Week 7

This is CS50. Harvard University. Fall 2015.

Anna Whitney

Table of Contents

1. Introduction

- In Problem Set 4, you were challenged to find as many of the staff from the photos you recovered and take selfies with them. Ken, a photographer on the CS50 staff, got selfies with 24 staff members! Of actual students, the winners were Lance, with 15 photos, and Bonnie, with 14 photos but some containing multiple staff. Their sections will get a catered lunch.
- **CS50 Lunch** on Friday, as usual; RSVP at cs50.harvard.edu/rsvp[1].
- **CS50 Seminars** are coming up!
  # We're getting to the point of the semester where you should start to think about final project ideas.
  # To this end, **pre-proposals** are soon due, in which you'll give your TF some ideas of what you're thinking about for a final project.
  # Many students end up doing web projects, and since we're just now getting into web programming, don't worry if you don't yet know exactly how you'd implement your ideas (your TF will help you figure out what's feasible).
  # CS50 Seminars give you a chance to explore tools and languages that aren't covered in the course itself, many of which might be of use on a final project.

[1] http://cs50.harvard.edu/rsvp

    # To see the list of seminars and register for any of them, go to cs50.harvard.edu/register[2].

2. HTTP

- We left off last time with `GET`, the messages that are actually passed in the "envelope", so to speak, from computer to computer.

    # A `GET` request looks something like this:

```
GET / HTTP/1.1
Host: www.google.com
...
```

        # This first line, `GET / HTTP/1.1`, means that we're requesting the main page of the website - the index, or default webpage - using the `HTTP/1.1` protocol.

    # The server would respond with something like the following:

```
HTTP/1.1 200 OK
Content-Type: text/html
...
```

        # We rarely see the status code `200 OK` as humans, because it means everything went fine, so the browser just shows us the webpage content.

        # The most common content type is `text/html`, which we'll discuss in more detail now.

3. HTML

- In the coming weeks, you can use any browser, but we recommend Chrome because it has lots of useful tools for developers.
- In Chrome, you can right-click/Ctrl-click on a webpage and select "inspect element" from the menu that appears to open Chrome's Inspector, a debugging tool that lets you see under the hood of what's happening on a webpage.

[2] http://cs50.harvard.edu/register

The Inspector has lots of features, but let's look at the **Network** tab.

We press the recording button so it's red, and check "preserve log", so that everything that happens with the network will be recorded. Then we go to www.facebook.com:

When you visit facebook.com, you don't just send one `GET` request, you actually send many `GET` requests to load different parts of the page.

The one we care about, though, is the request for the main Facebook page itself. Let's look at the actual header that Chrome sent to www.facebook.com:

Although this isn't formatted quite the way we wrote it before, we can see that it's the same kind of request, with the path `/` again representing the main page of the website, and `HTTP/1.1` being the protocol used.

The inspector also lets us look at the response headers that Facebook returned to our browser.

These headers are composed of many key-value pairs (including the one we're familiar with from before: `content type: text/html`.

- We can also right-click on the page and select "View Page Source" to see the HTML source of the webpage (this is not limited to Chrome - Firefox, IE, etc. all let you do this as well, although the menu options might look a little different).

The HTML that we see this way is sent in the response from Facebook, coming after those headers we saw in the inspector.

- Recall our simple webpage from last lecture, with this HTML source:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

