' blocks.

```
svg.selectAll("dot")
  .data(data)
  .enter().append("circle")
  .attr("r", 3.5)
  .attr("cx", function(d) { return x(d.date); })
  .attr("cy", function(d) { return y(d.close); });
```

And you will get...

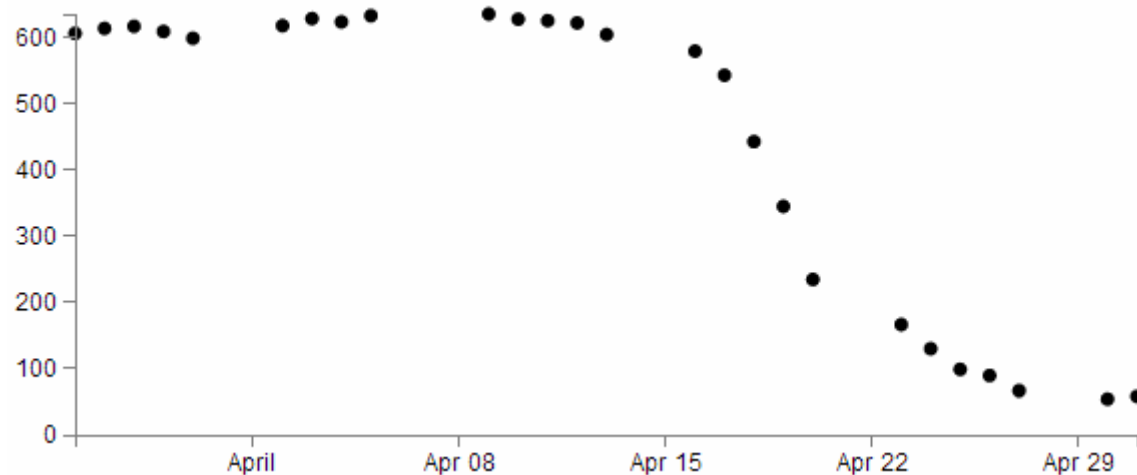


Now I deliberately put the dots after the line in the drawing section, because I thought they would look better, but you could put the block of code before the line drawing block to get the following effect;



(just trying to reinforce the concept that 'order' matters when drawing objects :-)).

You could of course just remove the line block all together...



But in my humble opinion it loses something.

So what do the individual lines in the scatter plot block of JavaScript do?

The first line (`svg.selectAll("dot")`) essentially provides a suitable grouping label for the svg circle elements that will be added. The next line associates the range of data that we have to the group of elements we are about to add in.

Then we add a circle for each data point (`.enter().append("circle")`) with a radius of 3.5 pixels (`.attr("r", 3.5)`) and appropriate x (`.attr("cx", function(d) { return x(d.date); })`) and y (`.attr("cy", function(d) { return y(d.close); })`) coordinates.

There is lots more that we could be doing with this piece of code (check out the scatter plot example (<http://bl.ocks.org/3887118>)) including varying the colour or size or opacity of the circles depending on the data and all sorts of really neat things, but for the mean time, there we go. scatter plot!

I've placed a copy of the file for drawing the scatter plot into the downloads section on d3noob.org with the general examples as `simple-scatterplot.html`.

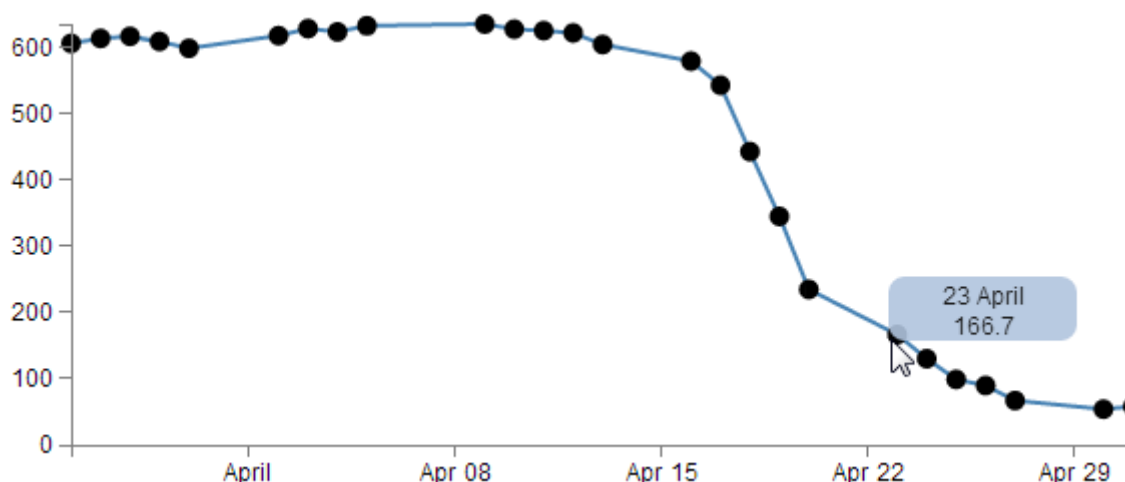
Adding tooltips.

Tooltips have a marvellous duality. They are on one hand a pretty darned useful thing that aids in giving context and information where required and on the other hand, if done with a bit of care, they can look very stylish :-).

Technically, they represent a slight move from what we have been playing with so far into a mildly more complex arena of 'transitions' and 'events'. You can take this one of two ways. Either accept that it just works and implement it as shown, or you will know what's going on and feel free to deride my efforts as those of a rank amateur :-).

The source for the implementation was taken from Mike Bostock's example here; <http://bl.ocks.org/1087001>. This was combined with a few other bit's and pieces (the trickiest being working out how to format the displayed date correctly and inserting a line break in the tooltip (which I found here; <https://groups.google.com/forum/?fromgroups=#!topic/d3-js/GgFTf24ltjc> (well done to all those participating in that discussion)). I make the assumption that any or all errors that occur in the implementation will be mine, whereas, any successes will be down to the original contributors.

So, just in case there is some degree of confusion, a tooltip (one word or two?) is a discrete piece of information that will pop into view when the mouse is hovered over somewhere specific. Most of us have seen and used them, but I suppose we all tend to call them different things such as 'infotip', 'hint' or 'hover box' I don't know if there's a right name for them, but here's an example of what we're trying to achieve;



You can see the mouse has hovered over one of the scatter plot circles and a tip has appeared that provides the user with the exact date and value for that point.

Now, you may also notice that there's a certain degree of 'fancy' here as the information is bound by a rectangular shape with rounded corners and a slight opacity. The other piece of 'fancy' which you don't see in a pdf is that when these tool tips appear and disappear, they do so in an elegant fade-in, fade-out way. Purty.

Now, before we get started describing how the code goes together, let's take a quick look at the two technique specifics that I mentioned earlier, 'transitions' and 'events'.

Transitions

From the main d3.js web page (d3js.org) transitions are described as gradually interpolating styles and attributes over time. So what I take that to mean is that if you want to change an object, you can do so by simply specifying the attribute / style end point that you want it to end up with and the time you want it to take and go!

Of course, it's not quite that simple, but luckily, smarter people than I have done some fantastic work describing different aspects of transitions so please see the following for a more complete description of the topic;

- Mike Bostock's Bar chart tutorial (<http://mbostock.github.com/d3/tutorial/bar-2.html>)
- Christophe Viau's 'Try D3 Now!' tutorial (http://christopheviau.com/d3_tutorial/)

Hopefully observing the mouseover and mouseout transitions in the tooltips example will whet your appetite for more!

Events

The other technique is related to mouse 'events'. This describes the browser watching for when 'something' happens with the mouse on the screen and when it does, it takes a specified action. A (probably non-comprehensive) list of the types of events are the following;

- mousedown: Triggered by an element when a mouse button is pressed down over it
- mouseup: Triggered by an element when a mouse button is released over it
- mouseover: Triggered by an element when the mouse comes over it
- mouseout: Triggered by an element when the mouse goes out of it
- mousemove: Triggered by an element on every mouse move over it.
- click: Triggered by a mouse click: mousedown and then mouseup over an element
- contextmenu: Triggered by a right-button mouse click over an element.
- dblclick: Triggered by two clicks within a short time over an element

How many of these are valid to use within d3 I'm not sure, but I'm willing to bet that there are probably more than those here as well. Please go to <http://javascript.info/tutorial/mouse-events> for a far better description of the topic if required.

Get tipping

So, bolstered with a couple of new concepts to consider, let's see how they are enacted in practice.

If we start with our simple-scatter plot graph there are 4 areas in it that we will want to modify (it may be easier to check the tooltips.html file in the example files in the downloads section on d3noob.org).

The first area is the CSS. The following code should be added just before the `</style>` tag;

```
div.tooltip {
  position: absolute;
  text-align: center;
  width: 60px;
  height: 28px;
  padding: 2px;
  font: 12px sans-serif;
  background: lightsteelblue;
  border: 0px;
  border-radius: 8px;
  pointer-events: none;
}
```

These styles are defining how our tooltip will appear . Most of them are fairly straight forward. The position of the tooltip is done in absolute measurements, not relative. The text is centre aligned, the height, width and colour of the rectangle is 28px, 60px and lightsteelblue respectively. The 'padding' is an interesting feature that provides a neat way to grow a shape by a fixed amount from a specified size.

