

HTML TECHNOLOGY FUNDAMENTALS

Presentation and Information visualization

HTML BASIC KNOWLEDGE

A smooth but
complete introduction
to HTML, CSS and
Javascript

Mireia Ribera





Data Science Master
Degree

Intro	2
Tools.....	2
Text editor.....	2
A server	2
Your HTML page code	3
Technology fundamentals	3
The Web.....	3
HTML.....	5
Content Plus Structure.....	5
Adding Structure with Elements	6
Common Elements.....	7
Attributes	9
Classes and IDs.....	9
Comments.....	10
DOM.....	10
Developer Tools	10
Rendering and the Box Model	12
CSS	13
Selectors	14
Properties and Values.....	15
Comments.....	16
Referencing Styles.....	16
Inheritance, Cascading, and Specificity.....	17
Layout	18
JavaScript	20
Hello, Console	21
Variables	21
Functions.....	22
Comments.....	23
Referencing Scripts	23
Javascript and Altair.....	24

Intro

Nowadays the web is the preferred platform to publish and share all the content. The web offers access to a wide public and also interactivity, it is multi-platform and it could be seen in a smartphone or a desktop screen (although for graphs this is not always true or the price to pay is quite high).

Thus, visualizations, often are included in a web page. Many of the languages and corresponding tools seen in this course offer a web code to embed the visualization in your web page. For example Tableau public always includes a

share icon on the lower right corner    that offers the URL of the visualization, and an embedding code `<div class='tableauPlaceholder' ... </script>`, that you can include in any page or blog. Highcharts editors offer a download HTML option which gives you also a code to embed in your page  `<div id="highcharts-9b ...</script>`. And C3 explains you how to create the HTML page to include the code (<https://c3js.org/gettingstarted.html>). Altair directly creates an HTML page with the chart you have created.

In order to better understand how to integrate these charts in a web page and how to combine them in a layout, this tutorial explains the basics of HTML, CSS and javascript, and puts some examples with Altair. Of course, you may find plenty of resources in the web, as for example the ones in w3schools.com (<https://www.w3schools.com/>).

Tools

In order to edit your HTML code you will need a text editor. In order to publish your HTML page you will need a server.

Text editor

A text editor such as Notepad++, Sublime, Atom... will do. Some of them do more work and have autocomplete and highlight features, choose as you like. In any case, try to work always with UTF-8 encoding

Once you choose an editor, you can have a look at extensions and plugins. For example, I use Notepad++, with a javascript plugin (<http://www.sunjw.us/jstoolnpp/index.php>) and an XML plugin.

A server

As long as you don't include external javascript in your pages you will be able to test them directly in your browser, but, when you try to visualize pages that need an external library to work (c3, Altair), you will need to publish them in a server (a computer permanently connected to the Internet with a special program to serve web pages).

The good news are that you can simulate a server with Python. From the Python command shell you can start a Python server:

```
v.3
>python -m http.server
v.2
>python -m SimpleHTTPServer 8888 &.
```

Alternatively you can download a Node.js module that executes a web server:

```
>npm install-g http-server
And then, from any command-line use
>http-server
```

After executing the server either way on your browser you will be able to see your pages with <http://localhost:8080/> or <http://localhost:8888/> or the URL given when initiating the server.

Be aware that the server initiates on the folder from which you executed it, so you must navigate to your html files from there, and you will not be able to reach hierarchically superior folders.

Your HTML page code

In order to have all your code it is best to set good practices since the beginning. Therefore we will create a folder with four subfolders: *lib* for javascript external libraries, *css* for CSS stylesheets, *img* for image files, and *js* for our javascript files. It is also a recommendation to name your initial web page as *index.html*.

Technology fundamentals

This is a selection and an adaptation of Scott Murray “chapter 3”, Interactive Data Visualization for the web, 2nd ed. O’Reilly Media, 2017 ISBN 1491921289.

The Web

If you’re brand new to making web pages, you will now have to think about things that most people blissfully disregard every day, such as this: how does the web actually work?

We think of the web as a bunch of interlinked pages, but it’s really a collection of conversations between web servers and web clients (browsers).

The following scene is a dramatization of a typical such conversation that happens whenever you or anyone else clicks a link or types an address into your browser (meaning, this brief conversation is had about 88 zillion times every day):

*CLIENT: I’d really like to know what’s going on over at **somewebsite.com**. I better call over there to get the latest info. [Silent sound of internet connection being established.]*

*SERVER: Hello, unknown web client! I am the server hosting **somewebsite.com**. What page would you like?*

*CLIENT: This morning, I am interested in the page at **somewebsite.com/news/**.*

SERVER: Of course. One moment.

Code is transmitted from SERVER to CLIENT.

CLIENT: I have received it. Thank you!

SERVER: You’re welcome! Would love to stay on the line and chat, but I have other requests to process. Bye!

Clients contact servers with *requests*, and servers respond with data. But what is a server and what is a client?

Web servers are internet-connected computers running server software, so called because they *serve* web documents as requested. Servers are typically always on and always connected, but web developers often also run *local* servers, meaning they run on the same computer that you’re working on. *Local* means here; *remote* means somewhere else, on any computer but the one right in front of you.

There are lots of different server software packages, but Apache is the most common. Web server software is not pretty, and no one ever wants to look at it.

In contrast, web *browsers* can be very pretty, and we spend a lot of time looking at them. Most people recognize names like Firefox, Safari, Chrome, and Internet Explorer, all of which are browsers or *web clients*.

Every web page, in theory, can be identified by its URL (Uniform Resource Locator) or URI (Uniform Resource Identifier). Most people don't know what *URL* stands for, but they recognize one when they see it. By obsolete convention, URLs commonly begin with *www*, as in *http://www.calmingmanatee.com*, but with a properly configured server, the *www* part is wholly unnecessary.