- Our code looks much prettier than the Facebook source code - we use indentation to make our HTML more readable, but the Facebook source is all strung together without whitespace in between. This is because whitespace takes up space! The file without the whitespace is smaller (fewer bytes) than the file with whitespace, because whitespace characters cost bytes too.
    # If just one extra space character were included in the HTML for Facebook's homepage, and they have a billion users who all access the homepage, that's an extra *gigabyte* of data that needs to be sent from Facebook's servers!
    # For this reason, it's common in web development to **minify** your code, or remove any superfluous characters to make the file sizes as small as possible.
    # As we're learning web programming, we'll start out by writing our code in a more human-readable fashion. (As an aside, nobody writes code by hand that looks like what we saw in the Facebook source - you write code that's human-readable, and once you've got it working, you run it through a program called a minifier that turns it into the uglier, but smaller version that actually gets sent across the network.)
- If we look in the Elements tab of the Chrome inspector, we can see the **pretty-printed** version of the HTML source - Chrome has conveniently de-obfuscated it for us.
    # We can start to see the hierarchy of the webpage and click through it in this view.
- So as we did last time, let's type our simple HTML page into a text editor and save it as `hello.html`:

```
<!DOCTYPE html>

<html>
    <head>
        <title>hello, world</title>
    </head>
    <body>
        hello, world
    </body>
</html>
```

Then we can double-click this file and visit this webpage in Chrome.

4. Web Servers

- The problem with this webpage we've just created is that we're the only ones who can see it, because it's just a file on David's Mac. It's not accessible outside of his computer (or your computer, if you follow the above steps on your own computer).
- Let's look at Cloud9, the service that hosts the CS50 IDE. It has all of our workspaces hosted somewhere on the Internet, so any files we create in the IDE are already publicly accessible.
- If we copy our same HTML document above into a new file in the CS50 IDE and save it as `hello.html`, how can we open it as a webpage?
- Built into the IDE, in addition to the debugger and all the other tools available to you, is a full-fledged **web server**.
  - # A web server is just a program whose job it is to serve up web pages - to listen for requests from other computers and respond with the virtual envelopes containing the headers and page content.
  - # Our IDE's web server is an open-source[3] server called **Apache**, to which we've written a more usable interface called `apache50`.
- In the IDE's terminal, we can type the following:

```
jharvard@ide50:~/workspace $ apache50 start .
```

[3] https://en.wikipedia.org/wiki/Open-source_software

```
Setting Apache's document root to /home/ubuntu/workspace ...
 * Starting web server apache2
 *
Apache started successfully!
Your site is now available at https://ide50-jharvard.c9.io
jharvard@ide50:~/workspace $
```

- Our web server is now listening on the web at the address `ide50-jharvard.c9.io` (your address will be a little different depending on your username), on TCP port 80.

  # You can see what this address will be in the upper-right corner of your IDE.

- If we click on the URL in question, we see a pretty ugly index - just a directory listing. But we can see `hello.html` saved in this directory, and if we click on it, we can see the same "hello, world!" page as before.

  # Note that we're looking at this webpage inside the IDE - not because it's hosted there (since our web server makes it visible to the whole world), like when `hello.html` was on David's computer and we were looking at it there, but because in addition to the web server, the IDE also provides a simple web browser.

  # We can also copy-paste this URL into the regular Chrome browser bar, and we'll see the exact same thing! (You could even have a friend on an entirely different computer go to your IDE's address while you're running your web server, and they'll be able to see the same thing.)

- For the purposes of the course, you have your own unique address, a **subdomain** of the course's overarching domain, cs50.io (AKA ide50.c9.io), represented as `ide50-username.c9.io`.

5. Working with HTML

- Let's look at a slightly more complicated HTML file, `paragraphs.html`[4] (note that if you click on that link, you'll likely have to view source to see the actual HTML code, since your browser knows how to interpret HTML and so does it for you rather than showing you the raw source):