We set the boarder to 0px so that it doesn't show up and a neat style (attribute?) called border-radius provides the nice rounded corners on the rectangle.

Lastly, but by no means least, the 'pointer-events: none' line is in place to instruct the mouse event to go "through" the element and target whatever is "underneath" that element instead (Read more here; <https://developer.mozilla.org/en-US/docs/CSS/pointer-events>). That means that even if the tooltip partly obscures the circle, the code will stll act as if the mouse is over only the circle.

The second addition is a simple one-liner that should (for forms sake) be placed under the 'parseData' variable declaration;

```
var formatTime = d3.time.format("%e %B");
```

This line formats the date when it appears in our tooltip. Without it, the time would default to a disturbingly long combination of temporal details. In the case here we have declared that we want to see the day of the month (%e) and the full month name(%B).

The third block of code is the function declaration for 'div'.

```
var div = d3.select("body").append("div")
  .attr("class", "tooltip")
  .style("opacity", 0);
```

We can place that just after the 'valueline' definition in the JavaScript. Again there's not too much here that's surprising. We tell it to attach 'div' to the body element, we set the class to the tooltip class (from the CSS) and we set the opacity to zero. It might sound strange to have the opacity set to zero, but remember, that's the natural state of a tooltip. It will live unseen until it's moment of revelation arrives and it pops up!

The final block of code is slightly more complex and could be described as a mutant version of the neat little bit of code that we used to do the drawing of the dots for the scatter plot. That's because the tooltips are all about the scatter plot circles. Without a circle to 'mouseover', the tooltip never appears :-).

So here's the code that includes the scatter plot drawing (it's included since it's pretty much integral);

```

svg.selectAll("dot")
  .data(data)
  .enter().append("circle")
    .attr("r", 5)
    .attr("cx", function(d) { return x(d.date); })
    .attr("cy", function(d) { return y(d.close); })
    .on("mouseover", function(d) {
      div.transition()
        .duration(200)
        .style("opacity", .9);
      div
        .html(formatTime(d.date) + "<br/>" + d.close)
        .style("left", (d3.event.pageX) + "px")
        .style("top", (d3.event.pageY - 28) + "px");
    })
    .on("mouseout", function(d) {
      div.transition()
        .duration(500)
        .style("opacity", 0);
    });

```

Before we start going through the code, the example file for tooltips that is on d3noob.org includes a brief series of comments for the lines that are added or changed from the scatter plot, so if you want to compare what is going on in context, that is an option.

The first six lines of the code are a repeat of the scatter plot drawing script. The only changes are that we've increased the radius of the circle from 3.5 to 5 (just to make it easier to mouse over the object) and we've removed the semicolon from the 'cy' attribute line since the code now has to carry on.

So the additions are broken into to areas that correspond to the two events. 'mouseover' and 'mouseout'. When the mouse moves over any of the circles in the scatter plot, the mouseover code is executed on the 'div' element. When the mouse is moved off the circle a different set of instructions are executed.

It would be a mistake to think of tooltips in the plural because there aren't a whole series of individual tooltips just waiting to appear for their specific circle. There is only one tooltip that will appear when the mouse moves over a circle. And depending on what circle it's over, the properties of the tooltip will alter slightly (in terms of its position and contents).

on.mouseover

The `.on("mouseover")` line initiates the introduction of the tooltip. Then we declare the element we will be introducing ('div') and that we will be applying a transition to it's introduction (`.transition()`). The next two lines describe the transition. It will take 200 milliseconds (`.duration(200)`) and will result in changing the elements opacity to .9 (`.style("opacity", .9);`). Given that the natural state of our tooltip is an opacity of 0, this make sense for something appearing, but it doesn't go all the way to a solid object and it retains a slight transparency just to make it look less permanent.

The following three lines format our tooltip. The first one adds an html element that contains our x and y information (the date and the d.close value). Now this is done in a slightly strange way. Other tooltips that I have seen have used a '.text' element instead of a '.html' one, but I have used '.html' in this case because I wanted to include the line break tag “
” to separate the date and value. I'm sure there are other ways to do it, but this worked for me. The other interesting part of this line is that this is where we call our time formatting function that we described earlier. The next two lines position the tooltip on the screen and to do this they grab the x and y coordinates of the mouse when the event takes place (with the d3.event.pageX and d3.event.pageY snippets) and apply a correction in the case of the y coordinate to raise the tooltip up by the same amount as its height (28 pixels).

on.mouseout

The `.on("mouseout"` section is slightly simpler in that it doesn't have to do any fancy text / html / coordinate stuff. All it has to do is to fade out the 'div' element. And that is done by simply reversing the opacity back to 0 and setting the duration for the transition to 500 milliseconds (being slightly longer than the fade-in makes it look slightly cooler IMHO).

Right, there you go. As a description it's ended up being a bit of a wall of text I'm afraid. But hopefully between the explanation and the example code you will get the idea. Please take the time to fiddle with the settings described here to find the ones that work for you and in the process you will reinforce some of the principles that help D3 do it's thing.

I've placed a copy of the file for drawing the tooltips into the downloads section on d3noob.org with the general examples as tooltips.html.

What are the predefined, named colours?

Throughout this document I have been using colours defined by name. This is mainly because I can and not for any other reason. In fact there several different ways to define colours used in D3 / JavaScript / CSS and HTML. I have no idea what the limitations for use are and / or how their use in different browsers impacts on correct representation. But I do know that they're used widely.

But, I was really interested in what the names were for the colours. After a cursory search I was able to find the following list on about.com

(http://webdesign.about.com/od/colorcharts/1/bl_namedcolors.htm).

The overriding point of all this is that there's more than one way to define colours in your graphs.

It means that

```
.style("fill", "steelblue")
```

and...


























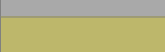




```
.style("fill", "#4682b4")
```

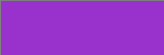











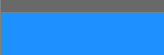




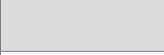













and...

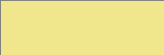





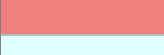
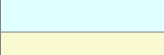




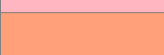















```
.style("fill", "rgb(70,130,180)")
```

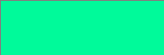




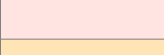









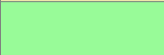
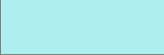














All result in the same colour being applied.





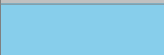
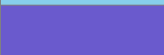








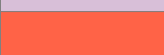



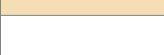
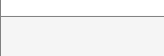


Now, some may (justifiably) question the wisdom of putting just over 5 pages of colours into a document. I really like them and I like to be able to refer to them easily. This works for me. I'm sure it works for some others as well :-).

Colour	Colour Name	Hexadecimal	RGB
	aliceblue	#f0f8ff	240,248,255
	antiquewhite	#faebd7	250,235,215
	aqua	#00ffff	0,255,255
	aquamarine	#7fffd4	127,255,212
	azure	#f0ffff	240,255,255
	beige	#f5f5dc	245,245,220
	bisque	#ffe4c4	255,228,196
	black	#000000	0,0,0
	blanchedalmond	#ffebcd	255,235,205
	blue	#0000ff	0,0,255
	blueviolet	#8a2be2	138,43,226
	brown	#a52a2a	165,42,42
	burlywood	#deb887	222,184,135
	cadetblue	#5f9ea0	95,158,160
	chartreuse	#7fff00	127,255,0
	chocolate	#d2691e	210,105,30
	coral	#ff7f50	255,127,80
	cornflowerblue	#6495ed	100,149,237
	cornsilk	#fff8dc	255,248,220
	crimson	#dc143c	220,20,60
	cyan	#00ffff	0,255,255
	darkblue	#00008b	0,0,139
	darkcyan	#008b8b	0,139,139
	darkgoldenrod	#b8860b	184,134,11
	darkgray	#a9a9a9	169,169,169
	darkgreen	#006400	0,100,0
	darkgrey	#a9a9a9	169,169,169
	darkkhaki	#bdb76b	189,183,107
	darkmagenta	#8b008b	139,0,139
	darkolivegreen	#556b2f	85,107,47
	darkorange	#ff8c00	255,140,0

Colour	Colour Name	Hexadecimal	RGB
	darkorchid	#9932cc	153,50,204
	darkred	#8b0000	139,0,0
	darksalmon	#e9967a	233,150,122
	darkseagreen	#8fbc8f	143,188,143
	darkslateblue	#483d8b	72,61,139
	darkslategray	#2f4f4f	47,79,79
	darkslategrey	#2f4f4f	47,79,79
	darkturquoise	#00ced1	0,206,209
	darkviolet	#9400d3	148,0,211
	deeppink	#ff1493	255,20,147
	deepskyblue	#00bfff	0,191,255
	dimgray	#696969	105,105,105
	dimgrey	#696969	105,105,105
	dodgerblue	#1e90ff	30,144,255
	firebrick	#b22222	178,34,34
	floralwhite	#fffaf0	255,250,240
	forestgreen	#228b22	34,139,34
	fuchsia	#ff00ff	255,0,255
	gainsboro	#dcdcdc	220,220,220
	ghostwhite	#f8f8ff	248,248,255
	gold	#ffd700	255,215,0
	goldenrod	#daa520	218,165,32
	gray	#808080	128,128,128
	green	#008000	0,128,0
	greenyellow	#adff2f	173,255,47
	grey	#808080	128,128,128
	honeydew	#f0ffff	240,255,240
	hotpink	#ff69b4	255,105,180
	indianred	#cd5c5c	205,92,92
	indigo	#4b0082	75,0,130
	ivory	#fffff0	255,255,240

