LEARN ENOUGH

HTML

TO BE DANGEROUS

WEB BASICS **01**

TUTORIAL BY

MICHAEL HARTL & LEE DONAHOE

Learn Enough HTML to Be Dangerous

A tutorial introduction to HTML

Michael Hartl and Lee Donahoe

Contents

About the authors

Michael Hartl is the creator of the *Ruby on Rails Tutorial*, one of the leading introductions to web development, and is cofounder and principal author at Learn Enough. Previously, he was a physics instructor at the California Institute of Technology (Caltech), where he received a Lifetime Achievement Award for Excellence in Teaching. He is a graduate of Harvard College, has a Ph.D. in Physics from Caltech, and is an alumnus of the Y Combinator entrepreneur program.

Learn Enough cofounder Lee Donahoe is an entrepreneur, designer, and front-end developer. In addition to doing the design for Learn Enough, Softcover, and the Ruby on Rails Tutorial, he is also a cofounder and front-end developer for Coveralls, a leading test coverage analysis service, and is tech cofounder of Buck Mason, a men's clothing company once featured on ABC's *Shark Tank*. Lee is a graduate of USC, where he majored in Economics and studied Interactive Multimedia & Technologies.

Chapter 1

Basic HTML

HyperText Markup Language, or *HTML* for short, is the universal language of the World Wide Web. Every time you visit a website, the site's web server sends HTML to your browser, which then renders it as the web page you see on your screen. Because this process is universal, anyone who works with web technologies—which these days means virtually all developers, designers, and even many managers—can benefit from knowing the basics of what HTML is and how it works. *Learn Enough HTML to Be Dangerous* is designed to give you this foundation in basic HTML.

Appropriately enough, there are lots of HTML tutorials on the Web, but most of them use toy examples, focusing on HTML syntax in isolation, without showing how HTML is written and deployed in real life. In contrast, *Learn Enough HTML to Be Dangerous* not only shows you how to make real HTML pages, it shows you how to deploy an actual site to the live Web. If you have previously attempted or completed an HTML tutorial, it's likely that *Learn Enough HTML to Be Dangerous* will help you "put everything together" in a way you haven't seen before, including an emphasis on expanding your skillset with *technical sophistication* (Box 1.1).

At the same time, there are many HTML books on the market that are substantially too long for what is, at its core, a simple subject. You don't need to master 500 pages of material to be highly productive with HTML. Thus, in keeping with the Learn Enough philosophy, this tutorial teaches only the most important aspects of the subject—enough to be *dangerous*.

Box 1.1. Technical sophistication

If tech is the new literacy, technical sophistication is like being able to read and write. This includes being able to figure things out on your own (like sounding out words while reading) and look things up when you need them (like consulting a dictionary or thesaurus while writing).

On the Web, the alphabet is HTML.

In *Learn Enough HTML to Be Dangerous*, we'll constantly be on the lookout for chances to improve our technical sophistication. We'll deploy our website immediately to production (Section 1.3), getting over any bumps along the way. We'll push ourselves to read HTML we don't quite understand, content to get the gist at first before deepening our mastery later. And we'll put all our tools to use, combining the command line, a text editor, and version control to learn how to make HTML websites the Right Way™—professional-grade from the start.

Because of our pragmatic approach, the tools we'll be using are all professional-grade (Figure 1.1). They are the same tools covered in the Learn Enough Developer Fundamentals sequence, which you should follow now if you're not already familiar with them:

1. *Learn Enough Command Line to Be Dangerous* on the Unix command line
2. *Learn Enough Text Editor to Be Dangerous* on text editors
3. *Learn Enough Git to Be Dangerous* on version control with Git

To get even more out of the sequence, you can join our subscription service, the Learn Enough Society, which includes streaming videos and special enhanced versions of the online tutorials, among other benefits.

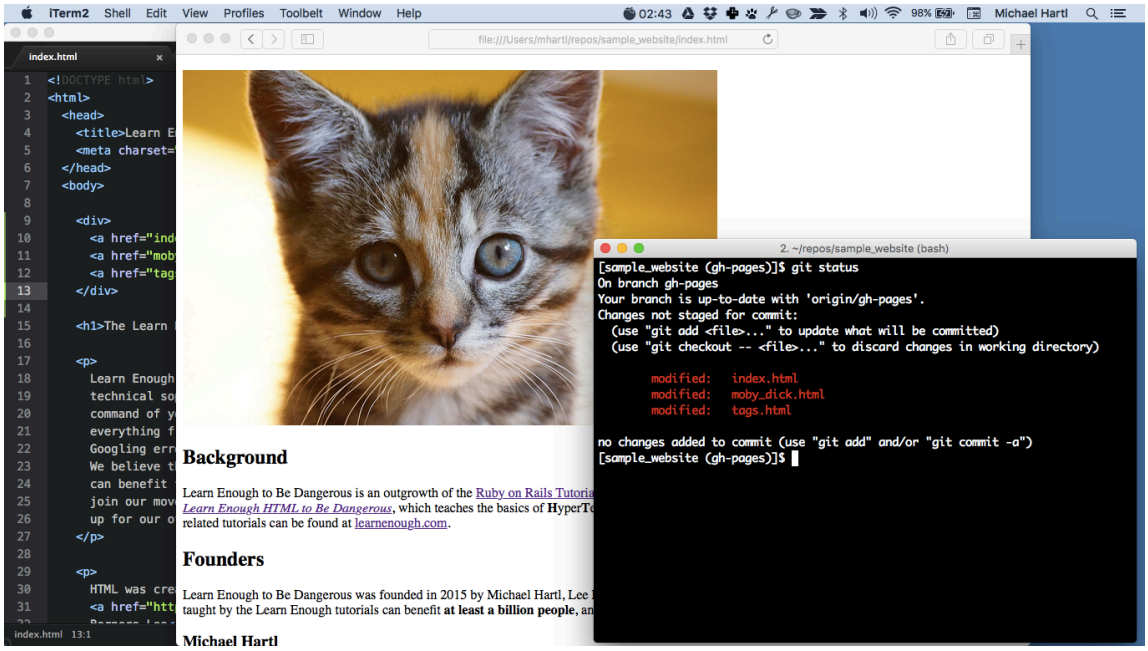


Figure 1.1: The tools of the trade (kitten not included).

If you're just getting started with HTML, the Developer Fundamentals sequence represents a little bit of overhead, but the benefits are enormous. To our knowledge, this combination of software development best practices and deploying to a live website is unique among introductory HTML tutorials, and gives you a tremendous advantage both when collaborating with others and when taking your skills to the next level by learning to build more complicated sites.

Learn Enough HTML to Be Dangerous focuses on core HTML, starting with a "hello, world!" page that we'll deploy to production (!) in Chapter 1. We'll then fill in the *index page* with formatted text, links, and images in Chapter 2, expanding it into a multiple-page site with more advanced features like tables and lists in Chapter 3. Finally, we'll add some inline styling in Chapter 4, which will allow us to see the effect of simple style rules on plain HTML elements.

The resulting site will be functional, but we'll run into several important limitations imposed by working with raw HTML. This will set the stage for the next Learn Enough tutorial, *Learn Enough CSS & Layout to Be Dangerous*,

which creates a fully modern website using *Cascading Style Sheets* (CSS) to separate the design of the site from its HTML structure, while covering site layouts and advanced styling as well.

1.1 Introduction

Underneath every website, no matter how simple or complex, you will find HTML. In this tutorial, by creating and deploying a simple but real website, we'll gain an understanding of the underlying structure that every site uses to organize and display content online.

As a technology standard, HTML has been constantly evolving ever since its introduction in 1993 by Tim Berners-Lee, the original "web developer" (Figure 1.2).[1] Nowadays, the specification of what's in HTML and what isn't is managed by the World Wide Web Consortium (W3C). The latest public release, which is what we will be using in this tutorial, is HTML5 (that is, version 5 of HTML). The companies that create web browsers take the specs from the W3C and implement the behaviors that are expected when the browser comes across any of the allowed formatting, such as **making text bold** or changing its color (or even **doing both** at the same time).

[1] Image retrieved from https://en.wikipedia.org/wiki/Tim_Berners-Lee#/media/File:Sir_Tim_Berners-Lee.jpg on 2016-01-12. Copyright © 2014 by Paul Clarke and used unaltered under the terms of the Creative Commons Attribution-ShareAlike 4.0 International license.

Figure 1.2: Sir Tim Berners-Lee, the original web developer.

Fortunately, we won't need to get into a lot of specifics or worry about what has changed from version to version. Just know that new features are being added regularly to expand browser functionality and modernize the technology. Common elements, including the ones we'll be covering in this tutorial, haven't changed much since the beginning, but that doesn't mean that they will always be safe—the HTML spec is a constantly evolving creature being assembled by a committee (Figure 1.3).[2] We'll discuss some practical effects of this in Section 1.2.

[2]Original image retrieved from https://www.flickr.com/photos/ginsnob/2099458654 on 2016-01-07. Copyright © 2007 by Chris Palmer and modified under the terms of the Creative Commons Attribution-ShareAlike 2.0 Generic license. Modified image copyright © 2016 by Lee Donahoe and released under the Creative Commons Attribution-ShareAlike 2.0 Generic license.

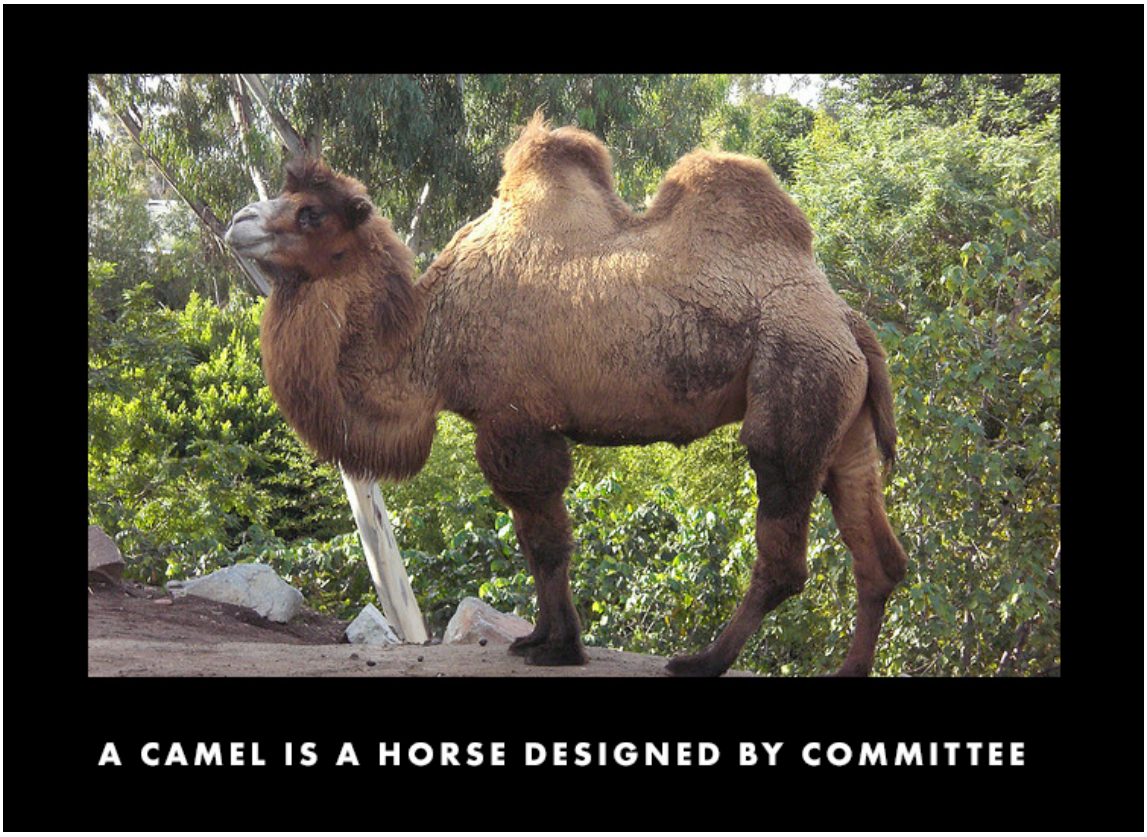


Figure 1.3: HTML in animal form.