[4] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/paragraphs.html

```
<!DOCTYPE html>

<!--

paragraphs.html

David J. Malan
malan@harvard.edu

Displays some text.

Demonstrates paragraphs.

-->

<html>
    <head>
        <title>paragraphs</title>
    </head>
    <body>
        <p>
            Lorem ipsum dolor sit amet, consectetur adipiscing elit.
 Nullam in tincidunt augue. Duis imperdiet, justo ac iaculis rhoncus,
 erat elit dignissim mi, eu interdum velit sapien nec risus. Praesent
 ullamcorper nibh at volutpat aliquam. Nam sed aliquam risus. Nulla rutrum
 nunc augue, in varius lacus commodo in. Ut tincidunt nisi a convallis
 consequat. Fusce sed pulvinar nulla.
        </p>
        <p>
            Ut tempus rutrum arcu eget condimentum. Morbi elit ipsum,
 gravida faucibus sodales quis, varius at mi. Suspendisse id viverra
 lectus. Etiam dignissim interdum felis quis faucibus. Integer et
 vestibulum eros, non malesuada felis. Pellentesque porttitor eleifend
 laoreet. Duis sit amet pellentesque nisi. Aenean ligula mauris, volutpat
 sed luctus in, consectetur id turpis. Phasellus mattis dui ac metus
 blandit volutpat. Donec lorem arcu, sollicitudin in risus a, imperdiet
 condimentum augue. Ut at facilisis mauris. Curabitur sagittis augue in
 dictum gravida. Integer sed sem sed justo tempus ultrices eu non magna.
 Phasellus semper eros erat, a posuere nisi auctor et. Praesent dignissim
 orci aliquam laoreet scelerisque.
        </p>
        <p>
            Mauris eget erat arcu. Maecenas ac ante vel ipsum bibendum
 varius. Nunc tristique nulla eget tincidunt molestie. Morbi sed mauris
 eu lectus vehicula iaculis ac id lacus. Etiam sit amet magna massa. In
 pulvinar sapien ac mi ultrices, quis consequat nisl hendrerit. Aliquam
 pharetra nec sem non vehicula. In et risus leo. Ut tristique ornare nisl
```

    # First, notice in lines 3-12, we have a comment, just as we had in our C code, but the format is a little different: comments are opened with `<!--` and closed with `-->`, instead of being opened with `/*` and closed with `*/`.

    # We've introduced a new tag - `<p>`, or paragraph.

- If we restart our web server using today's `src7m` directory as the root directory, we'll be able to see this and all the other source files:

```
jharvard@ide50:~/workspace $ apache50 stop
 * Stopping web server apache2
 *
jharvard@ide50:~/workspace $ apache50 start src7m/
Setting Apache's document root to /home/ubuntu/workspace/src7m ...
 * Starting web server apache2
 *
Apache started successfully!
Your site is now available at https://ide50-jharvard.c9.io
jharvard@ide50:~/workspace $
```

- Now when we go to `ide50-jharvard.c9.io`, we see a much longer list of files (the files in the `src7m` directory), among them `paragraphs.html`.

- If we click on `paragraphs.html`, we can see that those `<p>` tags make line breaks between our pseudo-Latin paragraphs.

    # If we replace the `<p>` tags with actual line breaks in our HTML source, the paragraphs all run together, because the browser just ignores extra whitespace - it only does exactly what it's told to do.

- HTML syntax seems to consist of opening a tag (e.g., `<head>` or `<p>`) and then closing those tags (e.g., `</head>`, `</p>`). Start tags just consist of a tag name in angle brackets, and end tags are similar, but the tag name is prefixed with a forward slash.

- What if we want more than just plain, 12pt Times New Roman text? We can make our text bigger and bold, as in `headings.html`[5]:

[5] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/headings.html

```
<html>
    <head>
        <title>headings</title>
    </head>
    <body>
        <h1>One</h1>
        <h2>Two</h2>
        <h3>Three</h3>
        <h4>Four</h4>
        <h5>Five</h5>
        <h6>Six</h6>
    </body>
</html>
```

We're not directly bolding our text or increasing the size of our text, but we're using **heading** tags, `<h1>` through `<h6>`, and our browser knows that each heading should be bold and a slightly different size.

- We can also make lists, as in `list.html`[6]:

```
<!DOCTYPE html>

<html>
    <head>
        <title>list</title>
    </head>
    <body>
        <ul>
            <li>Cabot</li>
            <li>Currier</li>
            <li>Pforzheimer</li>
        </ul>
    </body>
</html>
```

This displays a bulleted list of houses on the Quad. The `<ul>` tags stand for **unordered list** (i.e., bulleted as opposed to numbered - there's also an `<ol>` tag, ordered list, for numbered lists), while the `<li>` tags indicate list items.