Colour	Colour Name	Hexadecimal	RGB
	khaki	#f0e68c	240,230,140
	lavender	#e6e6fa	230,230,250
	lavenderblush	#fff0f5	255,240,245
	lawngreen	#7cfc00	124,252,0
	lemonchiffon	#ffffcd	255,250,205
	lightblue	#add8e6	173,216,230
	lightcoral	#f08080	240,128,128
	lightcyan	#e0ffff	224,255,255
	lightgoldenrodyellow	#fafad2	250,250,210
	lightgray	#d3d3d3	211,211,211
	lightgreen	#90ee90	144,238,144
	lightgrey	#d3d3d3	211,211,211
	lightpink	#ffb6c1	255,182,193
	lightsalmon	#ffa07a	255,160,122
	lightseagreen	#20b2aa	32,178,170
	lightskyblue	#87cefa	135,206,250
	lightslategray	#778899	119,136,153
	lightslategrey	#778899	119,136,153
	lightsteelblue	#b0c4de	176,196,222
	lightyellow	#ffffe0	255,255,224
	lime	#00ff00	0,255,0
	limegreen	#32cd32	50,205,50
	linen	#faf0e6	250,240,230
	magenta	#ff00ff	255,0,255
	maroon	#800000	128,0,0
	mediumaquamarine	#66cdaa	102,205,170
	mediumblue	#0000cd	0,0,205
	mediumorchid	#ba55d3	186,85,211
	mediumpurple	#9370db	147,112,219
	mediumseagreen	#3cb371	60,179,113
	mediumslateblue	#7b68ee	123,104,238

Colour	Colour Name	Hexadecimal	RGB
	mediumspringgreen	#00fa9a	0,250,154
	mediumturquoise	#48d1cc	72,209,204
	mediumvioletred	#c71585	199,21,133
	midnightblue	#191970	25,25,112
	mintcream	#f5fffa	245,255,250
	mistyrose	#ffe4e1	255,228,225
	moccasin	#ffe4b5	255,228,181
	navajowhite	#ffdead	255,222,173
	navy	#000080	0,0,128
	oldlace	#fdf5e6	253,245,230
	olive	#808000	128,128,0
	olivedrab	#6b8e23	107,142,35
	orange	#ffa500	255,165,0
	orangered	#ff4500	255,69,0
	orchid	#da70d6	218,112,214
	palegoldenrod	#eee8aa	238,232,170
	palegreen	#98fb98	152,251,152
	paleturquoise	#afeeee	175,238,238
	palevioletred	#db7093	219,112,147
	papayawhip	#ffefd5	255,239,213
	peachpuff	#ffdab9	255,218,185
	peru	#cd853f	205,133,63
	pink	#ffc0cb	255,192,203
	plum	#dda0dd	221,160,221
	powderblue	#b0e0e6	176,224,230
	purple	#800080	128,0,128
	red	#ff0000	255,0,0
	rosybrown	#bc8f8f	188,143,143
	royalblue	#4169e1	65,105,225
	saddlebrown	#8b4513	139,69,19
	salmon	#fa8072	250,128,114

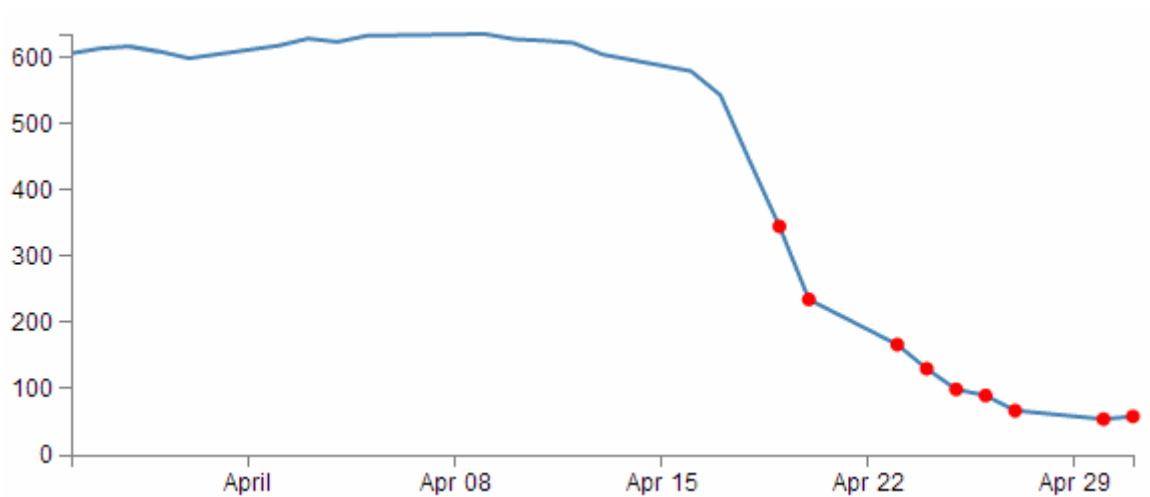
Colour	Colour Name	Hexadecimal	RGB
	sandybrown	#f4a460	244,164,96
	seagreen	#2e8b57	46,139,87
	seashell	#fff5ee	255,245,238
	sienna	#a0522d	160,82,45
	silver	#c0c0c0	192,192,192
	skyblue	#87ceeb	135,206,235
	slateblue	#6a5acd	106,90,205
	slategray	#708090	112,128,144
	slategrey	#708090	112,128,144
	snow	#fffafa	255,250,250
	springgreen	#00ff7f	0,255,127
	steelblue	#4682b4	70,130,180
	tan	#d2b48c	210,180,140
	teal	#008080	0,128,128
	thistle	#d8bfd8	216,191,216
	tomato	#ff6347	255,99,71
	turquoise	#40e0d0	64,224,208
	violet	#ee82ee	238,130,238
	wheat	#f5deb3	245,222,179
	white	#ffffff	255,255,255
	whitesmoke	#f5f5f5	245,245,245
	yellow	#ffff00	255,255,0
	yellowgreen	#9acd32	154,205,50

Selecting / filtering a subset of objects

OK, Imagine a scenario where you want to select (or should we say filter) a particular range of objects from a larger set for display in some way.

For example, what if we wanted to use our scatter plot example to show the line as normal, but we are particularly interested in the points where the values of the points fall below 400. And when it does we want them highlighted with a circle as we have done with all the point previously.

So that we end up with something that looks a little like this...



Err... Yes, for those among you who are of the observant persuasion, I have deliberately coloured them red as well (red for DANGER!).

This is a fairly simple example, but serves to illustrate the principle adequately.

From our simple scatter plot example we only need to add in two lines to the block of code that draws the circles as follows;

```
svg.selectAll("dot")
  .data(data)
  .enter().append("circle")
  .filter(function(d) { return d.close < 400 }) // <== This line
  .style("fill", "red") // <== and this one
  .attr("r", 3.5)
  .attr("cx", function(d) { return x(d.date); })
  .attr("cy", function(d) { return y(d.close); });
```

The first added line uses the '.filter' function to act on the data points and according to the arguments passed to it in this case, only return those where the value of d.close is less than 400 (**return** d.close < 400).

The second added line is our line that simply colours the circles red (.style("fill", "red")).

That's all there is to it. Pretty simple, but the filter function can be very powerful when used wisely.

I've placed a copy of the file for selecting / filtering into the downloads section on d3noob.org with the general examples as filter-selection.html.

Select items with an IF statement.

The filtering – selection section above is a good way to adapt what you see on a graph, but so is a more familiar friend the 'if' statement.

An if statement will act to carry out a task in a particular way dependant on a condition that you specify.

Here's an example, what if we wanted to show our scatter plot as normal, but all those with a 'close' value less than 400 should be coloured red. Sound familiar? Yes, I know it's similar to the example above, with the subtle difference that it is leaving the circles above 400 in place (more on that to follow).