1.2 HTML tags

As the name **H**yper**T**ext **M**arkup **L**anguage indicates, HTML is a *markup* language, not a programming language. HTML allows a web author to organize and define how content should be displayed, which means it can do things like add text formatting; make headings, lists, and tables; and include images and links. You can think of an HTML file as an ordinary written document that has been carefully annotated by the author. Some of the notes might highlight parts of the text, some might include an image that has been paper-clipped to the document, and others might tell you where to find additional information.

The "HyperText" part of the HTML acronym refers to the way links on the

Web allow you to move from one document to another in a non-linear fashion. For example, if you are reading the Wikipedia article on HTML and see a highlighted link to a related topic like CSS, you can click on that link and be taken immediately to the other article. It also allows a document like this one to link to Wikipedia. (You might notice that external links in this document open in a new browser tab. We'll learn how to do this ourselves in Section 3.3.)

Technologically, hypertext is a great improvement over non-linked documents, as it eliminates the need to flip or scroll through pages of content to find what you are looking for. These days, the ability to link between documents is something that we all take for granted, but when the HTML specification was created it was an innovation important enough to be included in the name of the technology.

HTML source is *plain text*, which makes it ideal for editing with a text editor (as discussed in *Learn Enough Text Editor to Be Dangerous*). Instead of using the convenient but inflexible What You See Is What You Get (WYSIWYG) approach of word processors, HTML indicates formatting using special *tags* (Figure 1.4),[3] which are the text annotations alluded to above.

[3]Image retrieved from https://www.flickr.com/photos/jdhancock/3814523970 on 2016-01-09. Copyright © 2009 by JD Hancock and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

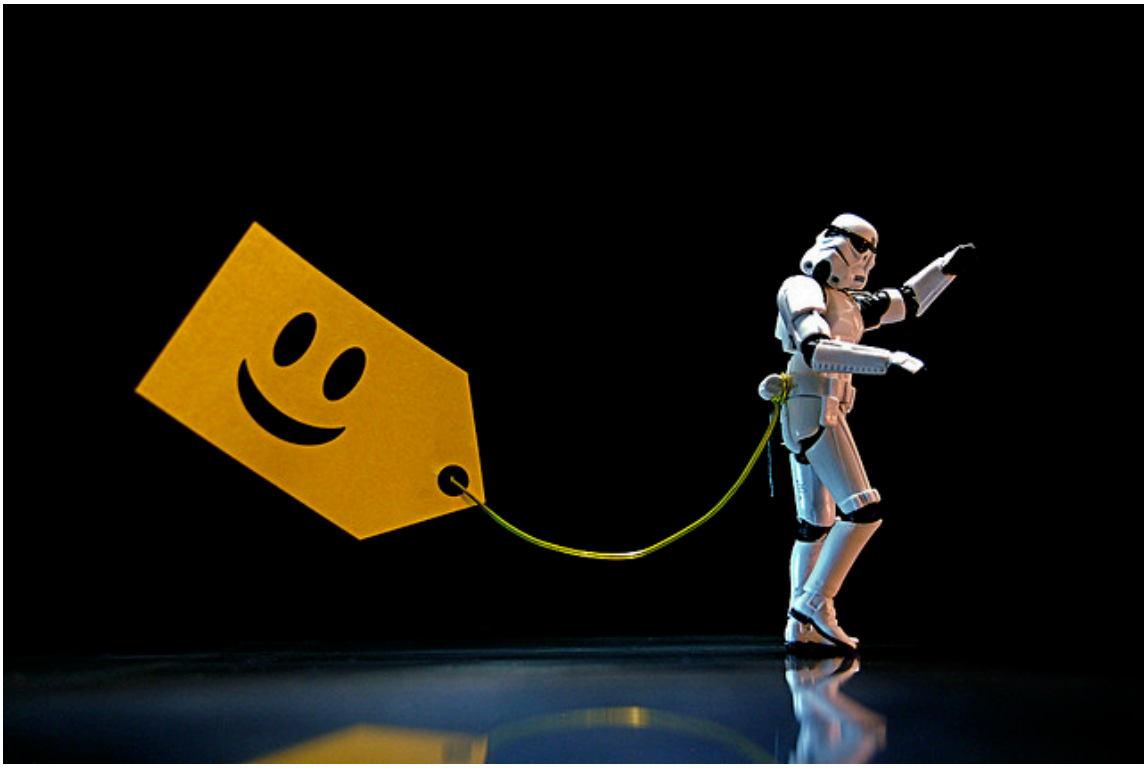


Figure 1.4: HTML uses tags for everything.

As we'll see, HTML supports more than one kind of tags, but the most common kind consist of strings (sequences of characters) enclosed in *beginning* and *ending* tags, like this:

```
<strong>make them strong</strong>
```

Figure 1.5 illustrates the detailed anatomy of this typical tag, including the name of the tag (**strong**, in this case), angle brackets, and a forward slash (**/**).

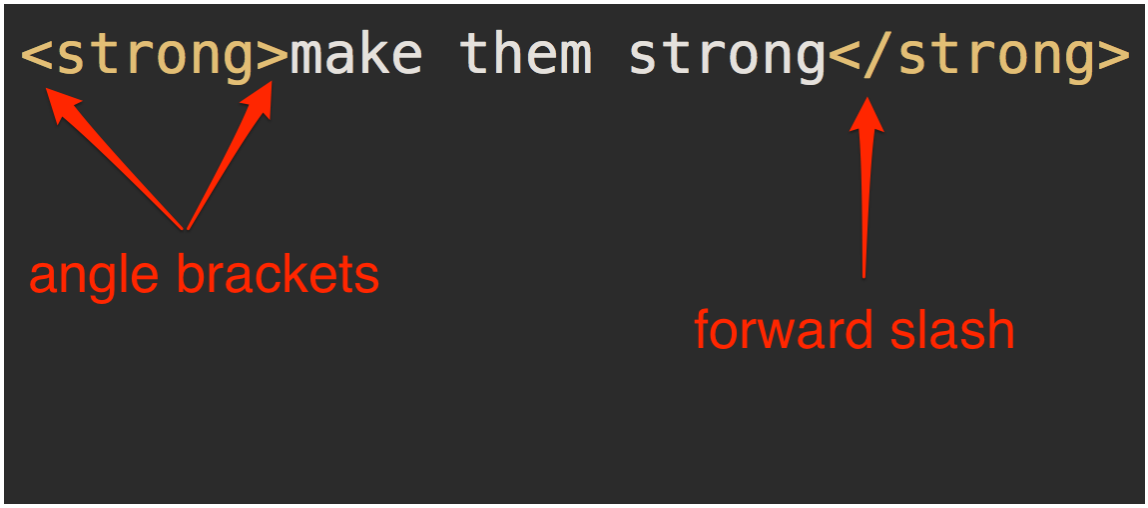


Figure 1.5: The anatomy of a typical HTML tag.

Although HTML tags are not visible to the end-user (i.e., the person viewing your website), they do give the web browser suggestions about how to format the content and how it should be displayed on the page. A simple example using the **strong** tag appears in Listing 1.1.

Listing 1.1: A string with HTML tags in the text.

```
I am a string about things. Some of those things are more important than others,
and I want to draw attention to them, so I will <strong>make them
strong</strong> so that they stand out among their less spectacular neighbors.
```

Most browsers render the **strong** tag as **boldface text**, so in a typical browser Listing 1.1 would appear something like this:

> I am a string about things. Some of those things are more important than others, and I want to draw attention to them, so I will **make them strong** so that they stand out among their less spectacular neighbors.

Note that, even though the contents of the **strong** tag are broken across a line in Listing 1.1, the browser ignores this extra space, and formats the string as a continuous line of text.

By the way, HTML does support a **b** (bold) tag in addition to **strong**, but over the years HTML has moved away from tag names that specify formatting (i.e., "make the text **bold**") and toward names that focus on *meaning*—or, in fancier terms, *semantics*—leading to an emphasis on so-called *semantic tags* (Box 1.2). For example, the semantic tag **strong** indicates that the enclosed text should be made to look "strong" somehow, letting the browser decide exactly how to do it.

Box 1.2. The cautionary (semantic) tale of <b> and <i>

When HTML was first created, the Internet made funny noises when you connected to it, and you paid for the connection in time or by amount of data sent. Those limitations meant that brevity was an important consideration when deciding on tags, and the whole endeavor was so new that there wasn't as much thought about the meaning the tags conveyed. So short tags were popular, and getting everything to display correctly to people was the end-goal.

As a result of this focus on concision, the original way to make text bold was to use the `b` tag (`<b>...</b>`), and to make text italic it was the `i` tag (`<i>...</i>`). This worked just fine (in fact, it still works even today), and no one was confused.

What some developers began to notice was that HTML tags were being defined only by how the content inside should be displayed in a browser, rather than by the meaning of the content. That's fine for people looking at content with good ol' eyeballs, but not so good for automated systems that might be rapidly scanning web pages and need to infer what the content wrapped in different HTML tags actually means.

To address this issue, a movement started that pushed for new tags based on semantic meaning instead of on appearance, thereby giving rise to the current preferred method for indicating bold or italicized text with `strong` (`<strong>...</strong>`) and `em` (`<em>...</em>`, for "emphasized"), respectively. The idea here is that the intent behind making text bold is to make it strongly **stand out** from the rest of the content, and the intent behind italicizing text is to *show emphasis*.

This might seem like a subtle difference, but semantic tags are used for a lot more than just defining strong or emphasized text. Semantic HTML tags will be further discussed in the *Learn Enough CSS & Layout to Be Dangerous*, where we will cover tag conventions and page layout in greater depth.

At this point, we've covered the conceptual core of HTML: HTML consists of text content wrapped in tags, which organize or indicate a change in the display of that content.

But the Devil, as they say, is in the details… and HTML has a whole lot of details.

1.2.1 Exercises

Note: Unlike most other Learn Enough tutorials, the results of some exercises will appear in future screenshots.

1. Identify all the tags in Listing 1.2. Notice that you don't have to know what a tag does to be able to identify it correctly. (This is a good example of technical sophistication (Box 1.1).)
2. Some HTML tags don't contain any content, and instead are known as *void elements*, also called *self-closing tags*. Which tag in Listing 1.2 is a void element?
3. HTML tags can be *nested*, which means that one tag can be put inside another. Which tags in Listing 1.2 are nested? Don't include any self-closing tags.

Listing 1.2: Shall I compare thee to a summer's day?

```
<p>
  William Shakespeare's <em>Sonnets</em> consists of 154 poems, each fourteen
  lines long (three
  <a href="https://en.wikipedia.org/wiki/Quatrain">quatrains</a>
```

```
  followed by a rhyming
  <a href="https://en.wikipedia.org/wiki/Couplet">couplet</a>).
  <strong>Sonnet 18</strong> is perhaps the most famous, and begins like this:
</p>
<blockquote>
  <p>
    Shall I compare thee to a summer's day?<br>
    Thou art more lovely and more temperate.<br>
    Rough winds do shake the darling buds of May,<br>
    And summer's lease hath all too short a date.
  </p>
</blockquote>
```

1.3 Starting the project

Now that we know the basic structure of markup and tags, it's time to get started with the project that will serve as our sample website for learning HTML. The sample project is a mock informational website, whose main page talks a little about this tutorial, the company behind it, and HTML itself. As we develop the home page and two ancillary pages, we'll see how to use a wide variety of HTML tags, while also showing you how to make the kind of public-facing site that you could use for your own work or as a service to other people. Although some of the information on the site is about Learn Enough to Be Dangerous, ultimately it is about *you*.

We'll begin by following the same steps used in *Learn Enough Git to Be Dangerous*, so this section will also serve as a review of how to use Git. (If you don't have Git proficiency at the level of *Learn Enough Git to Be Dangerous*, we recommend you read that tutorial at this time.) As in the Git tutorial, the result here will be that we can deploy our sample HTML site to the live Web using GitHub Pages (Box 1.3).