Complete URLs consist of four parts:

- An indication of the *communication protocol*, such as HTTP or HTTPS
- The *domain name* of the resource, such as *calmingmanatee.com*
- The *port number*, indicating over which port the connection to the server should be attempted
- Any additional locating information, such as the path of the requested file, or any query parameters

A complete URL, then, might look like this: *http://alignedleft.com:80/dashboard/*.

Typically, the port number is excluded, as web browsers will try to connect over port 80 by default. So the preceding URL is functionally the same as *http://alignedleft.com/dashboard/*.

Note that the protocol is separated from the domain name by a colon and two forward (regular) slashes. Why two slashes? No reason. The inventor of the web regrets the error.

HTTP stands for Hypertext Transfer Protocol, and it's the most common protocol for transferring web content from server to client. The "S" on the end of HTTPS stands for *Secure*. HTTPS connections are used whenever information should be encrypted in transit, such as for online banking or ecommerce.

Let's briefly step through the process of what happens when a person goes to visit a website.

1. User runs the web browser of her choice, then types a URL into the address bar, such as *vega.github.io/vega-lite/tutorials/explore.html*. Because she did not specify a protocol, HTTPS is assumed, and "https://" is prepended to the URL.
2. The browser then attempts to connect to the server behind *vega.github.io* across the network, via port 80, the default port for HTTP.
3. The server associated with *vega.github.io* acknowledges the connection and is taking requests. ("I'll be here all night.")
4. The browser sends a request for the page that lives at */tutorials/* and is called *explore.html*.
5. The server sends back the HTML content for that page.
6. As the client browser receives the HTML, it discovers references to *other files* needed to assemble and display the entire page, including CSS stylesheets and image files. So it contacts the same server again, once per file, requesting the additional information.

7. The server responds, dispatching each file as needed.
8. Finally, all the web documents have been transferred over. Now the client performs its most arduous task, which is to *render* the content. It first parses through the HTML to understand the structure of the content. Then it reviews the CSS selectors, applying any properties to matched elements. Finally, it plugs in any image files and executes any JavaScript code.

Can you believe that all that happens every time you click a link? It's a lot more complicated than most people realize, but it's important to understand that client/server conversations are fundamental to the web.

HTML

Hypertext Markup Language is used to structure content for web browsers. HTML is stored in plain-text files with the *.html* suffix. A simple HTML document looks like this:

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
    <h1>Page Title</h1>
    <p>This is a really interesting paragraph.</p>
  </body>
</html>
```

HTML is a complex language with a rich history. This overview will address only the current iteration of HTML (formerly known as HTML5) and will touch on only what is immediately relevant for our practice with visualization.

Content Plus Structure

The core function of HTML is to enable you to “mark up” content, thereby giving it structure. Take, for example, this raw text:

```
Amazing Visualization Tool Cures All Ills A new open-source tool designed for
visualization of data turns out to have an unexpected, positive side effect:
it heals any ailments of the viewer. Leading scientists report that the tool,
called D3000, can cure even the following symptoms: fevers chills general
malaise It achieves this end with a patented, three-step process. Load in data.
Generate a visual representation. Activate magic healing function.
```

Reading between the lines, we can infer that this is a very exciting news story. But as unstructured content, it is very hard to read. By adding structure, we can differentiate between the headline, for example, and the body of the story.

Amazing Visualization Tool Cures All Ills

A new open-source tool designed for visualization of data turns out to have an unexpected, positive side effect: it heals any ailments of the viewer. Leading scientists report that the tool, called D3000, can cure even the following symptoms:

- *fevers*
- *chills*
- *general malaise*

It achieves this end with a patented, three-step process.

1. *Load in data.*
2. *Generate a visual representation.*
3. *Activate magic healing function.*

That has the same raw text content, but with a *visual structure* that makes the content more accessible.

HTML is a tool for specifying *semantic structure*, or attaching hierarchy, relationships, and *meaning* to content. (HTML doesn't address the visual representation of a document's structure—that's CSS's job.) Here is our story with each chunk of content replaced by a *semantic description* of what that content is.

Headline

Paragraph text

- *Unordered list item*
- *Unordered list item*
- *Unordered list item*

Paragraph text

1. *Numbered list item*
2. *Numbered list item*
3. *Numbered list item*

This is the kind of structure we specify with HTML markup.

Adding Structure with Elements

“Marking up” is the process of adding *tags* to create *elements*. HTML tags begin with < and end with >, as in <p>, which is the tag indicating a paragraph of text. Tags usually occur in pairs, in which case adding an opening and closing pair of tags creates a new *element* in the document structure.

Closing tags are indicated with a slash that closes or ends the element, as in </p>. Thus, a paragraph of text may be marked up like the following:

```
<p>This is a really interesting paragraph.</p>
```

Some elements can be *nested*. For example, here we use the `em` element to add *emphasis*.

```
<p>This is a <em>really</em> interesting paragraph.</p>
```

Nesting elements introduces hierarchy to the document. In this case, `em` is a child of `p` because it is contained by `p`. (Conversely, `p` is `em`'s parent.)

When elements are nested, they cannot overlap closures of their parent elements, as doing so would disrupt the hierarchy. For example:

```
<p>This could cause <em>unexpected</p>
<p>results</em>, and is best avoided.</p>
```

Some tags never occur in pairs, such as the `img` element, which references an image file. Although HTML no longer requires it, you will sometimes see such tags written in *self-closing* fashion, with a trailing slash before the closing angle bracket:

```

```

As of HTML5, the self-closing slash is optional, so the following code is equivalent to the preceding code:

```

```

Common Elements

There are hundreds of different HTML elements. Here are some of the most common. Later on we will see some, specific for layout. Reference the excellent [Mozilla Developer Network documentation](#) for a complete listing.

```
<!DOCTYPE html>
```

The standard document type declaration. Must be the first thing in the document.