So, starting with the simple scatter plot example all we have to do is include the if statement in the block of code that draws the circles. Here's the entire block with the additions highlighted;

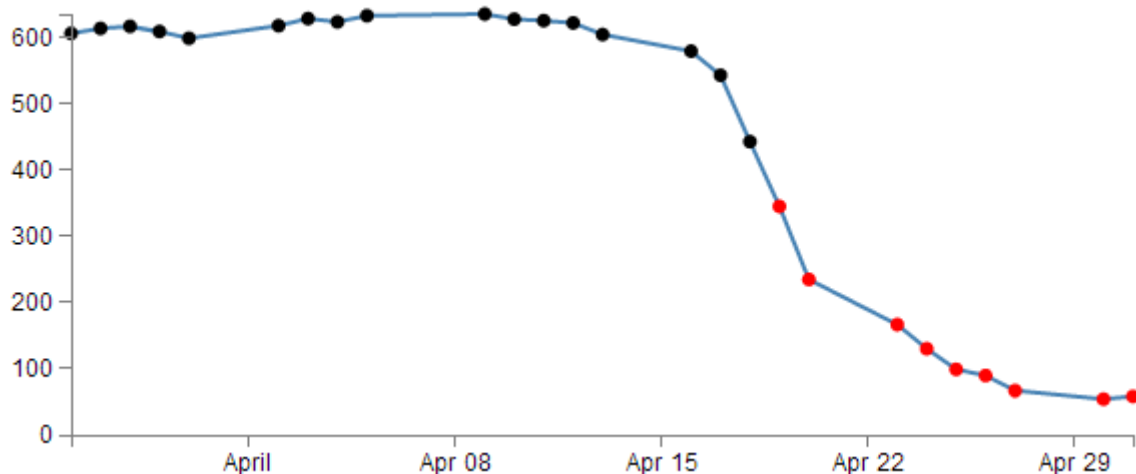
```
svg.selectAll("dot")
  .data(data)
  .enter().append("circle")
  .attr("r", 3.5)
  .style("fill", function(d) { // <== Add these
    if (d.close <= 400) {return "red"} // <== Add these
    else { return "black" } // <== Add these
  }) // <== Add these
  .attr("cx", function(d) { return x(d.date); })
  .attr("cy", function(d) { return y(d.close); });
```

Our first added line introduces the style modifier and the rest of the code acts to provide a return for the 'fill' attribute.

The second line introduces our if statement. They're all pretty easy to follow between languages. Just look out for maintaining the correct syntax and you should be fine. In this case we're asking if the value of d.close is less than or equal to 400 and if it is it will return the "red" statement for our fill.

The third line covers our rear and make sure that if the colour isn't going to be red, it's going to be black. The last line just closes the style and function statements.

The result?



Aww..... nice.

I've placed a copy of the file that uses the if statement into the downloads section on d3noob.org with the general examples as if-statement.html.

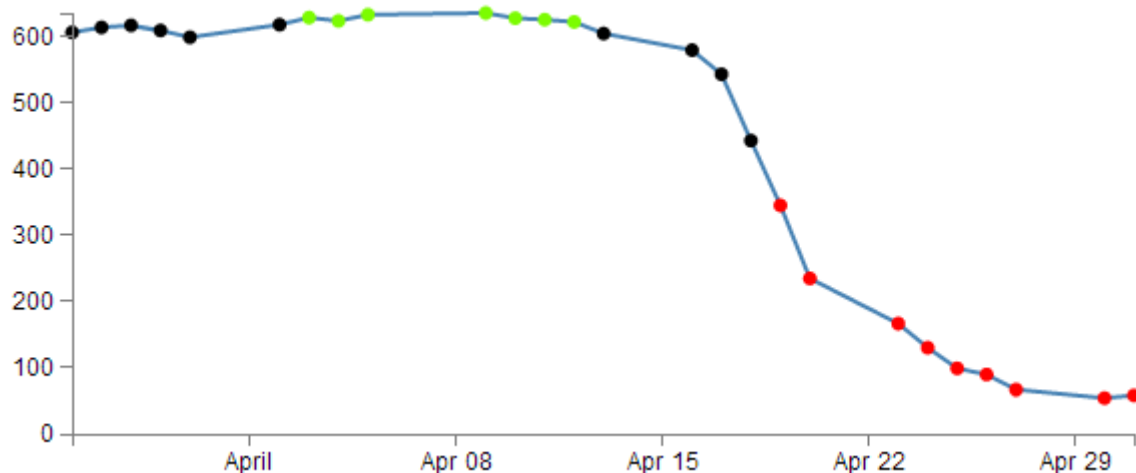
Could it be any cooler? I'm glad you asked.

What if we wanted to have all the points where close was less than 400 red and all those where close was greater than 620 green? Oh yeah! Now we're talking.

So with one small change to the if statement;


```
.style("fill", function(d) {
    if (d.close <= 400) {return "red"}
    else if (d.close >= 620) {return "lawngreen"} // <== Right here
    else { return "black" }
;})
```

Check it out...



Nice.

Applying a colour gradient to a line based on value.

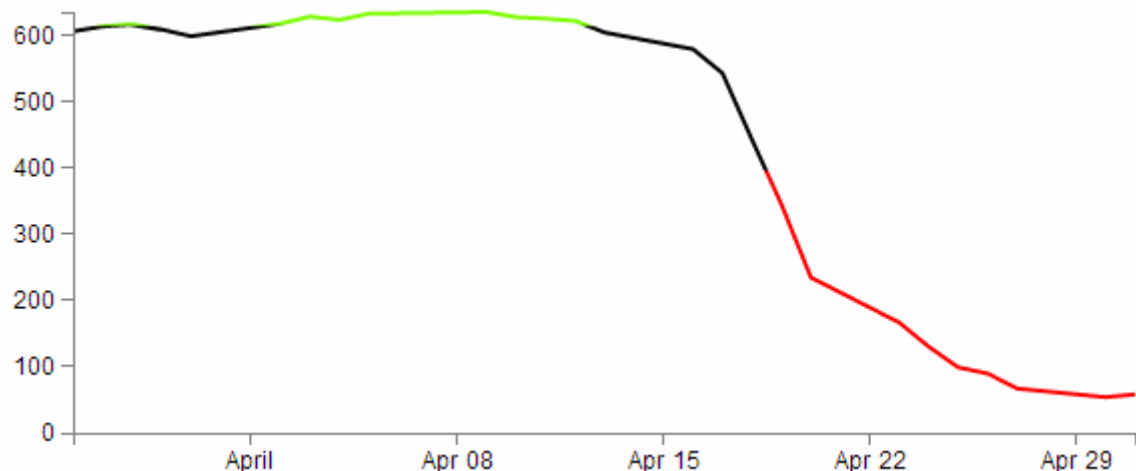
I just know that you were impressed with the changing dots in a scatter plot based on the value. But could we could one better?

How about we try to reproduce the same effect but by varying the colour of the plotted line.

This is a neat feature and a useful example of the flexibility of d3.js and svg in general.

I used the appropriate bits of code from Mike Bostock's Threshold Encoding example here <http://bl.ocks.org/3970883>. And I should take the opportunity to heartily recommend browsing through his collection of examples on [bl.ocks.org](http://bl.ocks.org/mbostock) (<http://bl.ocks.org/mbostock>).

Here then is a plotted line that is red below 400, green above 620 and black in between.



How cool is that?

Enough beating around the bush, how is the magic line produced?

Well starting with our simple line graph there are only two blocks of code to go in. One is CSS in the `<style>` area and the second is a tricky little piece of code that deals with gradients.

So, first the CSS.

```
.line {  
  fill: none;  
  stroke: url(#line-gradient);  
  stroke-width: 2px;  
}
```

This block can go in the `<style>` area towards the end.

There's the fairly standard fill of none and a stroke width of 2 pixels, but the stroke: `url(#line-gradient);` is something different.

In this case the stroke (which we remember is the colour of the line) is being set at a link within the page which is set by the anchor `#line-gradient`. As we will see shortly this is in our second block of code, so the colour is being defined in a separate portion of the script.

And now the JavaScript Gradient code;

```
svg.append("linearGradient")  
  .attr("id", "line-gradient")  
  .attr("gradientUnits", "userSpaceOnUse")  
  .attr("x1", 0).attr("y1", y(0))  
  .attr("x2", 0).attr("y2", y(1000))  
  .selectAll("stop")  
  .data([  
    {offset: "0%", color: "red"},  
    {offset: "40%", color: "red"},  
    {offset: "40%", color: "black"},  
    {offset: "62%", color: "black"},  
    {offset: "62%", color: "lawngreen"},  
    {offset: "100%", color: "lawngreen"}  
  ])  
  .enter().append("stop")  
  .attr("offset", function(d) { return d.offset; })  
  .attr("stop-color", function(d) { return d.color; });
```

There's our anchor on the second line!

But let's not get ahead of ourselves. This block should be placed after the x and y domains are set, but before the line is drawn.

Seems a bit strange doesn't it? This block is all about defining the actions of an element, but the element in this case is a gradient and the gradient acts on the line.

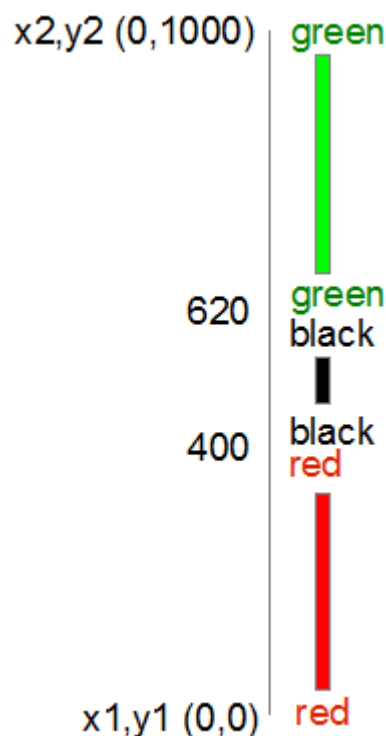
So, our first line adds our linear gradient. Gradients consist of continuously smooth colour transitions along a vector from one colour to another. We can have a linear one or a radial one and depending on which you select, there are a few options to define. There is some great information on gradients here <http://www.w3.org/TR/SVG/pservers.html> (more than I ever thought existed).