Box 1.3. GitHub Pages

Once you have an account at GitHub (and have verified your emailed address), you can use a free feature called *GitHub Pages* that allows you to host simple HTML sites for free on GitHub's infrastructure.

This is a major advance compared to the bad old days of the early Web. For example, if this were 1999, you'd not only have to pay money for the hosting, but you'd also be on the hook for the cost of transferring the data to the people visiting your site. For sites with even moderate traffic, the bills could add up fast.

Nowadays, we have many better options, GitHub Pages among them. Not only is GitHub Pages free, but it is incredibly easy to use. By updating a single configuration setting (described below), we can arrange for GitHub Pages to serve our website right from the `master` branch. As described in the free tutorial *Learn Enough Custom Domains to Be Dangerous*, you can even set up your GitHub Pages site to use a custom domain (like www.example.com). The result is an industrial-grade web presence with built-in Git backups, high performance, and zero cost.

That's a combination that's hard to beat!

We'll get started by making a directory and an initial repository for our sample website. First, open a terminal window[4] and make a directory called **`sample_website`**:[5]

```
$ mkdir -p repos/sample_website
```

Next, **`cd`** into the directory and **`touch`** the file for the main page of the site, which should be called **`index.html`**:

```
$ cd repos/sample_website
$ touch index.html
```

[4] *Note for Mac users*: Although it shouldn't matter in *Learn Enough HTML to Be Dangerous*, it is recommended that you use the Bourne-again shell (Bash) to complete this tutorial. To switch your shell to Bash, run **`chsh -s /bin/bash`** at the command line, enter your password, and restart your terminal program. Any resulting alert messages are safe to ignore. See the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials" for more information.

[5] The command **`mkdir -p`** is covered in *Learn Enough Git to Be Dangerous*. Every command we use in *Learn Enough HTML to Be Dangerous* is covered somewhere in a previous Learn Enough tutorial (unless otherwise indicated), so we recommend you search through the previous tutorials if you run across any commands that don't look familiar.

Then initialize the repository:

```
$ git init
$ git add -A
$ git commit -m "Initialize repository"
```

The reason we created a file using **`touch`** is because Git won't initialize an empty repository.[6] The reason we've called it **`index.html`** is because that's the default filename for "home" pages on the Web, and most sites will automatically serve up **`index.html`** when you hit the bare domain. In other words, when you point a browser at example.com, the server will automatically show you example.com/index.html. (Those links work, by the way; amazingly, the HTML standard specifically reserves the site example.com for examples just like this one!)

With the repo initialized, we're now ready to push our (nearly) empty repo up to GitHub. As in *Learn Enough Git to Be Dangerous*, you should go to github.com, log in if necessary, and then create a new repository using the name **`sample_website`** and the description "A sample website for Learn Enough HTML to Be Dangerous", as shown in Figure 1.6 and Figure 1.7.[7]

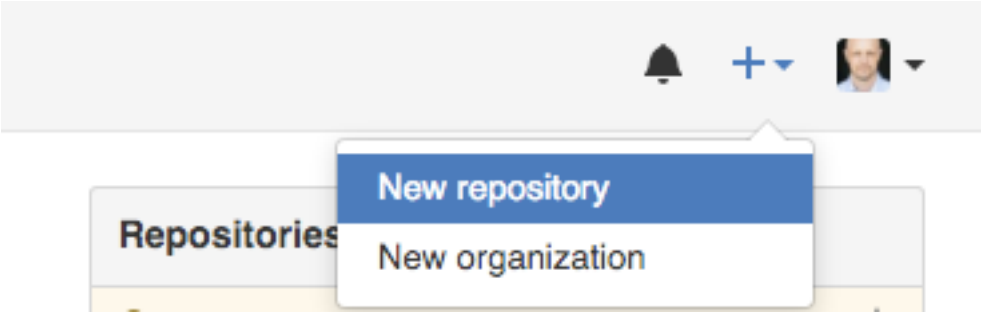


Figure 1.6: The GitHub new repository menu.

[6] As mentioned in *Learn Enough Command Line to Be Dangerous*, the **`touch`** technique is a personal favorite, but the file doesn't have to be empty; for example, **`echo hello > index.html`** would also have worked.

[7] GitHub is constantly updating its user interface (UI), so in these and other figures you may notice slight differences compared to what you see in your browser. Use your technical sophistication (Box 1.1) to resolve any discrepancies.

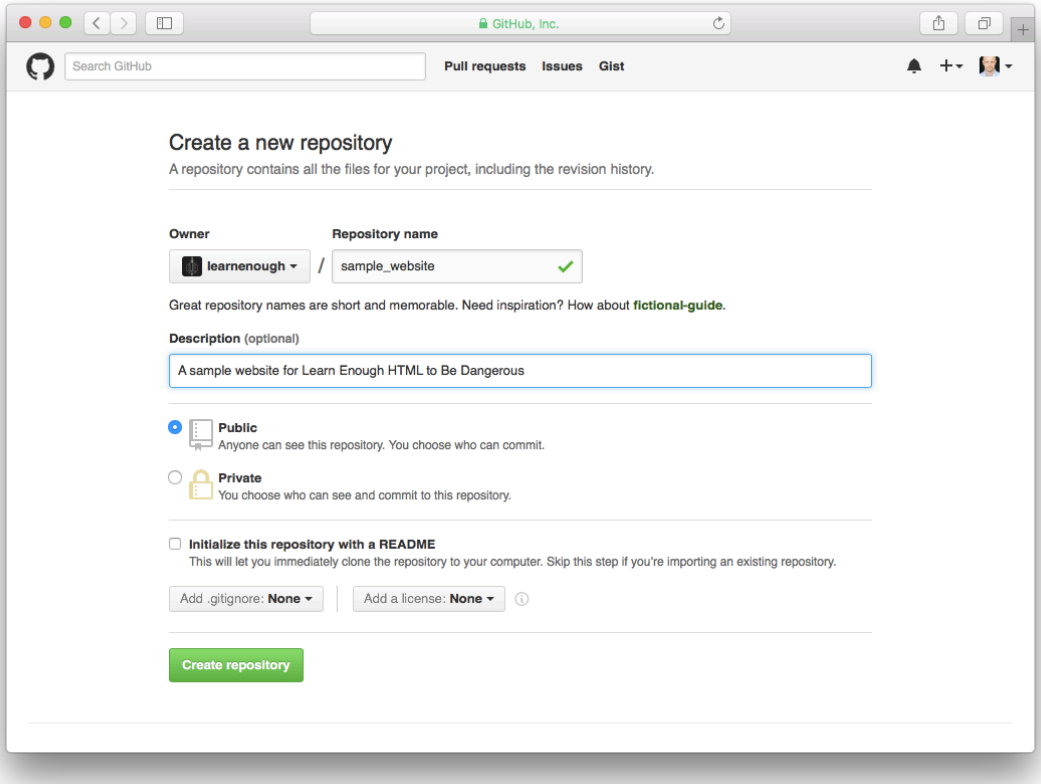


Figure 1.7: Creating a new GitHub repo for our website.

After creating the remote repo, you should set the *remote origin* as the main URL for the repo, which you can find as shown in Figure 1.8 (or just copy from the section "…or push an existing repository from the command line" on the GitHub setup page):

```
$ git remote add origin <repo url>
$ git push -u origin master
```

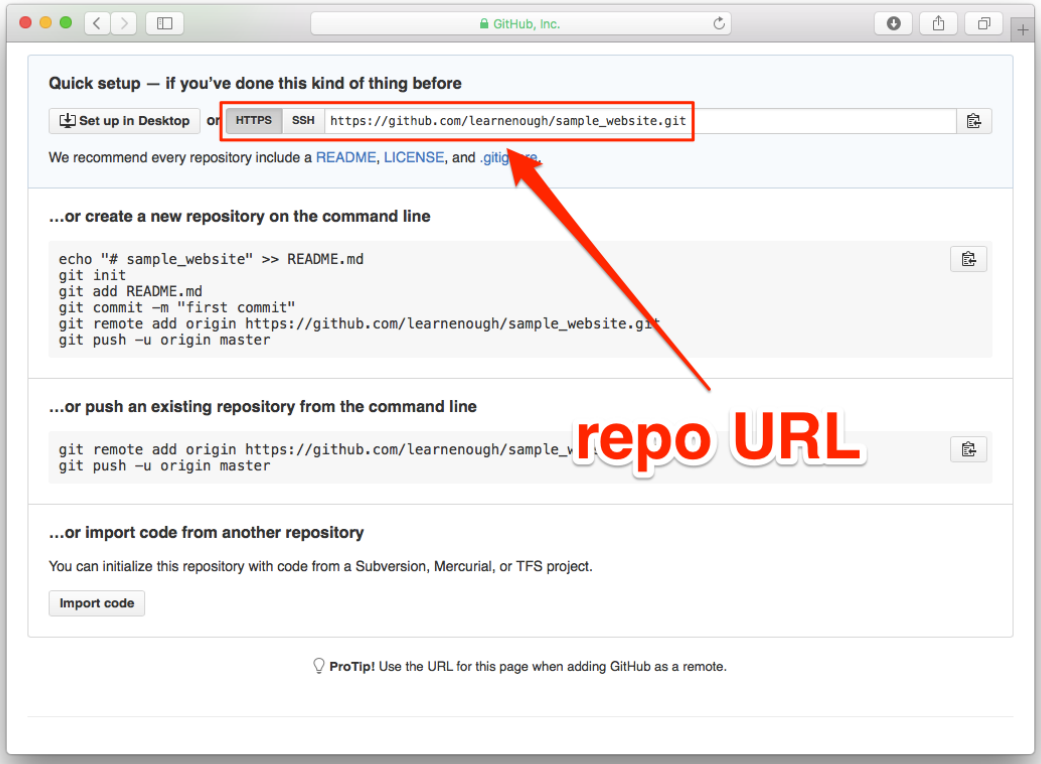


Figure 1.8: Finding the repo URL.

At this point, you should follow the instructions for serving your website from the **`master`** branch using GitHub Pages, as discussed in *Learn Enough Git to Be Dangerous* and shown in Figure 1.9 and Figure 1.10.

With that, our sample website is now live on the Web via GitHub Pages! Its location is given by a github.io URL based on the username and repo name (Listing 1.3).

Listing 1.3: The template for a GitHub Pages URL.

```
https://<username>.github.io/<repo_name>
```

For example, the Learn Enough version of the sample website lives at the URL

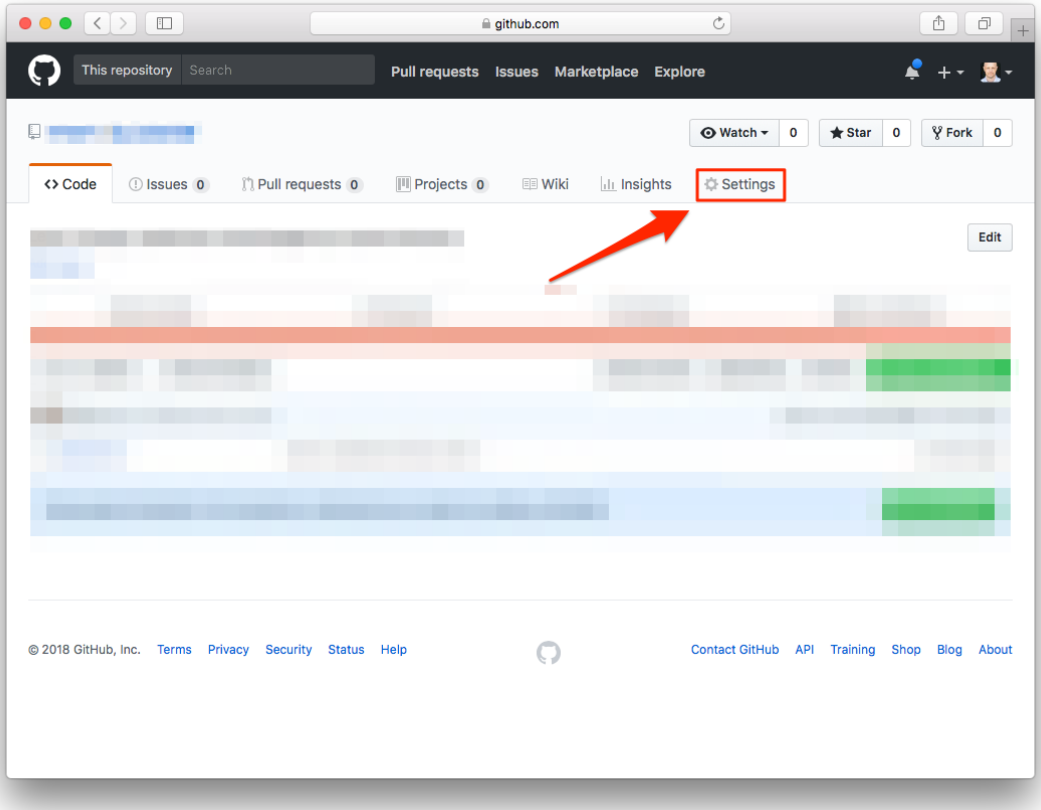


Figure 1.9: The settings for a GitHub repository.