```
html
```

Surrounds all HTML content in a document.

```
head
```

The document `head` contains all metadata about the document, such as its `title` and any references to external stylesheets and scripts.

```
title
```

The title of the document. Browsers typically display this at the top of the browser window and use this title when bookmarking a page.

```
body
```

Everything not in the `head` should go in the `body`. This is the primary visible content of the page.

```
h1, h2, h3, h4
```

These let you specify headings of different levels. `h1` is a top-level heading, `h2` is below that, and so on.

```
p
```

A paragraph!

```
ul, ol, li
```


Unordered lists are specified with `ul`, most often used for bulleted lists. Ordered lists (`ol`) are often numbered. Both `ul` and `ol` should include `li` elements to specify list items.

`em`

Indicates emphasis. Typically rendered in *italics*.

`strong`

Indicates additional emphasis. Typically rendered in **boldface**.

`a`

A link. Typically rendered as underlined, blue text, unless otherwise specified.

`span`

An arbitrary `span` of text, typically within a larger containing element like `p`.

`div`

An arbitrary *division* within the document. Used for grouping and containing related elements.

We could give our earlier example semantic structure by marking it up using some of these element's tags:

```
<h1>Amazing Visualization Tool Cures All Ills</h1>
<p>A new open-source tool designed for visualization of data turns out to have
an unexpected, positive side effect: it heals any ailments of the viewer.
Leading scientists report that the tool, called D3000, can cure even the
following symptoms:</p>
<ul>
  <li>fevers</li>
  <li>chills</li>
  <li>general malaise</li>
</ul>
<p>It achieves this end with a patented, three-step process.</p>
<ol>
  <li>Load in data.</li>
  <li>Generate a visual representation.</li>
  <li>Activate magic healing function.</li>
</ol>
```

When viewed in a web browser, that markup is rendered as shown in [Figure 3-1](#).

Amazing Visualization Tool Cures All Ills

A new open-source tool designed for visualization of data turns out to have an unexpected, positive side effect: it heals any ailments of the viewer. Leading scientists report that the tool, called D3000, can cure even the following symptoms:

- fevers
- chills
- general malaise

It achieves this end with a patented, three-step process.

1. Load in data.
2. Generate a visual representation.
3. Activate magic healing function.

Figure 1. Typical default rendering of simple HTML

Notice that we specified only the *semantic* structure of the content; we didn't specify any visual properties, such as color, type size, indents, or line spacing. Without such instructions, the browser falls back on its *default styles*, which, frankly, are not too exciting.

Attributes

We assign all HTML elements *attributes* by including property/value pairs in the opening tag.

```
<tagname property="value"></tagname>
```

The name of the property is followed by an equals sign, and the value is enclosed within double quotation marks.

Different kinds of elements can be assigned different attributes. For example, the `a` link tag can be given an `href` attribute, whose value specifies the URL for that link. (`href` is short for “hypertext reference.”)

```
<a href=" https://altair-viz.github.io/gallery/index.html ">Altair gallery</a>
```

Some attributes can be assigned to *any* type of element, such as `class` and `id`.

Classes and IDs

Classes and IDs are extremely useful attributes, as they can be referenced later to identify specific pieces of content. Your CSS and JavaScript code will rely heavily on classes and IDs to identify elements. For example:

```
<p>Brilliant paragraph</p>
<p>Insightful paragraph</p>
<p class="awesome">Awe-inspiring paragraph</p>
```

These are three very uplifting paragraphs, but only one of them is truly awesome, as I’ve indicated with `class="awesome"`. The third paragraph becomes part of a *class* of *awesome* elements, and it can be selected and manipulated along with other class members. (We’ll get to that in a moment.)

We can assign elements multiple classes, simply by separating them with a space:

```
<p class="uplifting">Brilliant paragraph</p>
<p class="uplifting">Insightful paragraph</p>
<p class="uplifting awesome">Awe-inspiring paragraph</p>
```

Now, all three paragraphs are `uplifting`, but only the last one is both `uplifting` *and* `awesome`.

IDs are used in much the same way, but there can be only one ID per element, and each ID value can be used only once on the page. For example:

```
<div id="content">
  <div id="visualization"></div>
  <div id="button"></div>
</div>
```

IDs are useful when a single element has some special quality, like a `div` that functions as a button or as a container for other content on the page.

As a general rule, if there will be only *one* such element on the page, you can use an `id`. Otherwise, use a `class`.

WARNING

Class and ID names cannot begin with numerals; they must begin with alphabetic characters. So `id="1"` won't work, but `id="item1"` will. The browser will not give you any errors; your code simply won't work, and you will go crazy trying to figure out why.

Comments

As code grows in size and complexity, it is good practice to include comments. These are friendly notes that you leave for yourself to remind you why you wrote the code the way you did. If you are like me, you will revisit projects only weeks later and have lost all recollections of it. Commenting is an easy way to reach out and provide guidance and solace to your future (and very confused) self.

If you're collaborating with anyone else—especially in a professional context—comments are essential.

In HTML, comments are written in the following format:

```
<!-- Your comment here -->
```

Anything between the `<!--` and `-->` will be ignored by the web browser.

DOM

The term *Document Object Model* refers to the hierarchical structure of HTML. Each pair of bracketed tags (or, in some cases, a single tag) is an *element*, and we refer to elements' relationships to each other in human terms: parent, child, sibling, ancestor, and descendant. For example, in this HTML:

```
<html>
  <body>
    <h1>Breaking News</h1>
    <p></p>
  </body>
</html>
```

`body` is the parent element to both of its children, `h1` and `p` (which are siblings to each other). All elements on the page are descendants of `html`.

Web browsers parse the DOM to make sense of a page's content. As coders building visualizations, we care about the DOM, because our code must navigate its hierarchy to apply styles and actions to its elements. We don't want to make *all* the `div` elements blue; we need to know how to select just the `divs` of the class `sky` and make *them* blue.

Developer Tools

In the olden days, the web development process went like this:

1. Write some code in a text editor.
2. Save the files.

3. Switch to a browser window.
4. Reload the page, and see if your code worked.
5. If it didn't work, take a guess at what went wrong inside the magical black box of the web browser, then go back to step 1.

Browsers were notoriously secretive about what went on *inside* the rendering engine, which made debugging a total nightmare. (Seriously, in the late 1990s and early 2000s, I literally had nightmares about this.) Fortunately, we live in a more enlightened age, and every modern-day browser has built-in *developer tools* that expose the inner workings of the beast and enable us to poke around under the hood (to mix incompatible metaphors).

All this is to say that developer tools are a big deal and you will rely on them heavily to both test your code and, when something breaks, figure out what went wrong.

Let's start with the simplest possible use of the developer tools: viewing the raw source code of an HTML page (see next figure).

Every browser supports this, although different browsers hide this option in different places. In Chrome, it's under View→Developer→View Source. In Firefox, look under Tools→Web Developer→Page Source. In Safari, it's under Develop→Show Page Source (although you must first set the "Develop" menu to display under Safari→Preferences→Advanced). Going forward, I'm going to assume that you're using the newest version of whatever browser you choose.

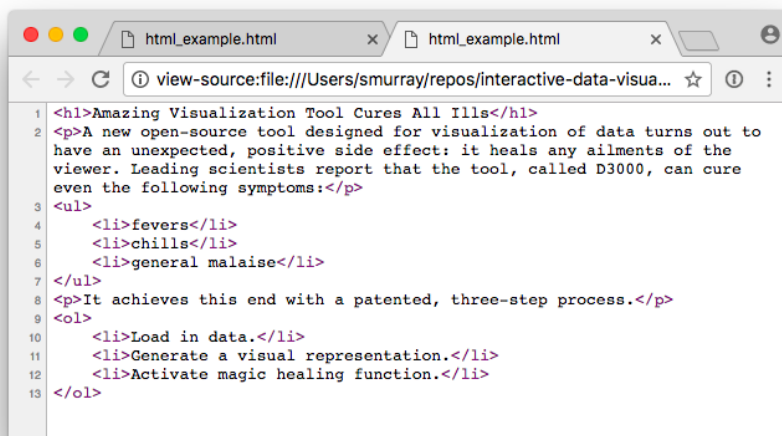


Figure 2. Looking at the source code in a new window in Chrome

That gets you the raw HTML, but if any JavaScript code has been executed, the current DOM may be vastly different.

Fortunately, your browser's developer tools enable you to see the current state of the DOM. And, again, the developer tools are different in every browser. We'll start with the element inspector. In Chrome, find them under View→Developer→Developer Tools→Elements. In Firefox, try Tools→Web Developer→Inspector. In Safari, first enable the developer tools (in Safari→Preferences→Advanced). Then, in the Develop menu, choose Show Web Inspector. In any

browser, you can also use the corresponding keyboard shortcut (as shown adjacent to the menu item) or right-click and choose “inspect element” or something similar.

I’ll use Chrome throughout this book, for consistency and because I prefer its developer tools. Your browser might look a bit different from my screenshots, but the functionality will be very similar.

[Figure 3-3](#) shows the Elements tab of Chrome’s web inspector. Here we can see the current state of the DOM. This is useful because your code will modify DOM elements dynamically. In the web inspector, you can watch elements as they change.

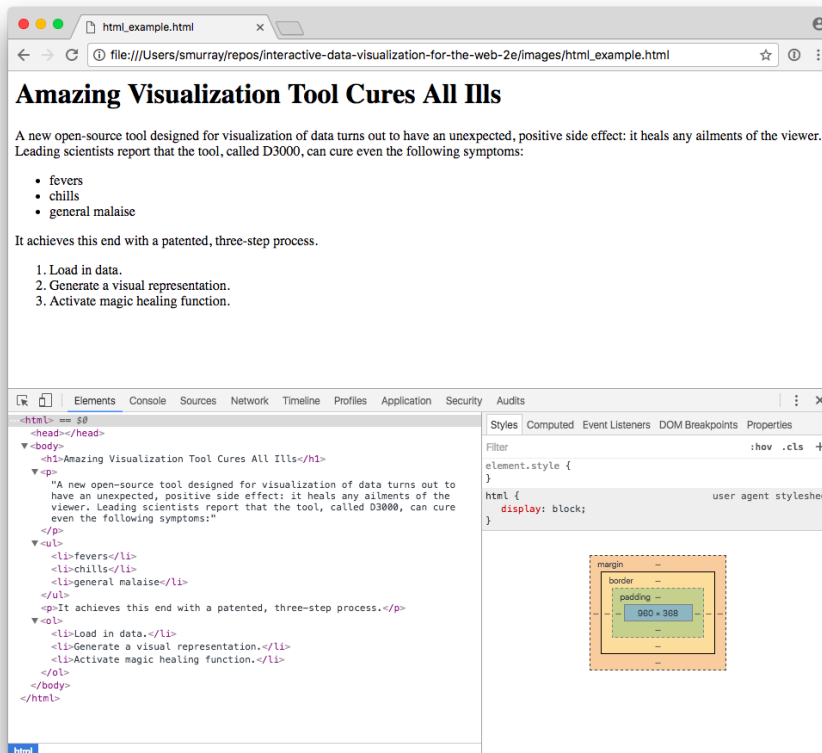


Figure 3. Chrome’s web inspector

If you look closely, you’ll already see some differences between the raw HTML and the DOM, including the fact that Chrome generated the required `html`, `head`, and `body` elements. (I was lazy and didn’t include them in my original HTML.)

Rendering and the Box Model

Rendering is the process by which browsers, after parsing the HTML and generating the DOM, apply visual rules to the DOM contents and draw those pixels to the screen.

The most important thing to keep in mind when considering how browsers render content is this: to a browser, everything is a box.

Paragraphs, `divs`, `spans`—all are boxes in the sense that they are two-dimensional rectangles, with properties that any rectangle can have, such as width, height, and positions in space. Even if

something looks curved or irregularly shaped, rest assured, to the browser, it is merely another rectangular box.

You can see these boxes with the help of the web inspector. Just mouse over any element, and the box associated with that element is highlighted in blue, as shown in [Figure 3-4](#).

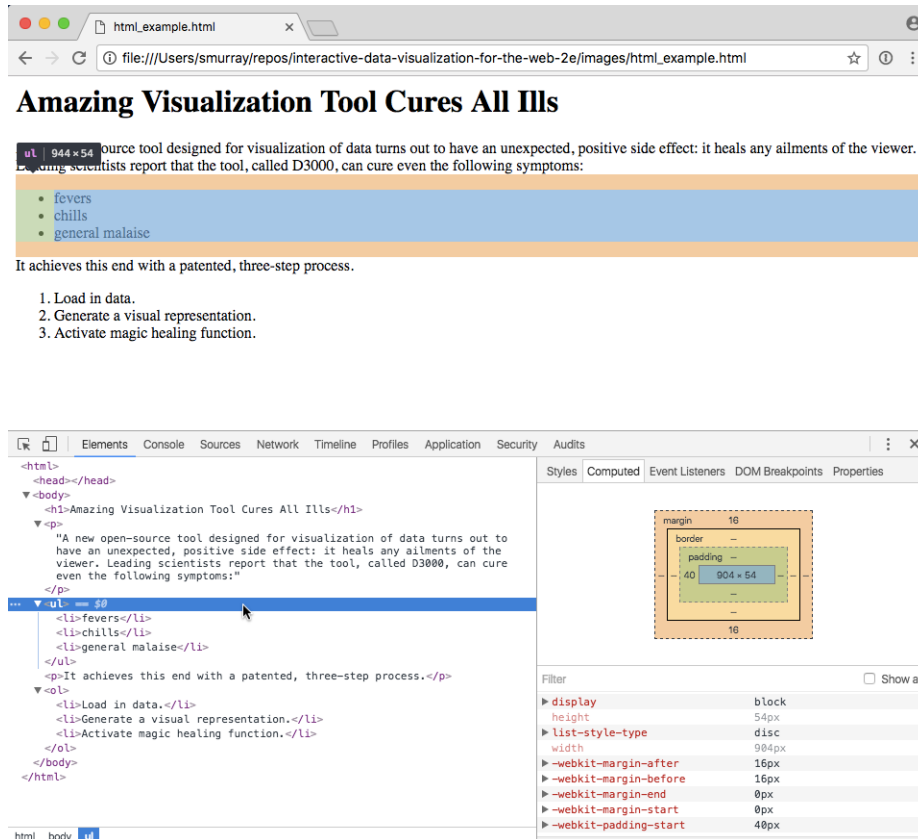


Figure 4. Inspector with element box highlighted

There's a lot of information about the `ul` unordered list here. Note that the list's total dimensions (width and height) are shown in a tooltip at the element's upper-left corner. Also, the list's position in the DOM hierarchy is indicated in the lower-left corner of the inspector: `html > body > ul`.

The box for the `ul` expands to fill the width of the entire window because it is a *block-level* element. (Note how under "Computed Style" is listed `display: block`.) This is in contrast to *inline* elements, which rest *in line* with each other, not stacked on top of each other like blocks. Common inline elements include `strong`, `em`, `a`, and `span`.

By default, block-level elements expand to fill their container elements and force any subsequent sibling elements further down the page. Inline elements do not expand to fill extra space, and happily exist side by side, next to their fellow inline neighbors. (Discussion question: what kind of element would you rather be?)

CSS

Cascading Style Sheets are used to style the visual presentation of DOM elements. CSS looks like the following:

```
body {
  background-color: white;
  color: black;
}
```

CSS styles consist of *selectors* and *properties*. Selectors are followed by properties, grouped in curly brackets. A property and its value are separated by a colon, and the line is terminated with a semicolon, like the following:

```
selector {
  property: value;
  property: value;
  property: value;
}
```

You can apply the same properties to multiple selectors at once by separating the selectors with a comma, as in the following:

```
selectorA,
selectorB,
selectorC {
  property: value;
  property: value;
  property: value;
}
```

For example, you might want to specify that both `p` paragraphs and `li` list items should use the same font size, line height, and color.

```
p,
li {
  font-size: 12px;
  line-height: 14px;
  color: orange;
}
```

Collectively, this whole chunk of code (selectors and bracketed properties) is called a *CSS rule*.

Selectors

Many javascript visualization libraries use CSS-style selectors to identify elements on which to operate, so it's important to understand how to use them.

Selectors identify specific elements to which styles will be applied. There are several different kinds of selectors. We'll use only the simplest ones.

Type selectors

These are the simplest. They match DOM elements with the same name:

```
h1      /* Selects all level 1 headings */
p       /* Selects all paragraphs */
strong  /* Selects all strong elements */
em      /* Selects all em elements */
div     /* Selects all divs */
```

Descendant selectors

These match elements that are contained by (or “descended from”) another element. We will rely heavily on descendant selectors to apply styles:

```
h1 em      /* Selects em elements contained in an h1 */
div p      /* Selects p elements contained in a div  */
```

Class selectors

These match elements of any type that have been assigned a specific class. Class names are preceded with a period, as shown here:

```
.caption   /* Selects elements with class "caption" */
.label     /* Selects elements with class "label"   */
.axis      /* Selects elements with class "axis"   */
```

Because elements can have more than one class, you can target elements with multiple classes by stringing the classes together, as in the following:

```
.div.vega-actions /* Could target vega options menu */
.div.vis          /* Could target the first chart  */
```

ID selectors

These match the single element with a given ID. (Remember, IDs can be used only once each in the DOM.) IDs are preceded with a hash mark.