The second line (`.attr("id", "line-gradient")`) sets our anchor for the CSS that we saw earlier.

The third fourth and fifth lines define the bounds of the area over which the gradient will act. Since the coordinates x1, y1, x2, y2 will describe an area. The values for y1 (0) and y2 (1000) are used more for convenience to align with our data (which has a maximum value around 630 or so. For more information on the gradientUnits attribute I found this page useful <https://developer.mozilla.org/en-US/docs/SVG/Attribute/gradientUnits>. We'll come back to the coordinates in a moment.

The next block selects all the 'stop' elements for the gradients. These stop elements define where on the range covered by our coordinates the colours start and stop. These have to be defined as either percentages or numbers (where the numbers are really just percentages in disguise (i.e. 45% = 0.45)).

The best way to consider the stop elements is in conjunction with the gradientUnits. The image following may help.



In this case our coordinates describe a vertical line from 0 to 1000. Our colours transition from red (0) to red (400) at which point they change to black (400) and this will continue until it gets to black (620). Then this changes to green (620) and from there, any value above that will be green.

Now, it might seem a little convoluted to be doubling up on the colours and values, but the reason is that the gradient functions have a lot more to them than we're using and we'll have a look at the possibilities once the explanation of the code is done.

So after defining the stop elements, we enter and append the elements to the gradient (`.enter().append("stop")`)

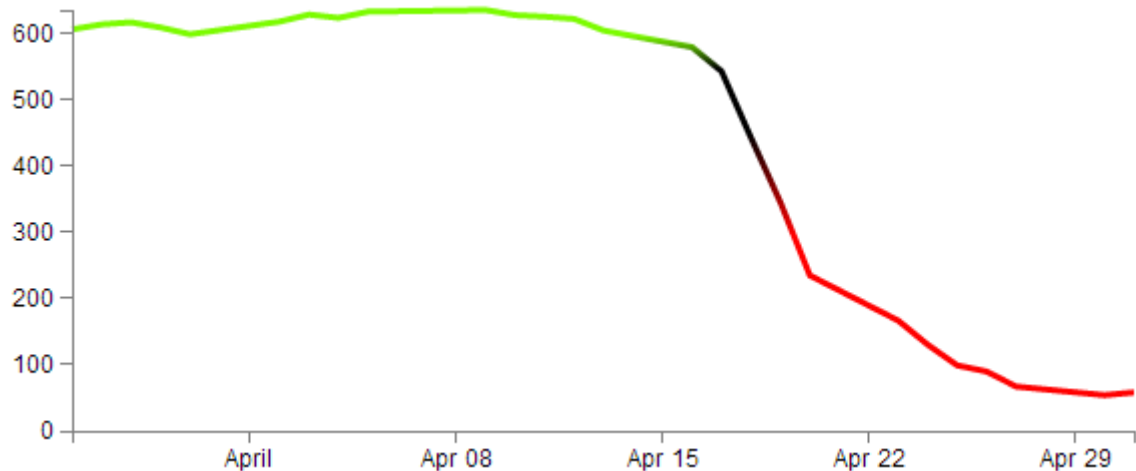
with attributes for offset and color that we defined in the stop elements area.

Now, the *IS* cool, but by now, I hope that you have picked that a gradient function really does mean a gradient, and not just a straight change from one colour to another.

So, let's try changing the stop element offsets to the following (and making the stroke-width slightly larger to see more clearly what's going on);

```
.data([
  {offset: "0%", color: "red"},
  {offset: "30%", color: "red"},
  {offset: "45%", color: "black"},
  {offset: "55%", color: "black"},
  {offset: "60%", color: "lawngreen"},
  {offset: "100%", color: "lawngreen"}
])
```

And here we go...



Ahh... A real gradient.

I have tended to find that I need to have a good think about how I set the offsets and bounds when doing this sort of thing since it can get quite complicated quite quickly :-)

Applying a colour gradient to an area fill.

The previous example of a varying gradient on a line is neat, but hopefully you're already thinking "Hang on, can't that same thing be applied to an area fill?".

Damn! You're catching on.

To do this there's only a few things we need to change;

First of all the CSS for the line needs to be amended to refer to the area. So this...

```
.line {
  fill: none;
  stroke: url(#line-gradient);
  stroke-width: 2px;
}
```

...gets changed to this...

```
.area {
  fill: url(#area-gradient);
  stroke-width: 0px;
}
```

We've defined the styles for the area this time, but instead of the stroke being defined by the separate script, now it's the area. While we've changed the url name, it's actually the same piece of code, with a different id (because it seemed wrong to be talking about an area when the label said line). We've also set the stroke width to zero, because we don't want any lines around our filled area.

Now we want to take the block of code that defined our line...

```
var    valueline = d3.svg.line()
      .x(function(d) { return x(d.date); })
      .y(function(d) { return y(d.close); });
```

... and we need to replace it with the standard block that defined an area fill.

```
var    area = d3.svg.area()
      .x(function(d) { return x(d.date); })
      .y0(height)
      .y1(function(d) { return y(d.close); });
```

So we're not going to be drawing a line at all. Just the area fill.

Next as I mentioned earlier, we change the id for the linearGradient block from "line-gradient" to "area-gradient"

```
.attr("id", "area-gradient")
```

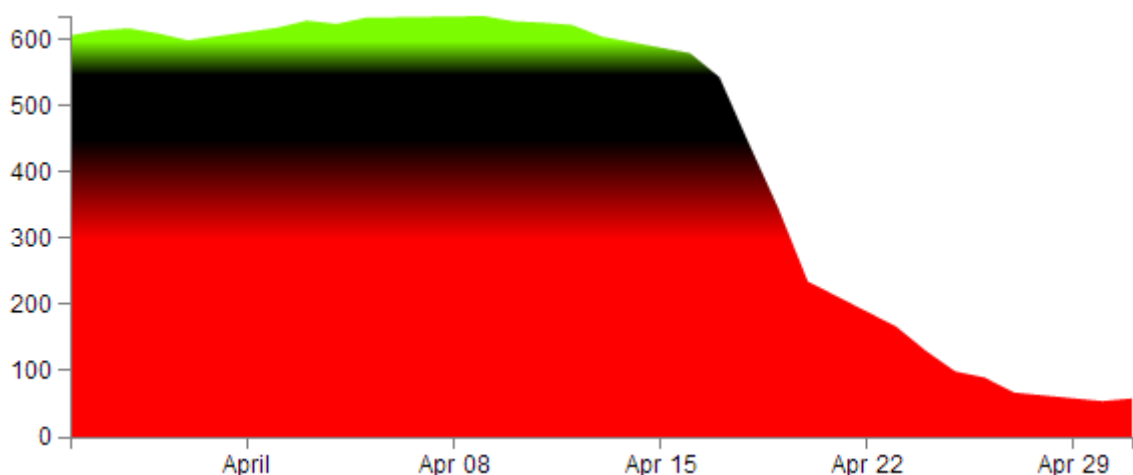
And lastly, we remove the block of code that drew the line and replace it with a block that draws an area. So change this....

```
svg.append("path")
  .attr("class", "line")
  .attr("d", valueline(data));
```

... for this;

```
svg.append("path")
  .datum(data)
  .attr("class", "area")
  .attr("d", area);
```

And then sit back and marvel at your creation;



For a slightly 'nicer' looking example, you could check out a variation of one of Mike Bostocks originals here; <http://bl.ocks.org/4433087>.

Using Plunker for development and hosting your D3 creations.

Recently Mike Bostock recommended 'Plunker' (<http://plnkr.co/>) as a tool for saving work online for collaboration and sharing. Although I had a quick look, I didn't quite 'get it' and although it looked like something that I should be interested in, I (foolishly) moved on to other things.

Quite frankly I should have persevered.

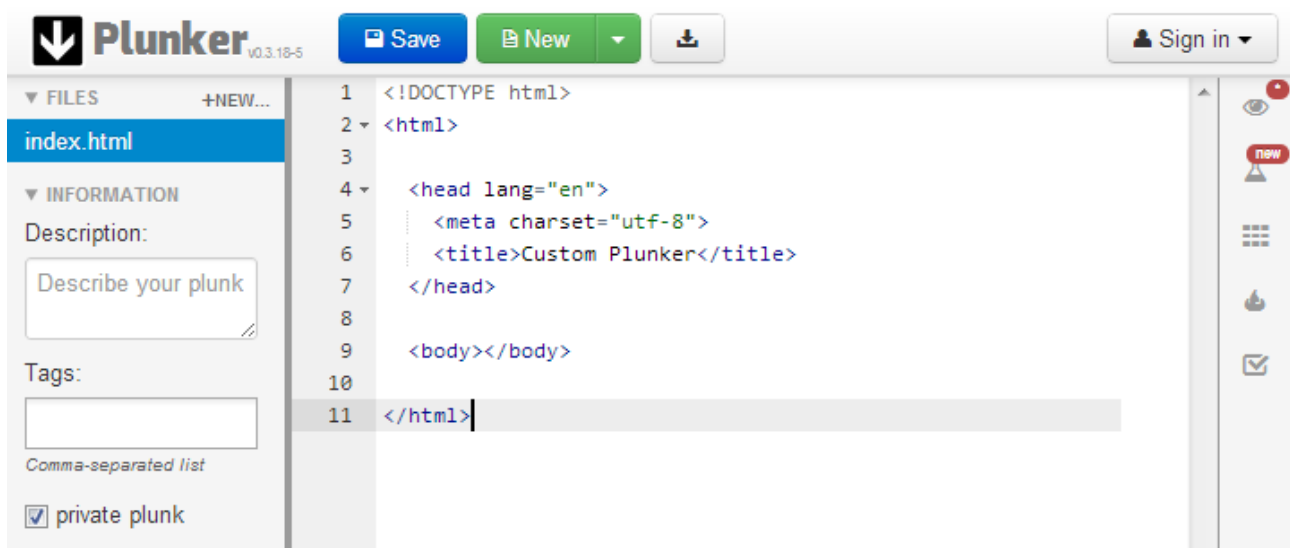
Plunker is awesome.