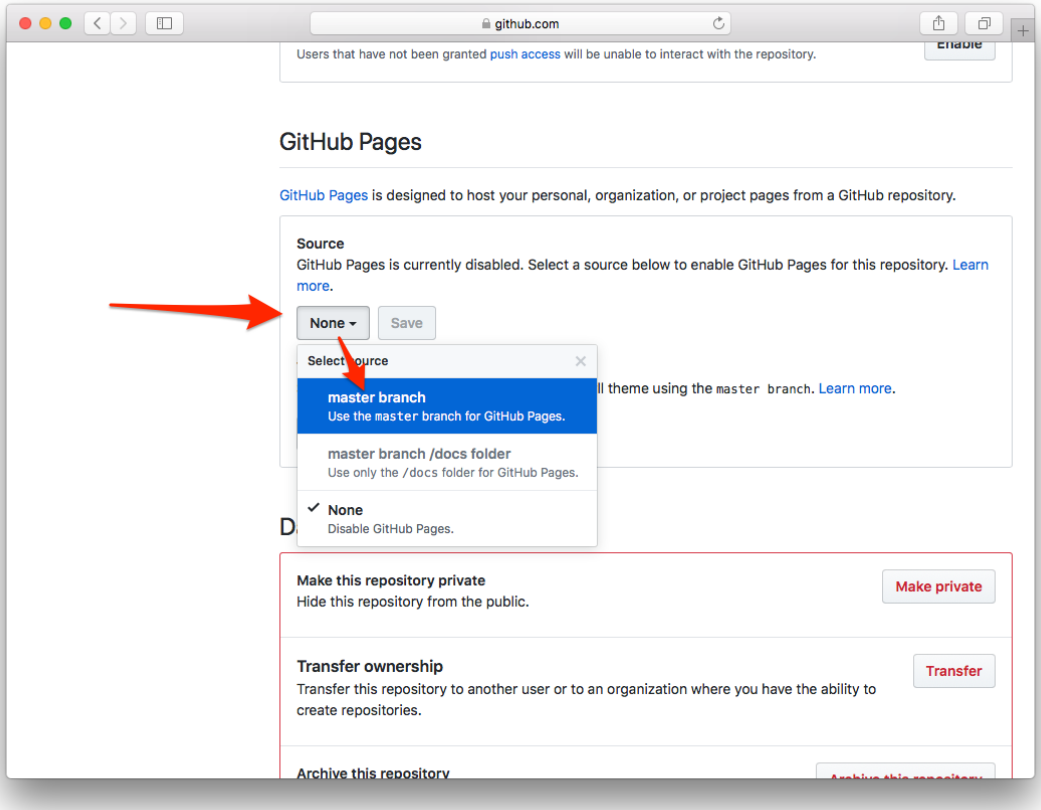


Figure 1.10: Serving our website from the **`master`** branch.

https://learnenough.github.io/sample_website_reference_implementation

Incidentally, that means it can be cloned from here for future reference:[8]

https://github.com/learnenough/sample_website_reference_implementation

If you visit your version of this site, it should resolve properly, and it should even automatically serve up the contents of `index.html`. Because those contents are empty, though, the current appearance is a little underwhelming (Figure 1.11). We'll take our first steps toward changing this sad state of affairs in Section 1.4, and then we'll knock it up a notch starting in Chapter 2.

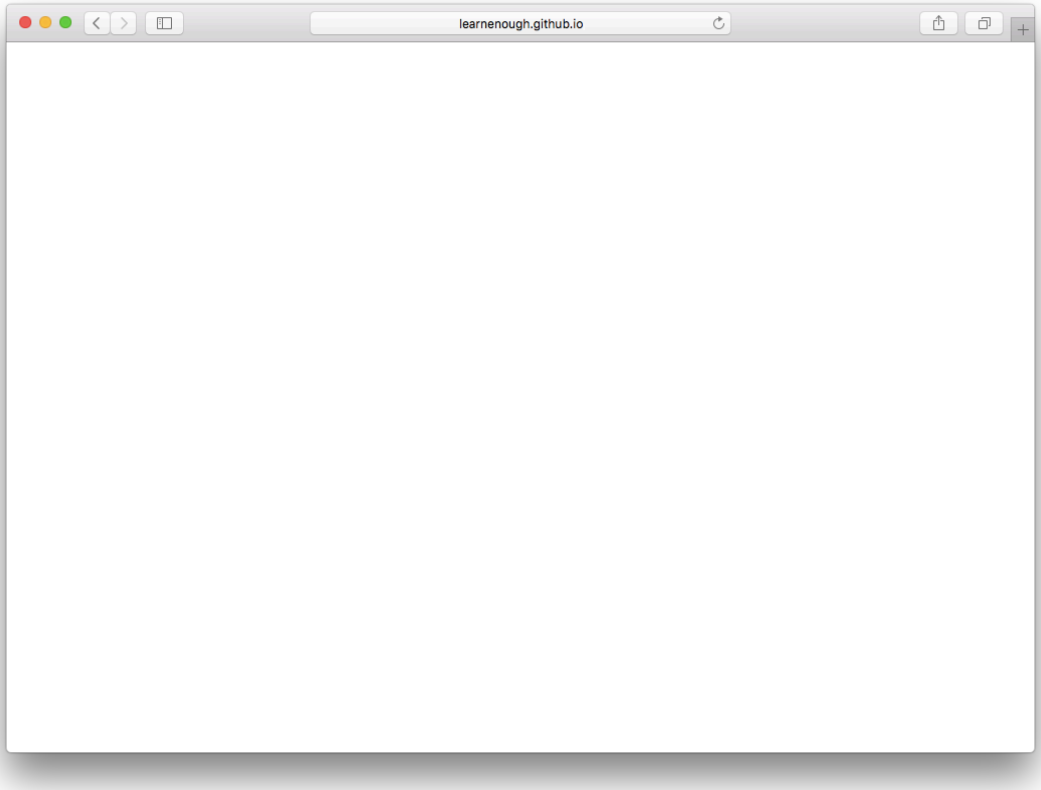


Figure 1.11: The current underwhelming appearance of our live website.

[8] Keeping the reference website in sync with the main tutorial is a challenging task, so reports of any discrepancies are gratefully received. You can send such reports to michael@learnenough.com.

1.3.1 Exercises

1. Add and commit a file called **README.md**, taking care to use at least a few Markdown tags. What is the result at GitHub?

2. What happens if you visit **<username>.github.io/<repo_name>/README.md** in a browser? What does this imply about including sensitive information in a public website repo?

1.4 The first tag

In order to initialize the Git repository in Section 1.3, we needed only an empty **index.html** file, but of course our sample site will eventually have much more than that. In this section, we'll begin by adding some content in a single tag—just enough to give us a site to view, commit, and deploy. That's a huge accomplishment, though, and will serve as an essential foundation for what follows.

Now that an empty index page has been created, you should open **index.html** using your favorite editor, which for the purposes of this tutorial we'll assume is Atom. It's possible to open the file directly using **File > Open**, but (as noted in *Learn Enough Text Editor to Be Dangerous*) these days all the cool kids open the full HTML project directly at the command line (Figure 1.12):[9]

```
$ atom .
```

(Recall from *Learn Enough Command Line to Be Dangerous* that **.** ("dot") refers to the current directory.)

[9] Image retrieved from https://www.flickr.com/photos/usfwshq/8247653540 on 2016-09-10. Copyright © 2012 by the U.S. Fish and Wildlife Service and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 1.12: All the cool kids open HTML projects directly at the command line.

Even though we have only one file for now (and possibly a README (Section 1.3.1)), opening the full project is a good habit to cultivate, since it allows us to easily open and edit multiple files in the same project. We'll put this technique to good use when we start making additional pages in Section 3.1.

At this point, we're ready to fill the index file with some content, which should consist of the phrase "Hello, world!" (Listing 1.4) enclosed in the *paragraph tag* `p`. Note that the `p` tag has exactly the same form as the `strong` tag (Figure 1.5).

Listing 1.4: A short paragraph with the contents "Hello, world!".
index.html

```
<p>Hello, world!</p>
```

We see in Figure 1.13 that Atom automatically highlights the HTML source, which it knows to do because of the `.html` extension on the filename. This *syntax highlighting* is irrelevant to the computer—in fact, it takes place purely in the editor and doesn't have anything to do with `index.html` itself—but it makes it easier for humans to distinguish the difference between tags and content. (This is also why we use syntax highlighting in this tutorial's code listings.)

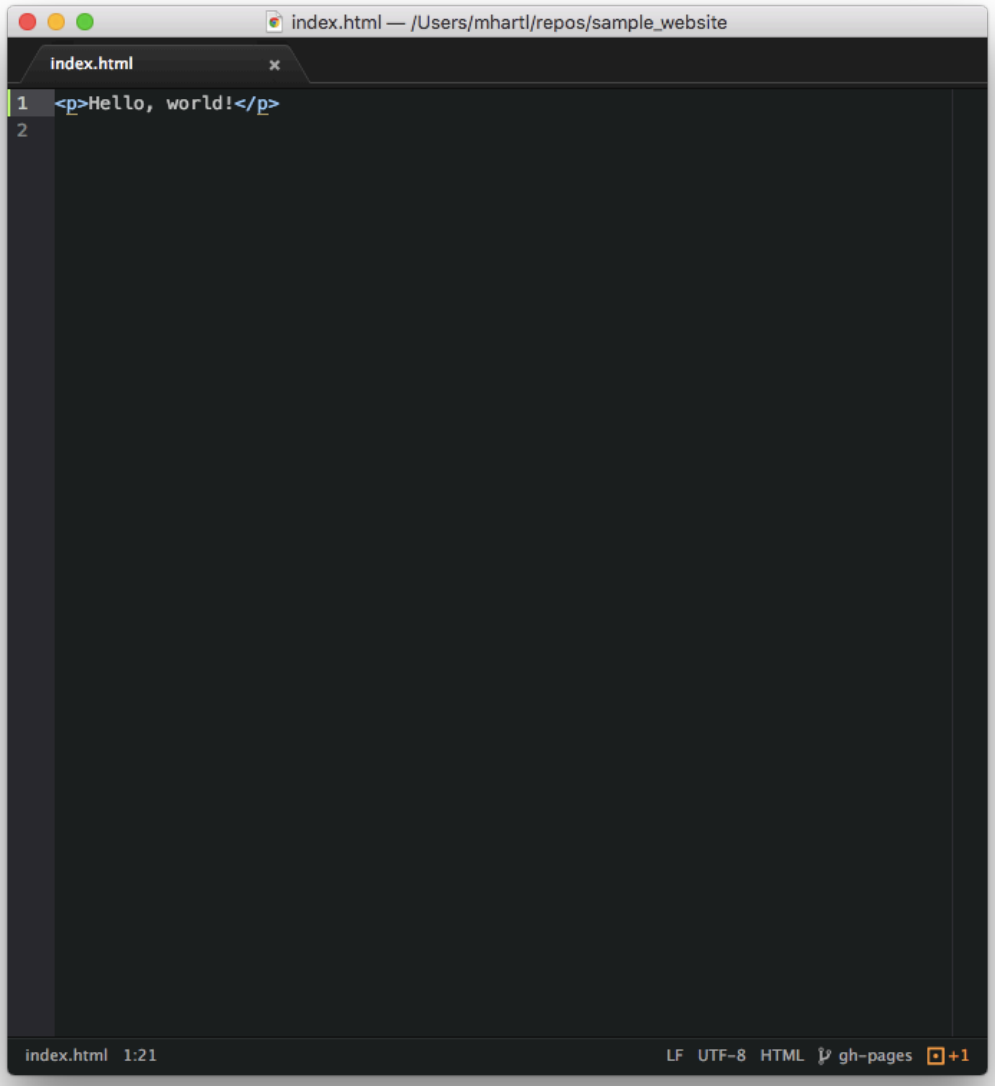


Figure 1.13: "Hello, world!" in a text editor.