```
#header    /* Selects element with ID "header"      */
#nav       /* Selects element with ID "nav"          */
#export    /* Selects element with ID "export"       */
```

Selectors get progressively more useful as you combine them in different ways to target specific elements. You can string selectors together to get very specific results. For example:

```
div.sidebar /* Selects divs with class "sidebar", but
              not other elements with that class */
#button.on  /* Selects element with ID "button", but
              only when the class "on" is applied */
```

Remember, because the DOM is dynamic, classes and IDs can be added and removed, so you might have CSS rules that apply only in certain scenarios.

For details on additional selectors, see the [Mozilla Developer Network](#).

Properties and Values

Groups of property/value pairs cumulatively form the styles:

```
margin: 10px;
padding: 25px;
background-color: yellow;
color: pink;
font-family: Helvetica, Arial, sans-serif;
```

At the risk of stating the obvious, notice that each property expects a different kind of information. `color` wants a color, `margin` requires a measurement (here in `px` or pixels), and so on.

By the way, colors can be specified in several different formats, such as:

- Named colors: `orange`
- Hex values: `#3388aa` or `#38a`
- RGB values: `rgb(10, 150, 20)`
- RGB with alpha transparency: `rgba(10, 150, 20, 0.5)`

You can find [exhaustive lists of properties online](#); I won't try to list them here. Instead, I'll just introduce relevant properties as we go.

Comments

```
/* By the way, this is what a comment looks like
   in CSS. They start with a slash-asterisk pair,
   and end with an asterisk-slash pair. Anything
   in between will be ignored. */
```

Referencing Styles

There are three common ways to apply CSS style rules to your HTML document.

EMBED THE CSS IN YOUR HTML

If you embed the CSS rules in your HTML document, you can keep everything in one file. In the document `head`, include all CSS code within a `style` element.

```
<html>
  <head>
    <style type="text/css">

      p {
        font-size: 24px;
        font-weight: bold;
        background-color: red;
        color: white;
      }

    </style>
  </head>
  <body>
    <p>If I were to ask you, as a mere paragraph, would you say that I have
style?</p>
  </body>
</html>
```

That HTML page with CSS renders as shown in next figure

**If I were to ask you, as a mere paragraph,
would you say that I have style?**

Figure 5. Rendering of an embedded CSS rule

Embedding is the simplest option, but I generally prefer to keep different kinds of code (e.g., HTML, CSS, JavaScript) in separate documents.

REFERENCE AN EXTERNAL STYLESHEET FROM THE HTML

To store CSS outside of your HTML, save it in a plain-text file with a `.css` suffix, such as `style.css`. Then use a `link` element in the document `head` to reference the external CSS file, like so:

```
<html>
  <head>
    <link rel="stylesheet" href="style.css">
```

```

</head>
<body>
  <p>If I were to ask you, as a mere paragraph, would you say that
    I have style?</p>
</body>
</html>

```

This example renders exactly the same as the prior example.

ATTACH INLINE STYLES

A third method is to attach style rules *inline* directly to elements in the HTML. You can do this by adding a `style` attribute to any element. Then include the CSS rules within the double quotation marks. The result is shown in [Figure 3-6](#).

```

<p style="color: blue; font-size: 48px; font-style: italic;">Inline styles are
kind of a hassle</p>

```

Inline styles are kind of a hassle

Figure 3-6. Rendering of an inline CSS rule

Because inline styles are attached directly to elements, there is no need for selectors.

Inline styles are messy and hard to read, but they are useful for giving special treatment to a single element, when that style information doesn't make sense in a larger stylesheet. We'll learn how to apply inline styles programmatically with D3 (which is much easier than typing them in by hand, one at a time).

Inheritance, Cascading, and Specificity

Many style properties are *inherited* by an element's descendants unless otherwise specified. For example, this document's style rule applies to the `div`:

```

<html>
  <head>
    <title></title>
    <style type="text/css">

      div {
        background-color: red;
        font-size: 24px;
        font-weight: bold;
        color: white;
      }

    </style>
  </head>
  <body>
    <p>I am a sibling to the div.</p>
    <div>
      <p>I am a descendant and child of the div.</p>
    </div>
  </body>
</html>

```

Yet when this page renders, the styles intended for the `div` (red background, bold text, and so on) are *inherited* by the second paragraph, as shown in [Figure 3-7](#), because that `p` is a descendant of the styled `div`. Notice how the second paragraph's text is white, even though we never specified `p { color: white; }`.

I am a sibling to the div.

I am a descendant and child of the div.

Figure 3-7. Inherited style

Inheritance is a great feature of CSS, as children adopt the styles of their parents. (There's a metaphor in there somewhere.)

Finally, an answer to the most pressing question of the day: why are they called Cascading Style Sheets? It's because selector matches cascade from the top down. When more than one selector applies to an element, the later rule generally overrides the earlier one. For example, the following rules set the text of all paragraph elements in the document to be blue *except* for those with the class of `highlight` applied, which will be black *and* have a yellow background, as shown in [Figure 3-8](#). The rules for `p` are applied first, but then the rules for `p.highlight` override the less specific `p` rules.