So what can it do for you?

Well, in short, this gives you a place to put your graphs on the web without the hassle of needing a web server as well as allowing others to view and collaborate! There are some limitations to hosting graphs in this environment, but there's no denying that for ease of use and visibility to the outside world, it's outstanding!

Time for an example. I'll try to go through the process of implementing the simple graph example on Plunker.

So it's as simple as going to <http://plnkr.co/edit/>



What you're seeing here is an area where you can place your entire HTML code. So let's replace the 11 lines of the place holder code with the simple graph example (just copy and paste it in there over the top of the current 11 lines);

Now, there are two important things we have to do before it will work.

1. We need to tell the script where to find d3.js
2. We need to make our data accessible

Helping the script find d3.js is nice and easy. Just replace this line in your plunk;

```
<script type="text/javascript" src="d3/d3.v3.js"></script>
```

...with this line...

```
<script src="http://d3js.org/d3.v3.min.js"></script>
```

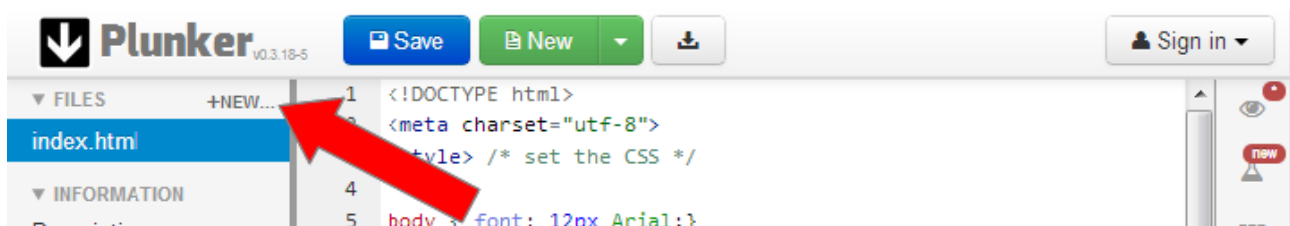
That will allow your plunk to use the version of d3.js that is hosted on d3js.org (it uses the minimised version (which is why it has the 'min' in it), but never fear, it's still d3, just distilled to enhance the flavour :-)).

Making our data available is only slightly more difficult.

In experimenting with Plunker, I found that there appears to be something 'odd' about accessing the tab separated values that we have been using thus far (in the data.tsv file), however, D3 to the rescue! We can simply use Comma Separated Values (csv) instead.

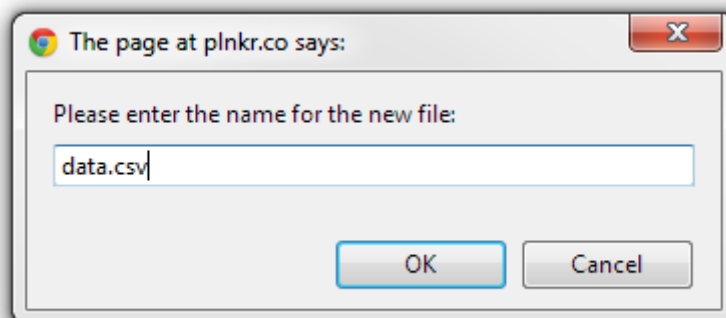
So in preparation for this exercise, please edit your data.tsv file to have the tabs separating the values replaced by commas and rename it data.csv.

We will host our data.csv file on plunker as well and there is built in functionality to let us do it.



In the top left hand corner, beside the 'FILES' note, there is a '+NEW...' section. Clicking on this will allow you to create another file that will exist with your plunk for its use, so let's do that.

This will open a dialogue box that will ask you to name your new file.



Enter the name data.csv.

Now another file has appeared under the 'Files' heading called data.csv. Click on it.



This now shows us a blank file called data.csv, so now open up your data.csv file in whatever editor you're using (I don't think a spreadsheet program is going to be a good idea since I doubt that it will maintain the information in a textual form as we're wanting it to do. So it's Geany for me). Copy the contents of your local data.csv file and past it into the new plunker data.csv file.

So now we have our data in there we need to tell our JavaScript where it is. So go back to the 'index.html' file (which is our simple graph code) and edit the line which finds the data.tsv file from this

```
d3.tsv("data/data.tsv", function(error, data) {
```

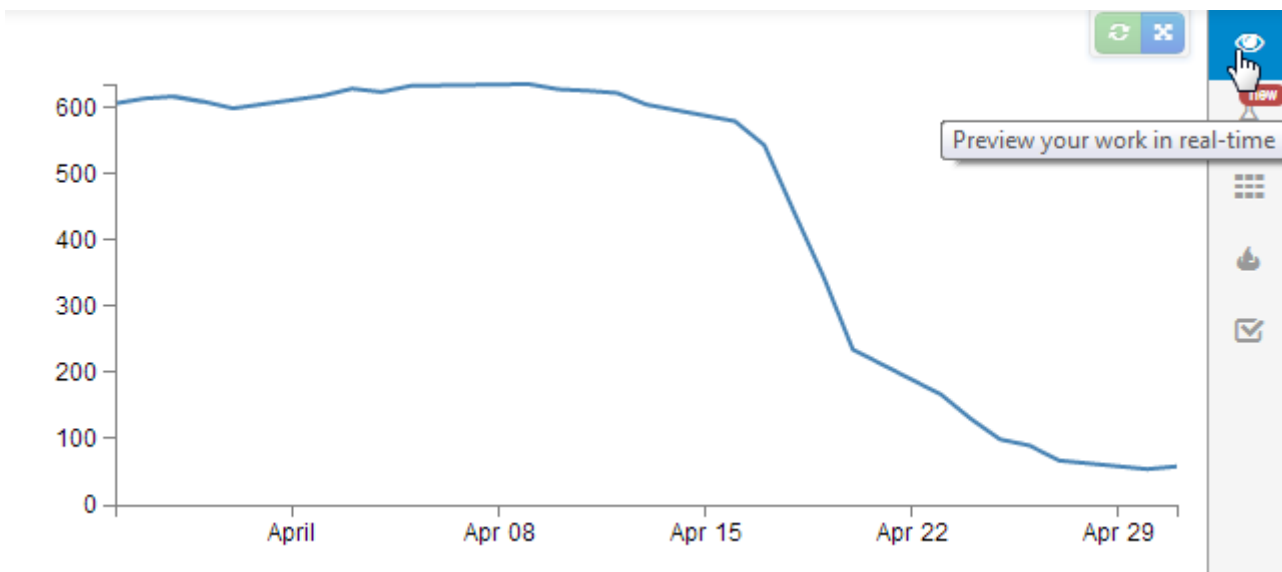
... to this ...

```
d3.csv("data.csv", function(error, data) {
```

Because we're using relative addressing, and plunker stores the files for the graphing script and the data side by side, we just removed the portion of the address that told our original code to look in the 'data' directory and told it to look in the current directory.

And that should be that!

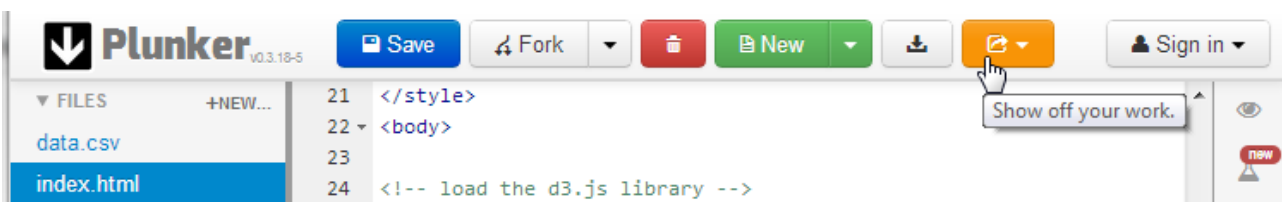
Now if you look on the right hand side of the screen, there is a little eye icon. If you click on it, it opens up a preview window of your file in action and viola!



If the graph doesn't appear, go through the steps outlined above and just check that the edits are made correctly. Unfortunately I haven't found a nice mechanism for troubleshooting inside Plunker yet (not like using F12 on Chrome).

But wait! There's more!

If you now click on the 'Save' button at the top of the screen, you will get some new button options. One of them is the orange one for showing off your work.



If you click on this, it will present you with several different options.