Having made a change to add content to **`index.html`**, let's view the result in a browser. On macOS, you can do this using the **`open`** command:

```
$ open index.html     # macOS only
```

On many Linux systems, you can use the similar **xdg-open** command:

```
$ xdg-open index.html     # Linux only
```

A technique that works on almost any system is to view the **sample_website** directory in graphical file browser and double-click the filename (Figure 1.14).

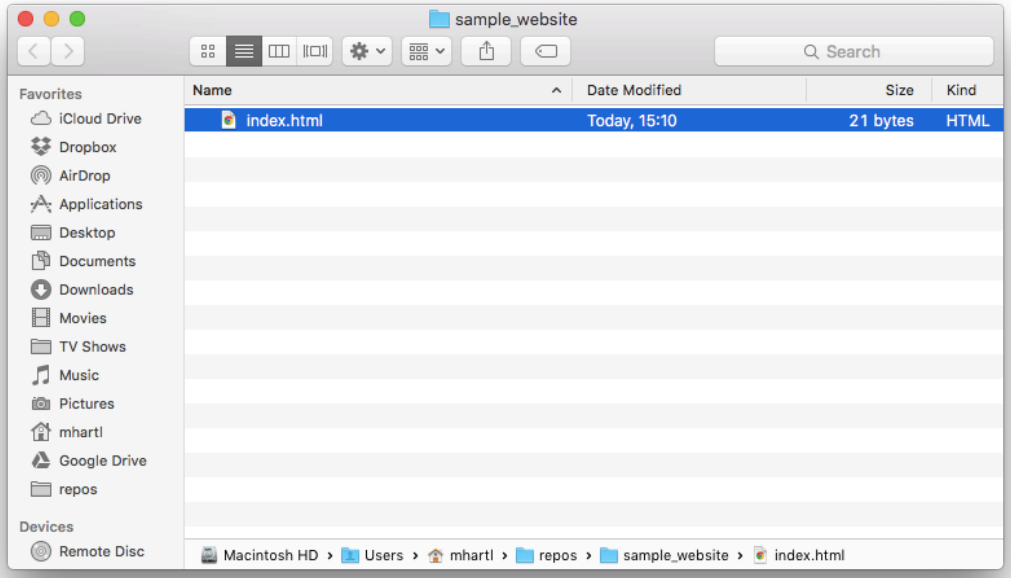


Figure 1.14: Double-clicking **index.html** should open it in the default browser.

No matter how you do it, the result should be to open **index.html** in the default browser on your system, which should appear something like Figure 1.15.

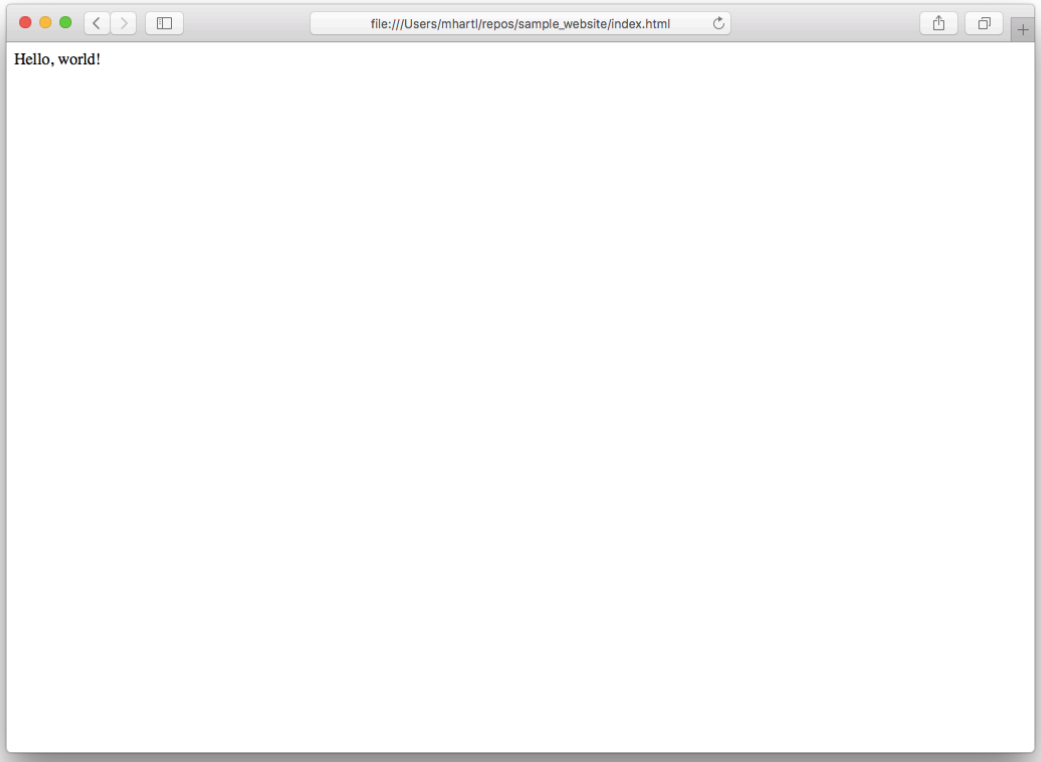


Figure 1.15: The index page in a local browser.

Note that the "URL" in this Figure 1.15 will be a *local* file, like this:

```
file:///Users/mhartl/repos/sample_website/index.html
```

This is because the index page is on our local system, and hasn't yet been deployed to the live Web.

We know how to remedy this, though—commit our changes to the local Git repository and push up to GitHub Pages:

```
$ git commit -am "Add a short paragraph"
$ git push
```

Upon refreshing the browser pointed at the sample website's GitHub Pages URL (Listing 1.3), the result should look something like Figure 1.16. (You may have to wait a few moments for GitHub Pages to load your site. This happens only the first time, and on subsequent requests the response will be lightning-fast.)

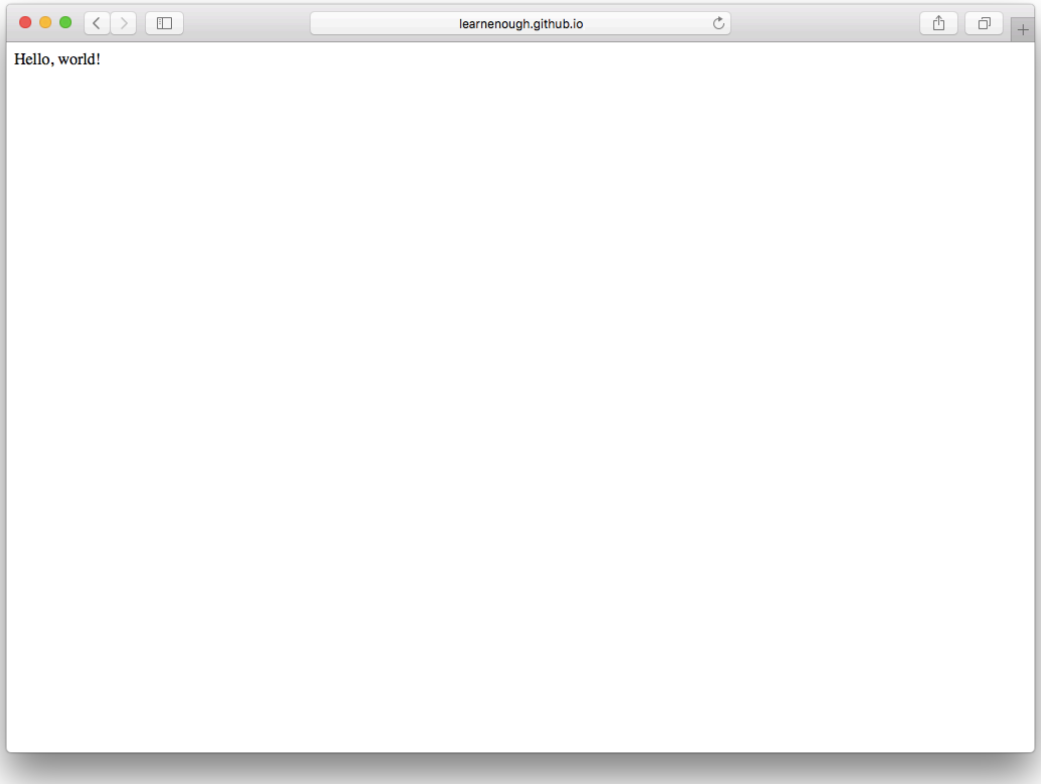


Figure 1.16: The index page on the live Web.

Although the appearance is identical to the local version in Figure 1.15, by inspecting the address bar you should be able to confirm that the URL is at github.io, which means that the page is now available on the live Web.

Congratulations! You've just published a production website.

1.4.1 Exercises

1. Replace the contents of `index.html` with the markup from Listing 1.2. Can you guess what the `a` tag does?

2. Use your browser's *web inspector* to inspect the source from the previous exercise. (Google for "<browser> web inspector" to learn how to use your browser's web inspector. Or just right-click.) Does it differ in any way from Listing 1.2?

1.5 An HTML skeleton

Although modern web browsers are highly fault-tolerant and will render simple HTML like Listing 1.4 just fine, it's dangerous to rely on this behavior. Indeed, we saw in *Learn Enough Git to Be Dangerous* that omitting a single tag resulted in the trademark character ™ not rendering properly. This is exactly the sort of thing that can go wrong when you don't use a fully valid HTML page. To avoid these sorts of problems, from now on all of our sample web pages will use valid HTML to ensure that they will render properly in the broadest possible range of browsers.

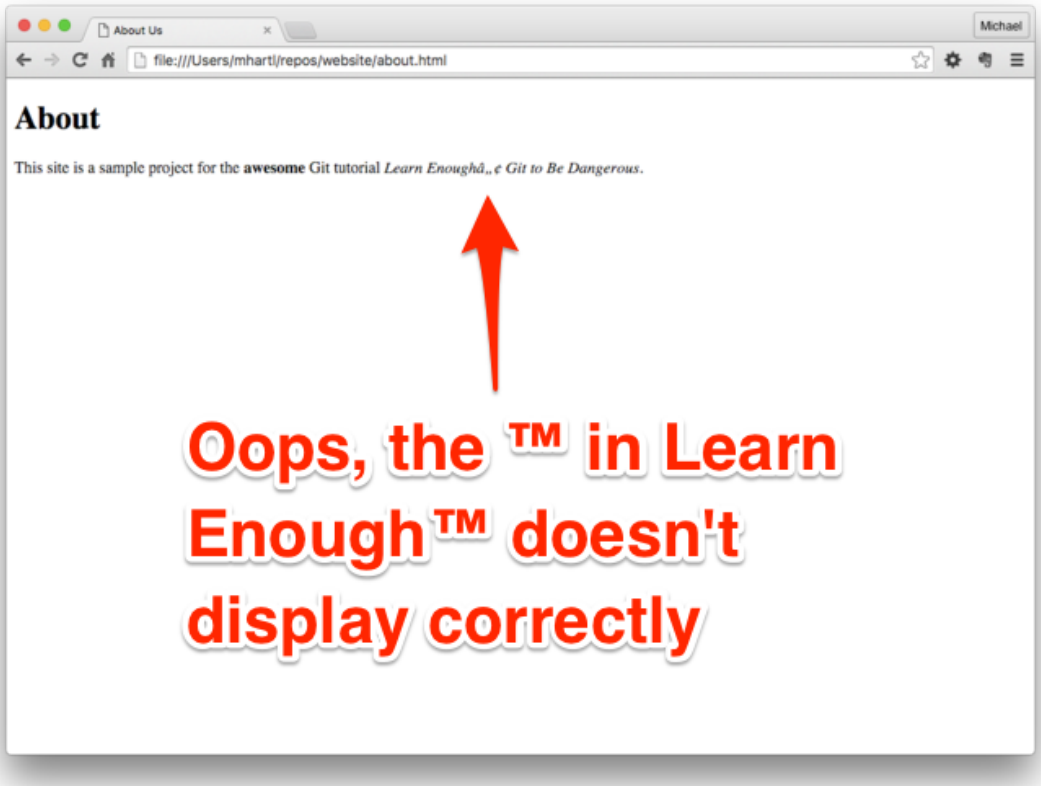


Figure 1.17: The broken About page from *Learn Enough Git to Be Dangerous*.