[6] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/list.html

    # All of our indentation and whitespace is just for our sake, not for the browser's (but you should still write your HTML in pretty-printed, correctly indented style for the sake of the humans reading it!).

- This is all still just text. How would we go about inserting an image?

    # Let's go find an image on Google Image Search of, say, Rick Astley:

    # We can insert this into a web page like so:

```
<!DOCTYPE html>

<html>
    <head>
        <title>Rick</title>
    </head>
    <body>
        This is Rick Astley: <img src="http://images.musictimes.com/
data/images/full/3561/happy-birthday-rick-astley.jpg"/>
    </body>
</html>
```

Note that in line 8, we have a new tag: `<img src="…"/>`. This is different from some of our other tags because it doesn't have a separate close tag - it's a **self-closing** tag, because the only content of the image tag is its source, so we don't need a space between a start and end tag.

If we look at this page in the browser, now we can see the above picture of Rick Astley.

- It's also possible to embed videos and other media types in HTML web pages.

  # If we go to a YouTube video, such as the video for this week's lecture,[7] under the video there's a Share tab; one option under that tab is "Embed". Under Embed, we have the following:

```
<iframe width="560" height="315" src="https://www.youtube.com/embed/
GUtPQIDSwrA" frameborder="0" allowfullscreen></iframe>
```

  # `<iframe …>` indicates an inline frame. This one isn't self-closing (it's possible to have an iframe with internal content), but like the image tag, it has **attributes** - additional syntax besides just the name of the tag that controls the behavior of the tag using key-value pairs `attribute="value"`.

- So even though these little web pages we've been creating are pretty boring, hopefully HTML itself seems pretty approachable - it's not a programming language; there are no functions, just a whole bunch of different tags, which can take various attributes.

[7] David Rickrolled the entire lecture hall, but if you're reading these notes rather than watching the lecture video, I won't do that to you.

    # Once you understand the basic framework, HTML is pretty self-teachable - there are plenty of online resources, and you can inspect the source code of websites you like.

- Another thing to keep in mind when constructing web pages is accessibility - for example, when we're putting an image in our web page, one way to help with this is to include "alternative text" with the `alt` attribute:

```
<img alt="Grumpy Cat" src="cat.jpg"/>
```

    # This lets someone who's using a screen reader, typically for reasons of sight, know what the image shows.

    # Note that in this example, our image source is not a URL elsewhere on the web but a local file called `cat.jpg`. You can download pictures from the Internet, or take your own pictures, and upload them to your IDE to use in a webpage.

        # Instead of a URL, we then can use a **relative path** (i.e., the path from our current directory to the image) to specify where the image is. If the image, like `cat.jpg`, is just in the same directory as the HTML file, then just the name of the file is needed; but if we move it to a subdirectory called `images`, then we could include it using `<img src="images/cat.jpg"/>`, for example.

- We have thus far been making very static web pages, but what if we want our site to actually do something? Let's go about building a search engine (of sorts).

    # We can start by setting up a simple form, as in `search-0.html`[8]:

[8] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/search-0.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>CS50 Search</title>
    </head>
    <body>
        <h1>CS50 Search</h1>
        <form action="https://www.google.com/search" method="get">
            <input name="q" type="text"/>
            <br/>
            <input type="submit" value="CS50 Search"/>
        </form>
    </body>
</html>
```

Recall from previously our concept of HTTP requests - this form sends a request to `google.com/search` using the method `GET`.

The `input` tags ask users for input, and the form then passes that input as parameters when it sends a request to the server specified in its `action` attribute (note that these are also self-closing).

- Here we have a parameter `q` that will be of type `text`. (Recall how we last time showed that queries to Google can be written as URLs of the form `google.com/search?q=query`!)
- The other `input` tag isn't really a parameter, but instead just a button that lets the user submit the form.

- These web pages are still pretty ugly. We can use something called **CSS**, or Cascading Style Sheets, to do some more interesting formatting.

Here's an example of using CSS to style a web page, in `css-0.html`[9]:

[9] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/css-0.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>css-0</title>
    </head>
    <body>
        <div style="text-align: center;">
            <div style="font-size: 36px; font-weight: bold;">
                John Harvard
            </div>
            <div style="font-size: 24px;">
                Welcome to my home page!
            </div>
            <div style="font-size: 12px;">
                Copyright &#169; John Harvard
            </div>
        </div>
    </body>
</html>
```