Share link:

Share preview:

Embed:

```

<iframe style="width: 100%; height: 300px"
src="http://embed.plnkr.co/QSCkG8Rf2qFgrCqq7Vfn"
" frameborder="0" allowfullscreen="allowfullscreen">
</iframe>

```

Tweet
0

Share
0

The first one is a link that will give others the option to collaborate on the script.

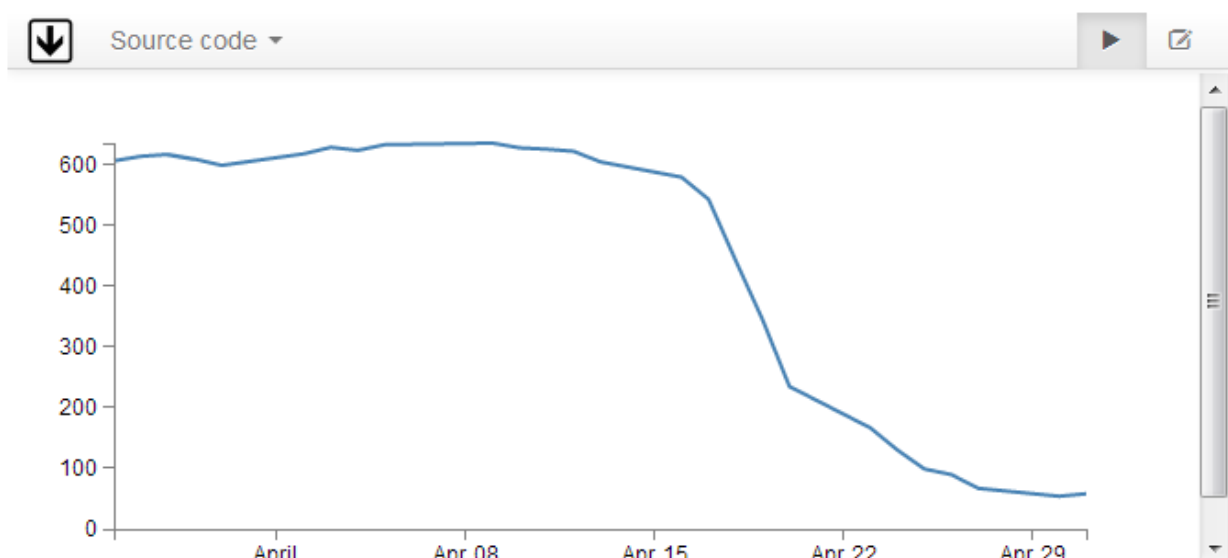
The second is a link that will allow others to preview the work;

<http://embed.plnkr.co/QSCkG8Rf2qFgrCqq7Vfn>

The last will allow you to embed your graph in a separate web page somewhere. Which I've tested with blogger and seems to work really well! (see image below).

Testing Plunker iframe insert

This is a test of inserting a Plunker iframe into a blogger post.



So, I'm impressed, Nice work by Plunker and it's creator Geoff Goodman.

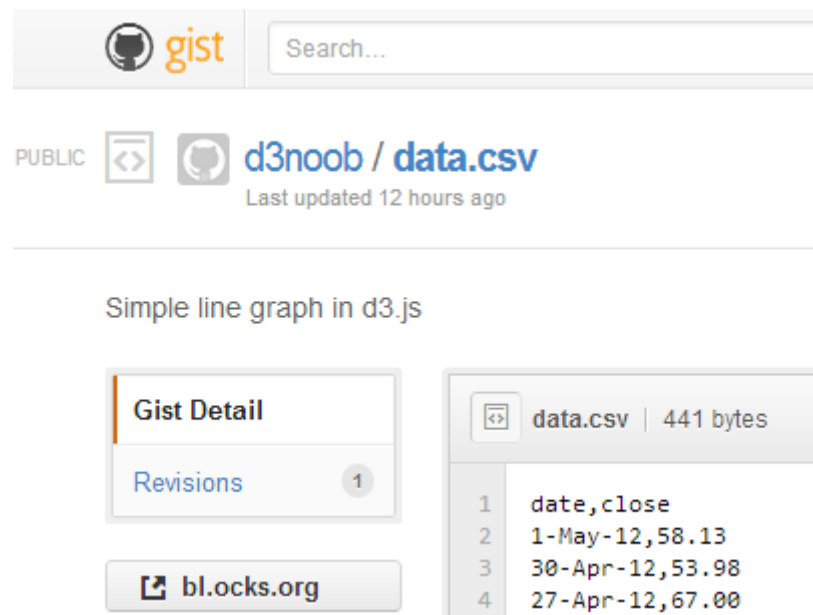
Loading a thumbnail into Gist for bl.ocks.org d3 graphs

This description will start on the assumption that the user already has a GitHub / Gist account set up and running. It's purpose is to demonstrate how to upload an image as a file named `thumbnail.png` to a Gist so that when viewing the users home page on bl.ocks.org you see a nice little preview of what a visitor can anticipate when they go to look at your work :-)

This description is a fleshed out version of the one provided by Christophe Viau on Google Groups (<https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc>).

Setting the scene:

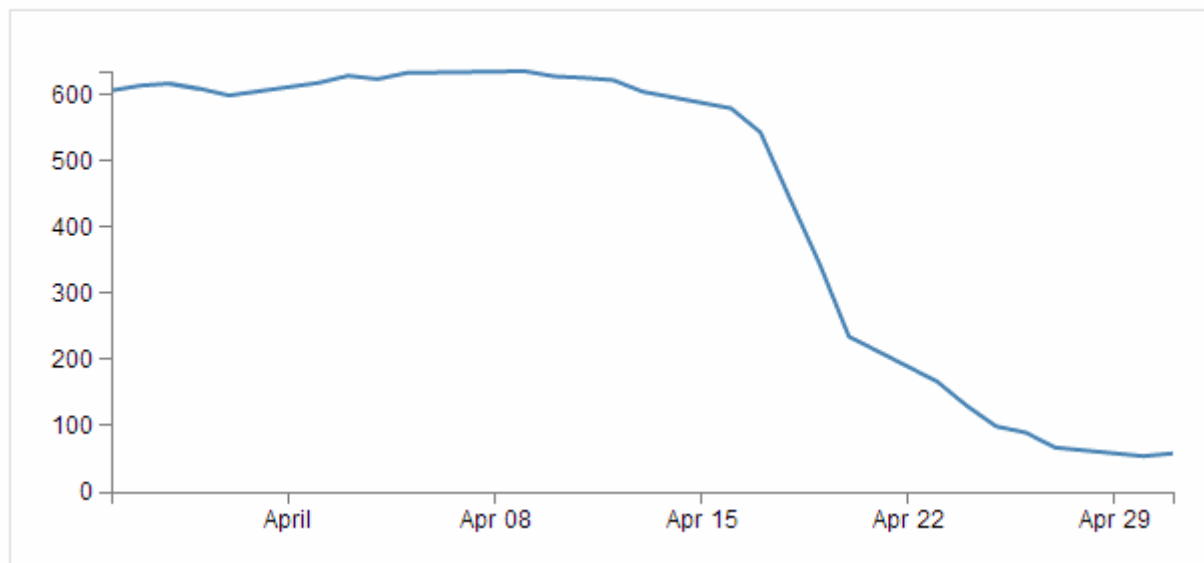
There you are a fresh faced d3.js user keen to share your work with the world. You set yourself up a GitHub / Gist account and put your code into a gist.



Your graph is a thing of rare beauty and the community needs to marvel at your brilliance. Of course this is a breeze with bl.ocks.org. Once you have all the code sorted and any data files accessible, bl.ocks.org can display the graph with the code and can even open the graph in its own window. The person responsible for bl.ocks.org? Mike Bostock of course (wherever does he get the time?).

So clicking on the bl.ocks.org button on the gist page (load the extension available from the main page of bl.ocks.org) takes you to see your graph.

d3noob's block #4414436



Wow! Impressive.

So you think that will make a fine addition to your collection of awesome graphs and if you click on your GitHub user name that is in the top left of the screen you go to a page that lays out all your graphs with a thumbnail giving a sneak preview of what the user can expect.

about [blocks.org](#)

d3noob's blocks

Simple line graph in d3.js

December 31, 2012

Aww... Rats! There's a nice place holder, but no pretty picture.

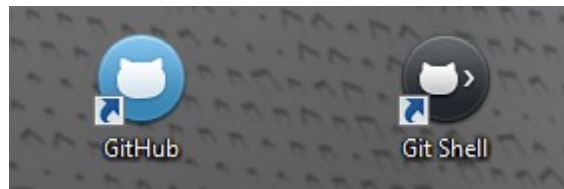
Hang on, what had Mike said on the [blocks.org](#) main page?

“The main source code for your example should be named `index.html`. You can also include a `README.md` using [Markdown](#), and a `thumbnail.png` for preview.”

Ahh.. you need to include a `thumbnail.png` file in your Gist!

So how to get it there? Well Gist is a repository, so what you need to do is to put the code in there somehow. Now from the Gist web page this doesn't appear to be a nice (gui) way to do this. So from here you will need to suspend your noob status and hit the command line.

The good news (if you're a windows user (and sorry, I haven't done this in Linux or on a Mac)) is that as part of the GitHub for windows installation a command line tool was installed as well! So you're going to use the Git Shell.