The prototypical HTML skeleton begins with an **`html`** tag containing two elements, a **`head`** and a **`body`**. These latter tags are *nested* inside the **`html`** tag, as follows:

```
<html><head></head><body></body></html>
```

Because this is hard to read, it's conventional to format the tags using spaces and newlines to make the structure more apparent at a glance:

```
<html>
  <head>
  </head>
  <body>
  </body>
</html>
```

Because HTML generally ignores extra space, it makes no difference in the appearance of the page, but this formatting makes it easier for us to understand the source of the document (Box 1.4).

Box 1.4. Formatting HTML

In order to make HTML easier to read, it's conventional to add extra spaces and newlines to make the document's structure clear at a glance. It might look a little strange at first, but it's a style convention in the development world that helps keep source code readable (and coders sane).

Generally speaking, HTML ignores extra space when displaying in a browser, so

```
<p>Hello, world!</p>
```

will look the same as

```
<p>Hello,
              world!

              </p>
```

It's a good practice to keep your HTML formatting tidy, and it's clear that the former is easier to read.

The readability of the markup can change with time, though, especially after the text grows to more than a couple of sentences, or when there are additional HTML elements nested inside. To keep our content straight and not lose track of tags, in these cases we'll need to format the code more rigorously.

There are no universal rules for formatting markup, but a good rule of thumb is to put new tags on their own line unless they fit easily on one, with lines inside those tags indented one level:

```
<p>Hello, world!</p>

<p>
  Lorem ipsum dolor sit amet, consectetur
  adipisicing elit, sed do eiusmod tempor
  incididunt ut labore et dolore magna aliqua.
  Ut enim ad minim veniam
</p>
```

The main exception to the new-line rule involves tags that modify text in a paragraph. For instance, most people do not add line breaks or indentation for elements that are inside lines of text (referred to as *inline elements*) like the `<strong>make them strong</strong>` example from Listing 1.1. We'll talk more about these types of inline elements, and how they differ from so-called *block elements*, in Section 3.2.

What constitutes an indentation "level" varies by developer, but we prefer the two-space convention shown above. Four spaces are also common, but in our experience this sends the content running off the right side of the page a little too fast. Some developers use tabs instead of spaces, but (at the risk of starting a holy war) we think this should be strenuously avoided. The main issue is that the display of tab characters is device-dependent, so markup that looks great in a text editor could look terrible using `less` at the command line.

It's important to make sure your editor is configured properly to use spaces (sometimes confusingly called *emulated tabs*) instead of genuine tabs. Refer to

Learn Enough Text Editor to Be Dangerous or use your technical sophistication (Box 1.1) to figure out how.

Finally, it's worth noting that many modern text editors include a way to format HTML automatically. For example, in Atom it's `Edit > Lines > Auto Indent`. This can be a big help when indenting larger HTML files, especially if they come from an outside source and aren't already nicely formatted.

The section of the HTML that is wrapped by **<head>** and **</head>** is a header container that defines *metadata*, a fancy word that just means data about data. This **<head>** section is not displayed to users in their browser window, and therefore gives developers the ability to tell the browser where to find other files (such as CSS and JavaScript) that will be used to properly display the page's content, *without* having that information show up in the actual content of the web page. (We'll cover more things that can be added to the HTML header starting in *Learn Enough CSS & Layout to Be Dangerous*.)

Meanwhile, the content inside **<body>** and **</body>** is what gets displayed to the browser. Every website you've ever seen consists of content inside an HTML **body** tag. Once we've defined the contents of the **head** tag, most of the modifications we'll make to the site will be inside the **body** tag.

To complete the skeleton, there are only two more required elements, one required and one optional but strongly recommended. First, we need to tell browsers what the *document type* is, and inside the **head** tag we need to define a nonempty **title**, as seen in Listing 1.5.

Listing 1.5: A nearly complete HTML skeleton.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
  </head>
  <body>
  </body>
</html>
```

Here the **DOCTYPE** is an irregular tag, and you should not spend even a millisecond worrying about its exact form. (Why is there an exclamation point before **DOCTYPE**? We have no idea.) Meanwhile, the **title** tag has exactly the same form as the **p** tag we saw in Section 1.4 (and the **strong** tag before that (Figure 1.5)).

As you can verify by using the W3C HTML validator, the page in Listing 1.5 validates as HTML5 (Figure 1.18). (An empty **title** would be invalid, but an empty **body** is fine. Also note that, although there are no *errors* on the page, there is a *warning* regarding the absence of a **lang** (language) attribute; we'll address this detail in Section 3.3.1.)

Figure 1.18: The HTML in Listing 1.5 is valid but incomplete.

On the other hand, we learned in *Learn Enough Git to Be Dangerous* (Figure 1.17) that we need one more thing: we need to tell the browser which *character set* to use so that it can handle the expanded range of characters (called Unicode), which includes symbols like ™ and ©, accented characters (as in *voilà*), etc. We can do this by adding the **meta** tag to the **head**, as shown in Listing 1.6.

Listing 1.6: Adding a `meta` tag to define the character set.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
    <meta charset="utf-8">
  </head>
  <body>
  </body>
</html>
```

By the way, the `meta` tag is a special kind of tag called a *void element*, and doesn't have a closing tag. Because of this, void elements are also called *self-closing tags*.

With that, the skeleton is complete (Figure 1.19),[10] and is listed again for reference in Listing 1.7.

[10]Image retrieved from https://www.flickr.com/photos/puuikibeach/6168133195 on 2016-01-14. Copyright © 2011 by davidd and modified under the terms of the Creative Commons Attribution 2.0 Generic license. Modified image copyright © 2016 by Michael Hartl and released under the Creative Commons Attribution 2.0 Generic license.

Figure 1.19: A complete HTML skeleton.

Listing 1.7: The skeleton for a basic HTML document.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
    <meta charset="utf-8">
  </head>
  <body>
  </body>
</html>
```

Because of the importance of this HTML skeleton, let's review its elements line by line:

1. The doctype declaration
2. Opening **`html`** tag
3. Opening **`head`** tag
4. Opening and closing **`title`** tags (with the content of the page title)
5. The **`meta`** tag defining the character set
6. Closing **`head`** tag
7. Opening **`body`** tag
8. Closing **`body`** tag
9. Closing **`html`** tag

Combining the original paragraph from Listing 1.4 with the skeleton in Listing 1.7 gives us the code for the first valid HTML page in our sample website, as shown in Listing 1.8.

Listing 1.8: A valid "Hello, world!" page.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title</title>
    <meta charset="utf-8">
  </head>
  <body>
    <p>Hello, world!</p>
  </body>
</html>
```

Note in Listing 1.8 that we've placed the paragraph from Listing 1.4 inside the **`body`** tag, as required by the HTML standard.

After refreshing the browser, the result of Listing 1.8 is virtually the same as we saw in Figure 1.16. The only visible difference in the body of the page is a small amount of additional space around the paragraph, as shown in Figure 1.20.

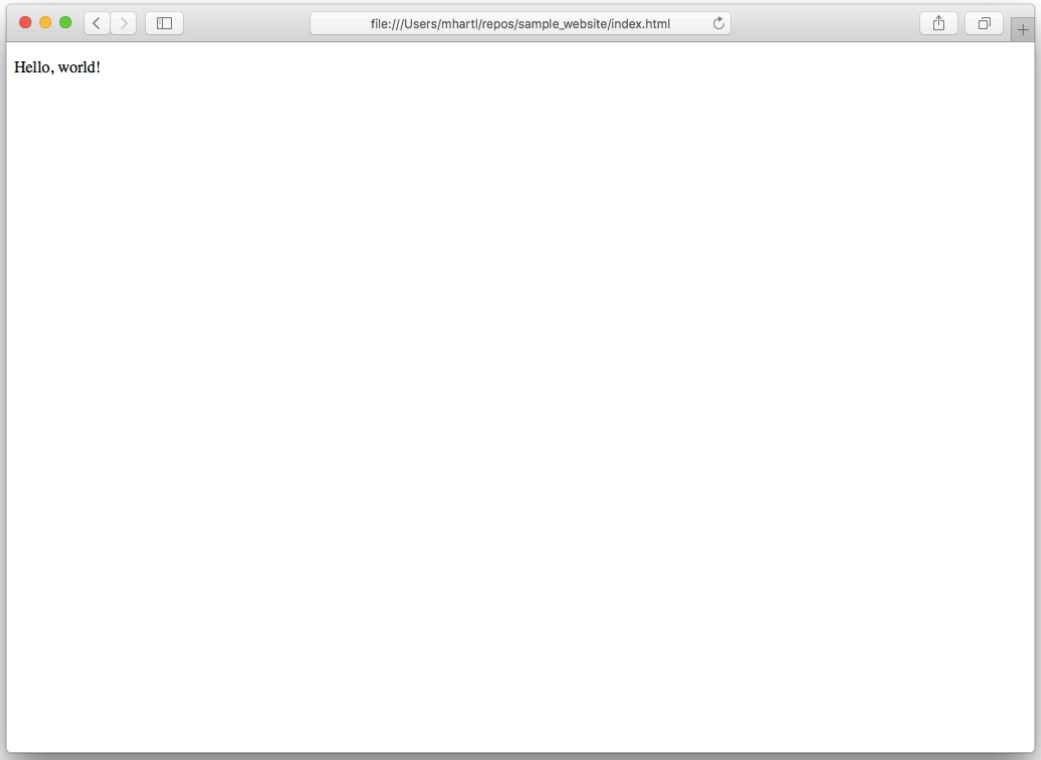


Figure 1.20: A valid "Hello, world!" page.

A second, browser-dependent difference involves the page title, which some browsers display in the default tab (Figure 1.21), and others don't show unless you have a second tab in addition to the first (Figure 1.22). In any case, the page title is needed even if it's not displayed, as it's required by the HTML standard, and is important for screen readers and the web spiders used by search engines to index the Web.