```
p {
  color: blue;
}

p.highlight {
  color: black;
  background-color: yellow;
}
```

A regular paragraph

A highlighted paragraph

Figure 3-8. CSS cascading and inheritance at work

Later rules generally override earlier ones, but not always. The true logic has to do with the *specificity* of each selector. The `p.highlight` selector would override the `p` rule even if it were listed first, simply because it is a more specific selector. If two selectors have the same specificity, then the later one will be applied.

This is one of the main causes of confusion with CSS. The rules for calculating specificity are inscrutable, and I won't cover them here. To save yourself headaches later, keep your selectors clear and easy to read. Start with general selectors on top, and work your way down to more specific ones, and you'll be all right.

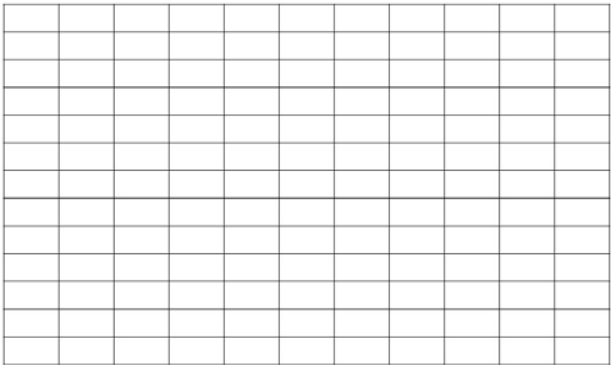
Layout

CSS is very useful also to create a layout. A layout is a grid where we position elements in the screen.

We will define it with `display:grid`, `grid-template-columns`, `grid-template-rows`, `grid-column-start`, `grid-column-end`, `grid-row-start` and `grid-row-end` properties.

The first step is to mentally divide the page into a basic grid

A screen of 1100 px x 1300 px



Then, we create our layout, the spaces that will guide our composition

c1r1	c2r1	c3r1	c4r1
c1r2	c2r2	c3r2	c4r2
c1r3	c2r3	c3r3	c4r3
c1r4	c2r4	c3r4	c4r4
c1r5	c2r5	c3r5	c4r5

```
<style>
  #main{
    display:grid;
    grid-template-columns:300px 300px 300px 200px;
    grid-template-rows: 300px 300px 300px 200px 200px;
  }
```

And finally we position our elements in this layout

Elements are positioned on the grid with `grid-column-start`, `grid-column-end`, `grid-row-start` and `grid-row-end` styles

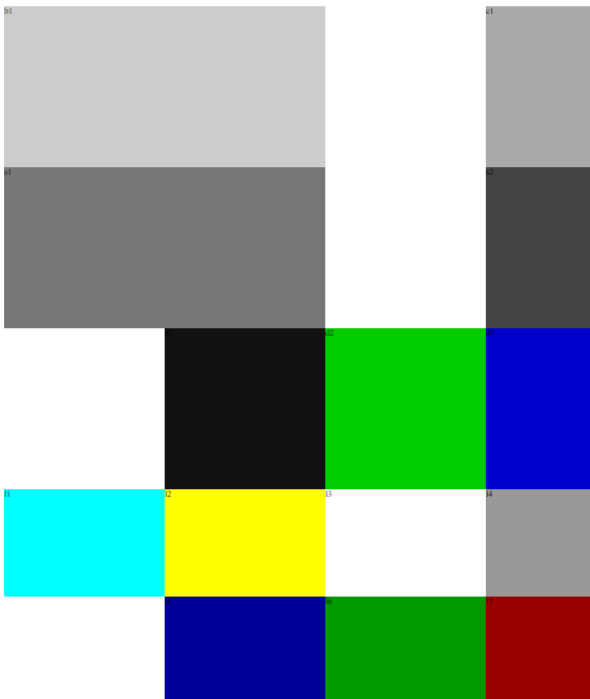
c1r1	c2r1 Bar-chart	c3r1	c4r1 colorLegend
c1r2	c2r2 scat1	c3r2	c4r2 sizeLegend
c1r3	c2r3 donut-chart-product	c3r3 donut-chart-product1	c4r3 donut3
c1r4 line1	c2r4 line2	c3r4 line3	c4r4 line4
c1r5	c2r5 line5	c3r5 line6	c4r5 line7

```
#bar-chart {grid-column-start:1;
  grid-column-end:3;
  grid-row-start:1;
  background-color:#CCC
}

#colorLegend {grid-column-start:4;
  grid-row-start:1;
  background-color:#AAA
}

...
#line7 {grid-column-start:4;
  grid-row-start:5;
  background-color:#900
}
```

Producing this final effect (with charts in real visualizations)



See *grid.html* on the *grid-demo* folder to see the working example.

JavaScript

JavaScript is the scripting language that can make pages dynamic by manipulating the DOM after a page has already loaded in the browser.

See *page5.html* on the *demos* folder to see several examples of console output.

As I mentioned earlier, working with visualizations at a high level is also a process of getting to know JavaScript. We'll dig in deeper as we go, but here is a taste to get you started.

Hello, Console

Normally, we write JavaScript code (or, a “script”) in a text file, and then load that file to the browser in a web page. But you can also just type JavaScript code directly into your browser. This is an easy and quick way to get your feet wet and test out code. We’ll also use the JavaScript console for debugging, as it’s an essential tool for seeing what’s going on with your code.

In Chrome, select View→Developer→JavaScript Console. In Firefox, choose Tools→Web Developer→Web Console. In Safari, go to Develop→Show Error Console (see [Figure 3-9](#)).

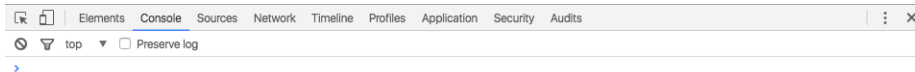


Figure 3-9. A fresh JavaScript console...delicious!

The console accepts one line of code at a time, and it always spits back the result of whatever you input. For example, if you enter the number 7, the console returns the mind-numbingly obvious result of 7. Brilliant.

Other times, you’ll want your script to print values out to the console automatically, so you can keep an eye on what’s going on. As you’ll see in some examples that follow, you can use `console.log("something");` to do that.

See [page3.html](#) and the corresponding script on the [demos](#) folder to see several examples of console output.

Variables

Variables are containers for data. A simple variable holds one value:

```
var number = 5;
```

Also note that in JavaScript, statements are concluded with a semicolon.

JSON

At some point you will encounter JavaScript Object Notation. You can [read up on the details](#), but JSON is basically a specific syntax for organizing data as JavaScript objects. The syntax is optimized for use with JavaScript (obviously) and AJAX requests, which is why you’ll see a lot of web-based application programming interfaces (APIs) that return data formatted as JSON. It’s faster and easier to parse with JavaScript than XML, and of course many visualizations work well with it.

All that, and it doesn’t look very strange:

```
{  
  
  "kind": "grape",
```

```

    "color": "red",
    "quantity": 12,
    "tasty": true
  }

```

The data property names are surrounded by double quotation marks `" "`, making them string values.

JSON objects, like all other JavaScript objects, can of course be stored in variables like so:

```

var jsonFruit = {
  "kind": "grape",
  "color": "red",
  "quantity": 12,
  "tasty": true
};

```

GEOJSON

Just as JSON is just a formalization of existing JavaScript object syntax, GeoJSON is a formalized syntax of JSON objects, optimized for storing geodata. All GeoJSON objects are JSON objects, and all JSON objects are JavaScript objects.

GeoJSON can store points in geographical space (typically as longitude/latitude coordinates), but also shapes (such as lines and polygons) and other spatial features. If you have a lot of geodata, it's worth it to parse it into GeoJSON format for best use with D3.

We'll get into the details of GeoJSON when we talk about geomaps, but for now, just know that this is what simple GeoJSON data looks like:

```

{
  "type": "FeatureCollection",
  "features": [
    {
      "type": "Feature",
      "geometry": {
        "type": "Point",
        "coordinates": [ 150.1282427, -24.471803 ]
      },
      "properties": {
        "type": "town"
      }
    }
  ]
}

```

Confusingly, longitude is always listed before latitude. Get used to thinking in terms of lon/lat instead of lat/lon. This makes sense once you realize that lon/lat is sort of equivalent to x/y, and in graphics we are used to putting x first. x and longitude move things horizontally, while y and latitude move things vertically.

Functions

Functions are chunks of code that do things.

More specifically, functions are special because they can take arguments or parameters as input, and then return values as output. Parentheses are used to *call* (execute) a function. If that function requires any arguments (input values), then you *pass* them to the function by including them in the parentheses.

Whenever you see something like the following code, you know it's a function:

```
showError(el, error);
```

In fact, you've already seen functions at work with `console.log`, as in the following:

```
console.log("Look at me. I can do stuff!");
```

Comments

```
/* JavaScript supports CSS-style comments like this. */  
  
// But double-slashes can be used as well.  
// Anything following // on the same line will be ignored.  
// This is helpful for including brief notes to yourself  
// as to what each line of code does, as in:  
  
console.log("Brilliant"); //Prints "Brilliant" to the console
```

Referencing Scripts

Scripts can be included directly in HTML, between two `script` tags, as in:

```
<body>  
  <script type="text/javascript">  
    alert("Hello, world!");  
  </script>  
</body>
```

or stored in a separate file with a `.js` suffix, and then referenced somewhere in the HTML (could be in the `head`, as shown here, or also just before the end of the closing `body` tag):

```
<head>  
  <title>Page Title</title>  
  <script type="text/javascript" src="myscript.js"></script>  
</head>
```


Javascript and Altair

The HTML code generated by Altair is standalone. You can use it directly, without any further work. If you look at the code you'll find known elements as `html`, `head`, `body` and also `style` and `script` elements. In the head you will find references to external javascript libraries:

```
<script type="text/javascript"
src="https://cdn.jsdelivr.net/npm//vega@4"></script>

<script type="text/javascript" src="https://cdn.jsdelivr.net/npm//vega-
lite@2.6.0"></script>

<script type="text/javascript" src="https://cdn.jsdelivr.net/npm//vega-
embed@3"></script>
```

You will find also a basic styling for action buttons and error messages:

```
<style>
.vega-actions a {
  margin-right: 12px;
  color: #757575;
  font-weight: normal;
  font-size: 13px;
}
.error {
  color: red;
}
</style>
```

And in the body, as the only html element there is a div without anything inside:

```
<div id="vis"></div>
```

Followed by a long script, where

- the specification of the graph is given:

```
var spec = {"config": {"view": {"width": 400, "height": 300}}, "hconcat":...
"transform": [{"filter": {"selection": "selector001"}}]}], "$schema":
"https://vega.github.io/schema/vega-lite/v2.6.0.json"};
```

- vega-lite options are defined

```
var embedOpt = {"mode": "vega-lite"};
```

- and a function for showing errors

```
function showError(el, error){
  el.innerHTML = ('<div class="error" style="color:red;">'
    + '<p>JavaScript Error: ' + error.message + '</p>'
    + '<p>This usually means there's a typo in your chart
specification. "
    + "See the javascript console for the full
traceback.</p>'
    + '</div>');
  throw error;
}
```

Finally, *el* gets the DOM element called vis

```
const el = document.getElementById('vis');
```

and embeds in it the spec with the options

```
vegaEmbed("#vis", spec, embedOpt)
```

For every chart, all these elements have the same names.

If we want to combine two charts in the same page, we will have to merge the codes, and change the identifiers to vis2, spec2, and el2.

```
<div id="vis"></div>
<div id="vis2"></div>
<script>
var spec = { "...
var spec2 = { "...
var embedOpt = {"mode": "vega-lite"}; // same code for both charts; otherwise
change
function showError(el, error){... //again the same for both charts
const el = document.getElementById('vis');
const el2 = document.getElementById('vis2');
vegaEmbed("#vis", spec, embedOpt)
    .catch(error => showError(el, error));
vegaEmbed("#vis2", spec2, embedOpt)
    .catch(error => showError(el2, error));
```

See altair-demo on the exercices folder to see two Altair charts as they have been published by Altair, and then combined.