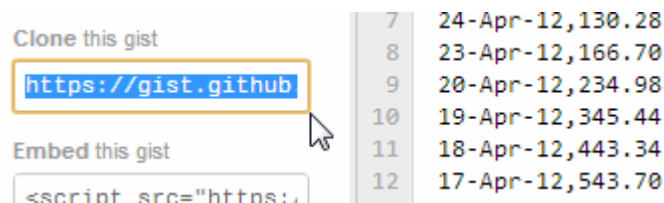


Enough of the scene setting. Let's git going :-).

Right, I'm going to describe the steps here in a pretty verbose fashion with pretty pictures and everything, but at the end I will put a simple set of steps in the form that Christophe Viau outlined on Google Groups²⁰.

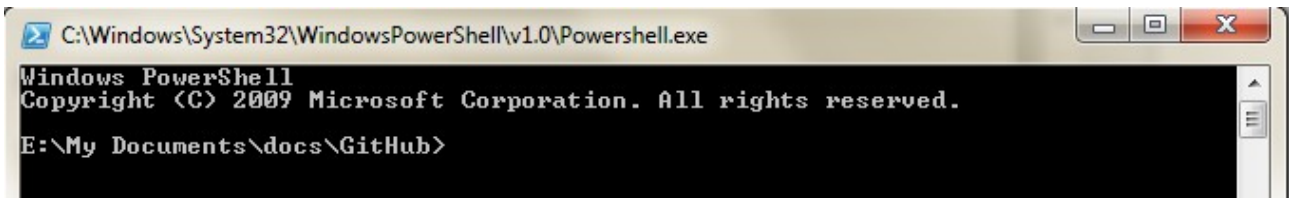
First you will want to have your image ready. It needs to be a png with dimensions of 230 x 120 pixels. It should also be less than 50kB in size.

Now go to your public Gist that you have already set up and copy the link in the “Clone this gist” box.



(this should look something like <https://gist.github.com/4414436.git>)

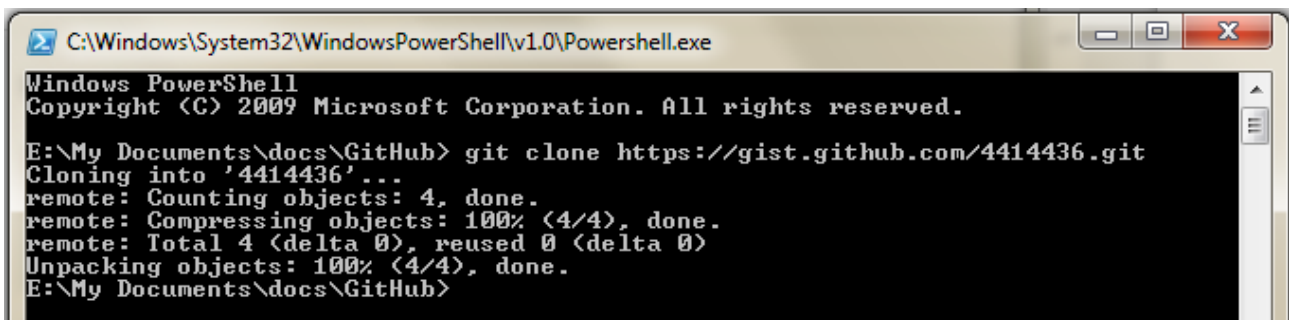
Now you're going to clone this gist to a local repository using the Git Shell. Open it up from the desktop icon and you should see something like the following;



So you can clone the gist to a local folder with the command;

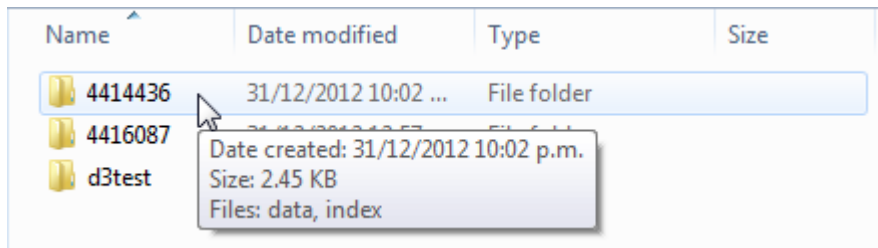
```
git clone https://gist.github.com/4414436.git
```

(The url is the one copied from the 'Clone this gist' box.)



²⁰ <https://groups.google.com/forum/?fromgroups=#!topic/d3-js/FBosXiTB9Pc>

This will create a folder with the id (the number) of the gist in your local GitHub working directory.



And there it is (Ooo... Look almost New Years!).

Copy your thumbnail.png file into this directory.

Back to the Git Shell and change into the directory (4414436) . We can now add the thumbnail.png file to the gist with the command;

```
git add thumbnail.png
```

```
E:\My Documents\docs\GitHub> cd 4414436
E:\My Documents\docs\GitHub\4414436 [master +1 ~0 -0 !]> git add thumbnail.png
E:\My Documents\docs\GitHub\4414436 [master +1 ~0 -0]> _
```

And now commit it to your gist with the following command in the Git Shell;

```
git commit -m "Thumbnail image added"
```

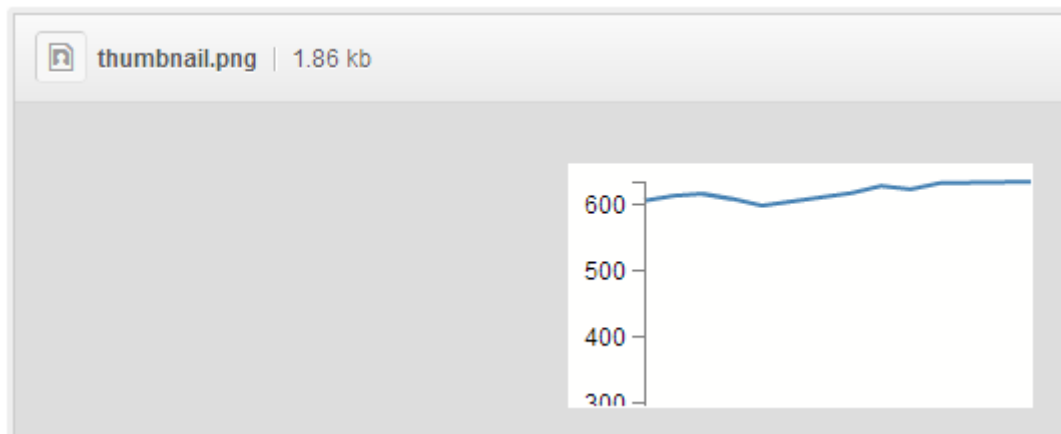
```
E:\My Documents\docs\GitHub\4414436 [master +1 ~0 -0]> git commit -m "Thumbnail
image added"
[master f1308b5] Thumbnail image added
1 file changed, 0 insertions(+), 0 deletions(-)
create mode 100644 thumbnail.png
E:\My Documents\docs\GitHub\4414436 [master]> _
```

Now we need to push the commit to the remote gist (you may be asked for your GitHub user name and password if you haven't done this before) with the following command;

```
git push
```

```
E:\My Documents\docs\GitHub\4414436 [master]> git push
Counting objects: 4, done.
Delta compression using up to 2 threads.
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 1.95 KiB, done.
Total 3 (delta 0), reused 0 (delta 0)
To https://gist.github.com/4414436.git
8128d45..f1308b5 master -> master
E:\My Documents\docs\GitHub\4414436 [master]>
```

OK, now you can go back to the web page for your gist and refresh it and scroll on down...



Woo Hoo!

(I know it doesn't look like much, but this is a VERY simple graph :-)).

Now for the real test. Go back to your home page for your blocks on bl.ocks.org and refresh the page.

d3noob's blocks



Oh yes. You may now bask in the sweet glow of victory. And as a little bit of extra fancy, if you move your mouse over the image it translates up slightly!

Wrap up.

The steps to get your thumbnail into the gist aren't exactly point and click, but the steps you need to take are fairly easy to follow. As promised, here is the abridged list of steps that will avoid you going through the several previous pages.

1. Create your public gist on <https://gist.github.com/>
2. Get an image ready (230 x 120 pixels, named thumbnail.png)
3. Under "Clone this gist", copy the link (i.e., <https://gist.github.com/4414436.git>)
4. If you have the command line git tools (Git Shell), clone this gist to a local folder:

```
git clone https://gist.github.com/4414436.git
```

It will add a folder with the gist id as a name (i.e., 4414436) under the current working directory.
5. Navigate to this folder via the command line in Git Shell: `cd 4414436` (dir 4414436 on windows)
6. Navigate to this folder in file explorer and add your image (i.e., thumbnail.png)
7. Add it to git from the command line: `git add test.png`

8. Commit it to git: `git commit -m "Thumbnail added"`
9. Push this commit to your remote gist (you may need your Github user name and password):
`git push`
10. Go back and refresh your Gist on <https://gist.github.com/> to confirm that it worked
11. Check your blocks home page and see if it's there too. <http://blocks.org/<yourusername>>

Just to finish off. A big thanks to Christophe Viau for the hard work on finding out how it all goes together and if there are any errors in the above description I have no doubt they will be mine.