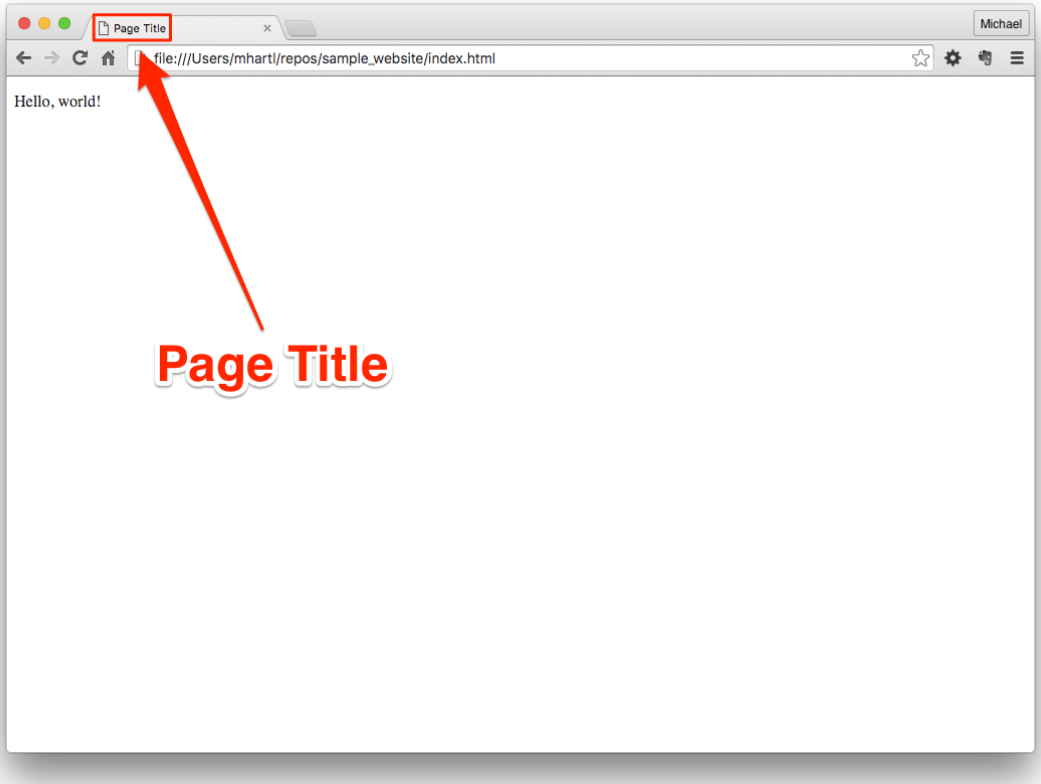


Figure 1.21: The page title in Chrome.

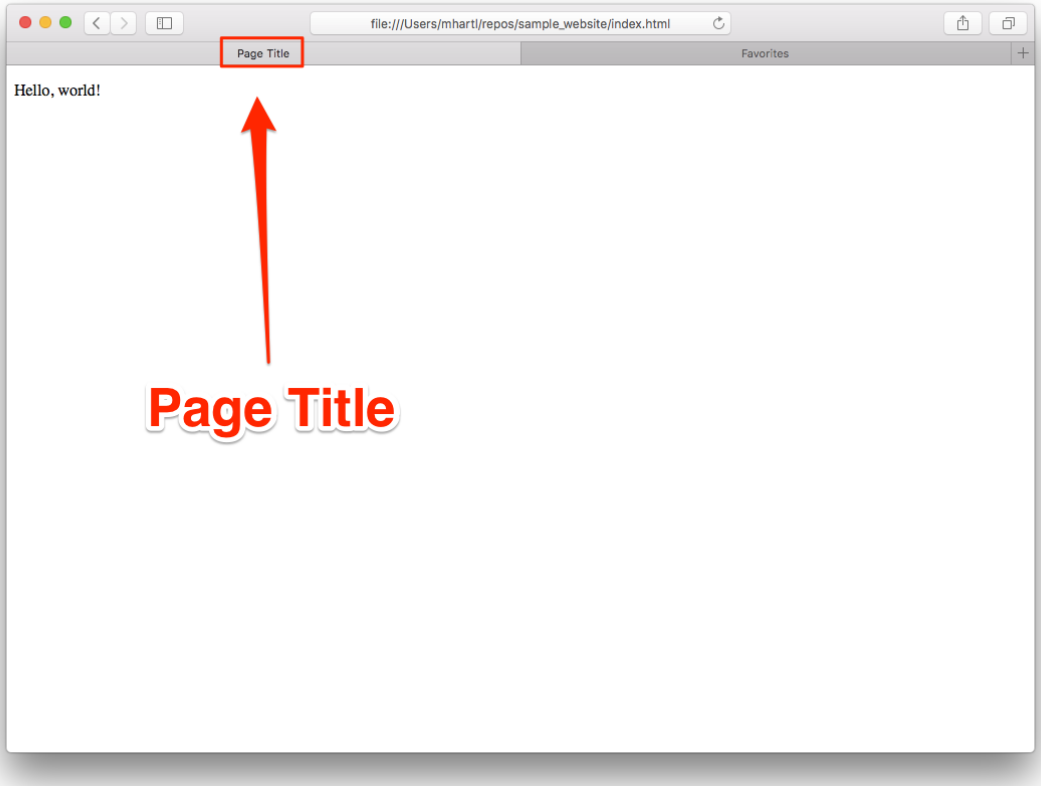


Figure 1.22: The page title in Safari.

With that, we're ready to commit our changes and push the results to GitHub Pages:

```
$ git commit -am "Convert index page to fully valid HTML"
$ git push
```

The result appears in Figure 1.23.

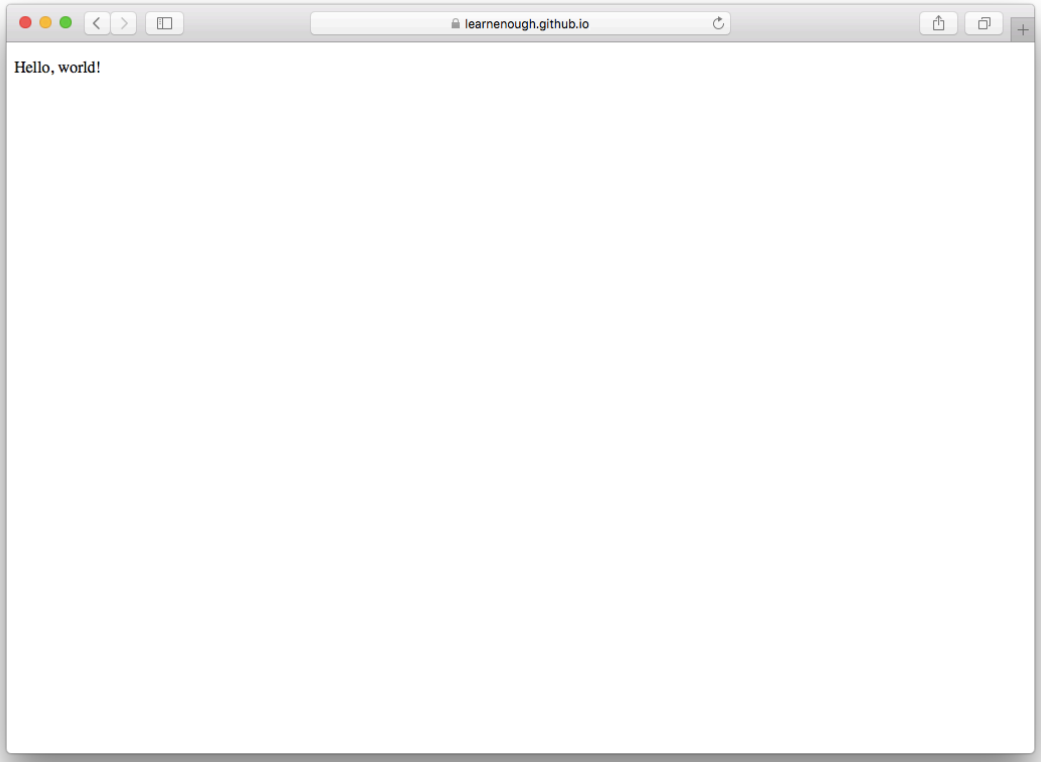


Figure 1.23: The valid "Hello, world!" page in production.

1.5.1 Exercises

1. Using the HTML validator, confirm that Listing 1.6 is valid HTML.
2. Remove `</title>` from Listing 1.9 and verify that it breaks the page (Figure 1.24). This underscores the importance of closing your tags. Confirm using the HTML validator that the resulting code fails validation.
3. By pasting in the contents of Listing 1.10 into `index.html`, confirm that the browser ignores the extra whitespace (including newlines) in the mailing address shown in Listing 1.10.

4. By adding the break tag **
** to the end of each of the first two lines of the address, obtain the nicely formatted address shown in Figure 1.25.

Listing 1.9: The index page with a missing closing tag.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Page Title
    <meta charset="utf-8">
  </head>
  <body>
  </body>
</html>
```

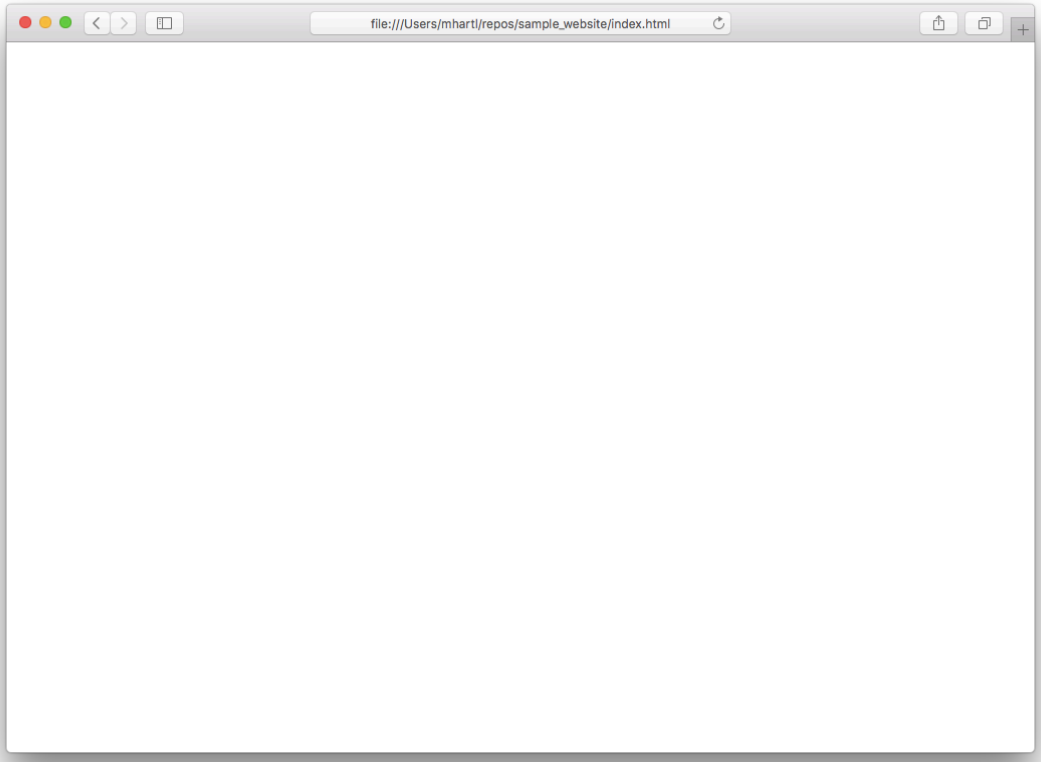


Figure 1.24: The look of failure… Close your tags!

Listing 1.10: An unformatted address.