A `<div>` just means a division, a section of the webpage that has some attributes.

Within the `style` attribute of these `div`s, we can write CSS - a series of key-value pairs written as `key: value;`.

We can see in the first `<div>` tag that we're centering everything on the page, and then the `<div>` containing the text "John Harvard" is bold. Only the text "John Harvard" will be bolded, because the style attributes of a `div` only extend from its start tag to its end tag.

We can also use CSS attributes to change text color, background color and more.

- The problem with this format is that we're commingling two different languages - we've got a little bit of CSS in the midst of mostly HTML. Just as we could factor out some of our C code into header files, so too can we factor out our CSS from our HTML.

  # To apply CSS attributes to a part of our HTML without having it sitting right there in the tag, we can use **CSS selectors**, as in `css-1.html`[10]:

[10] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/css-1.html

```
<html>
    <head>
        <style>

            body
            {
                text-align: center;
            }

            #top
            {
                font-size: 36px;
                font-weight: bold;
            }

            #middle
            {
                font-size: 24px;
            }

            #bottom
            {
                font-size: 12px;
            }

        </style>
        <title>css-1</title>
    </head>
    <body>
        <div id="top">
            John Harvard
        </div>
        <div id="middle">
            Welcome to my home page!
        </div>
        <div id="bottom">
            Copyright &#169; John Harvard
        </div>
    </body>
</html>
```

    # This is structurally the exact same web page, but we've rearranged some things. We've given each `div` a unique identifier using the `id` attribute, which lets us refer to it inside the `<style>` tag in the `<head>` of the document.

        - We can refer to a tag by its unique `id` using `#id`.
        - Note that we can also style a type of tag, as we do with the `body` tag above.
        - Both of these (referring to a tag by type or by id in CSS) are examples of CSS selectors.

- We can go one better, though, and actually factor out our CSS into an entirely separate file, as in `css-2.html`[11]. In this case, we replace our `<style>` tag in the header with the following:

```
<link href="css-2.css" rel="stylesheet"/>
```

    # This means "link in the contents of the file css-2.css, and use it as a stylesheet."

    # In css-2.css, we have the exact same content that was previously in our style tag.

    # Factoring out our CSS into a separate file has a bunch of advantages:

        # We can apply the same styles across multiple HTML files.

        # If we want to make a change to the styles, we can change them in one place and have those changes automatically propagate to all our HTML files without us doing anything to individual HTML files.

        # It looks much cleaner!

- We can also have pages link to each other, as in `link.html`[12]:

[11] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/css-2.html
[12] http://cdn.cs50.net/2015/fall/lectures/7/m/src7m/link.html

```
<!DOCTYPE html>

<html>
    <head>
        <title>link</title>
    </head>
    <body>
        This is <a href="https://cs50.harvard.edu/">CS50</a>.
    </body>
</html>
```

This creates a **hyperlink** from our page, `link.html`, to the CS50 homepage.

Note, however, that we can make the link text say whatever we want - so malicious email spammers can include a line like this:

```
click here to reset your <a href="http://www.badplace.com/">https://
www.paypal.com/</a> password
```

The browser can actually show you where the link goes when you hover over it, so make sure to do this before clicking on any suspicious links.

- On Wednesday, we'll move on to a programming language called **PHP**, which we can use to dynamically generate webpages, and thus not be limited to static content hardcoded into our HTML.