```
<!DOCTYPE html>
<html>
  <head>
    <title>Who am I?</title>
    <meta charset="utf-8">
  </head>
  <body>
    Jean Valjean
    55 Rue Plumet
    Amonate, VA 24601
  </body>
</html>
```

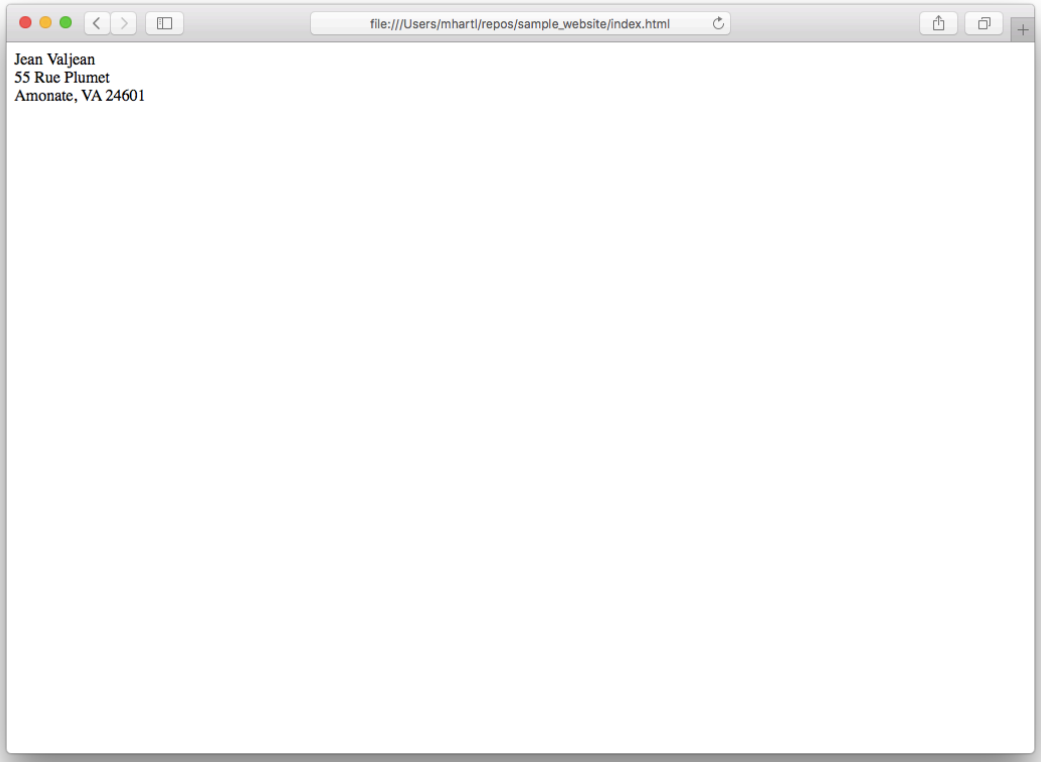


Figure 1.25: A nicely formatted address.

Chapter 2

Filling in the index page

Now that we've created and deployed a valid HTML page, it's time to start filling in the sample website. We'll begin by scoping out the structure of our index page while adding a more normal-sized paragraph (Section 2.1). We'll then format the resulting text (Section 2.2) while adding links (Section 2.3) and images (Section 2.4). Starting in Chapter 3, we'll add two more pages to go along with our index page, introducing several more important HTML tags along the way.

2.1 Headings

As noted in Section 1.3, our main index page will include some information about Learn Enough to Be Dangerous, so we'll start by updating the `title` tag contents with a new title, as shown in Listing 2.1.

Listing 2.1: Adding a new title.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="utf-8">
  </head>
  <body>
```

```
    <p>Hello, world!</p>
  </body>
</html>
```

Next, we'll replace the paragraph in the HTML **body** with several *headings*, constituting an outline of our document. The results appear in Listing 2.2.

Listing 2.2: Scoping out our index page with headings.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="utf-8">
  </head>
  <body>

    <h1>The Learn Enough Story</h1>

    <h2>Background</h2>

    <h2>Founders</h2>

    <h3>Michael Hartl</h3>

    <h3>Lee Donahoe</h3>

    <h3>Nick Merwin</h3>

  </body>
</html>
```

Listing 2.2 shows how to use the HTML header tags **h1**, **h2**, and **h3**, which represent three levels of headings. In this case, the top-level **h1** heading contains the main subject of the page:

```
<h1>The Learn Enough Story</h1>
```

The next two headings indicate additional subjects—in this case, some background on the company and some details about the founders:

```
<h2>Background</h2>

<h2>Founders</h2>
```

Because these subjects are subsidiary to the main story, they use the second-level heading **h2**. Finally, Listing 2.2 uses the third-level heading **h3** to list the three Learn Enough founders:

```
<h3>Michael Hartl</h3>

<h3>Lee Donahoe</h3>

<h3>Nick Merwin</h3>
```

As you might guess, most browsers render the top-level **h1** heading in a large font size, with **h2** and **h3** getting progressively smaller. (Figuring out how many heading sizes HTML supports is left as an exercise (Section 2.1.1).) The result appears in Figure 2.1.

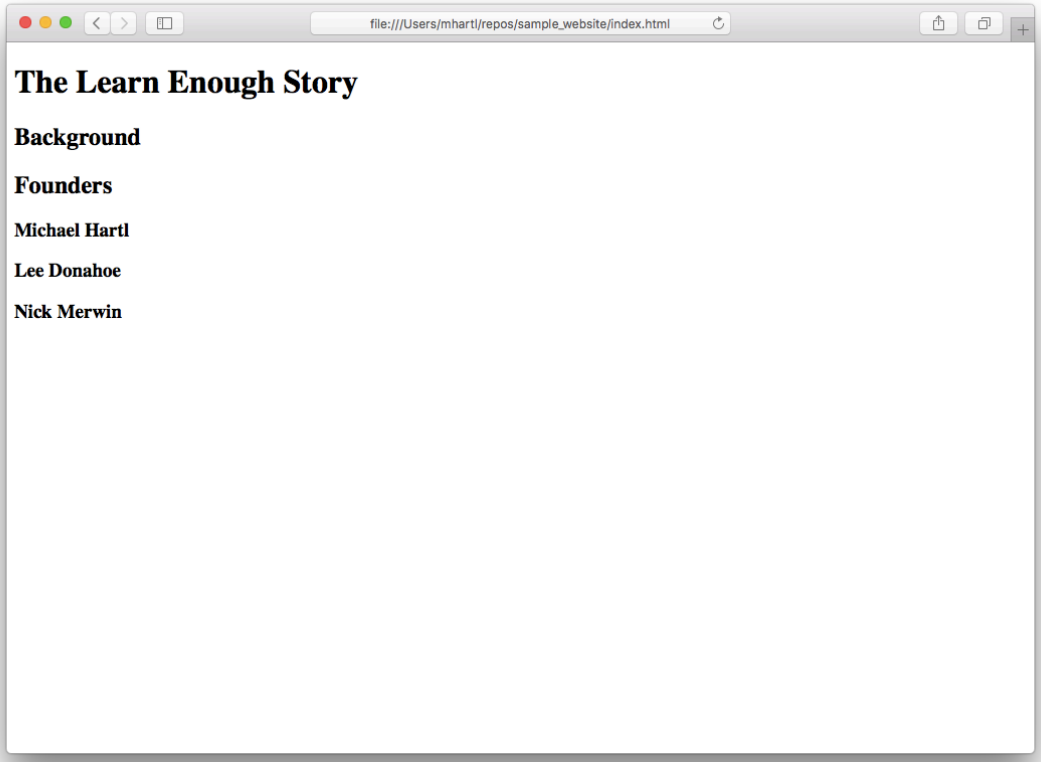


Figure 2.1: The initial headings for the index page.

2.1.1 Exercises

1. Listing 2.2 uses headings `h1` down to `h3`. By experimenting directly in `index.html`, determine how many levels of headings HTML supports.

2.2 Text formatting

Having added the headings to block out the structure of our document, let's now add an introductory paragraph describing the subject of the page. Unlike our previous one-line paragraph (Listing 1.4), this paragraph will include sev-

eral lines, as well as requiring some text formatting (this section) and a link (Section 2.3).

The paragraph itself appears in Listing 2.3, where the vertical dots indicate omitted content (just so we don't have to include all the tags in every code listing).

Listing 2.3: Adding a paragraph.
index.html

```
    .
    .
    .
    <h1>The Learn Enough Story</h1>

    <p>
      Learn Enough to Be Dangerous is a leader in the movement to teach
      technical sophistication, the seemingly magical ability to take
      command of your computer and get it to do your bidding. This includes
      everything from command lines and coding to guessing keyboard shortcuts,
      Googling error messages, and knowing when to just reboot the darn thing.
      We believe there are at least a billion people who can benefit from
      learning technical sophistication, probably more. To join our
      movement, sign up for our official email list now.
    </p>
    .
    .
    .
```

Note that the content is now indented inside the `p` tag:

```
<p>
  content
</p>
```

As discussed in Box 1.4, this makes the structure easier to see without affecting the appearance of the page.

It's also worth noting that Listing 2.3 includes newlines at the end of each line, but this is mainly so that the paragraph contents fit within in the constraints of a book. As seen in Figure 2.2, these newlines don't have any effect on the display, and in real life it's probably more common to have the content be all on one line, and simply enable word wrap (called "soft wrap" in Atom), as

described in *Learn Enough Text Editor to Be Dangerous*. This is what we recommend you do if you type the contents of Listing 2.3 in by hand.

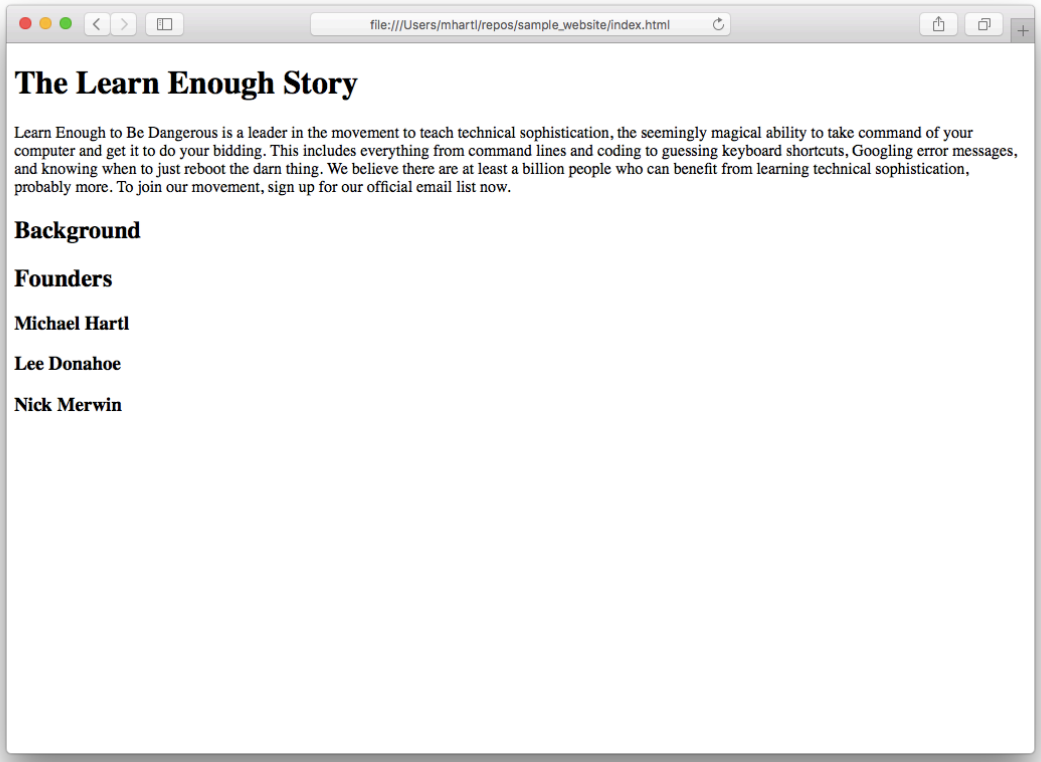


Figure 2.2: Adding a paragraph.

2.2.1 Emphasized text

The paragraph in Listing 2.3 has the right content, but it also introduces the new term *technical sophistication*, and it's a common typesetting convention to *emphasize* such terms using italics. As mentioned briefly in Box 1.2, we can accomplish this using the `em` tag, like this:

```
<em>technical sophistication</em>
```

Applying this idea to Listing 2.3 gives the result shown in Listing 2.4.

Listing 2.4: Emphasized text.
index.html

```
  .
  .
  .
  <h1>The Learn Enough Story</h1>

  <p>
    Learn Enough to Be Dangerous is a leader in the movement to teach
    <em>technical sophistication</em>, the seemingly magical ability to take
    command of your computer and get it to do your bidding. This includes
    everything from command lines and coding to guessing keyboard shortcuts,
    Googling error messages, and knowing when to just reboot the darn thing.
    We believe there are at least a billion people who can benefit from
    learning technical sophistication, probably more. To join our
    movement, sign up for our official email list now.
  </p>
  .
  .
  .
```

We can confirm that this worked by refreshing the browser, which shows that *technical sophistication* is properly emphasized (Figure 2.3).

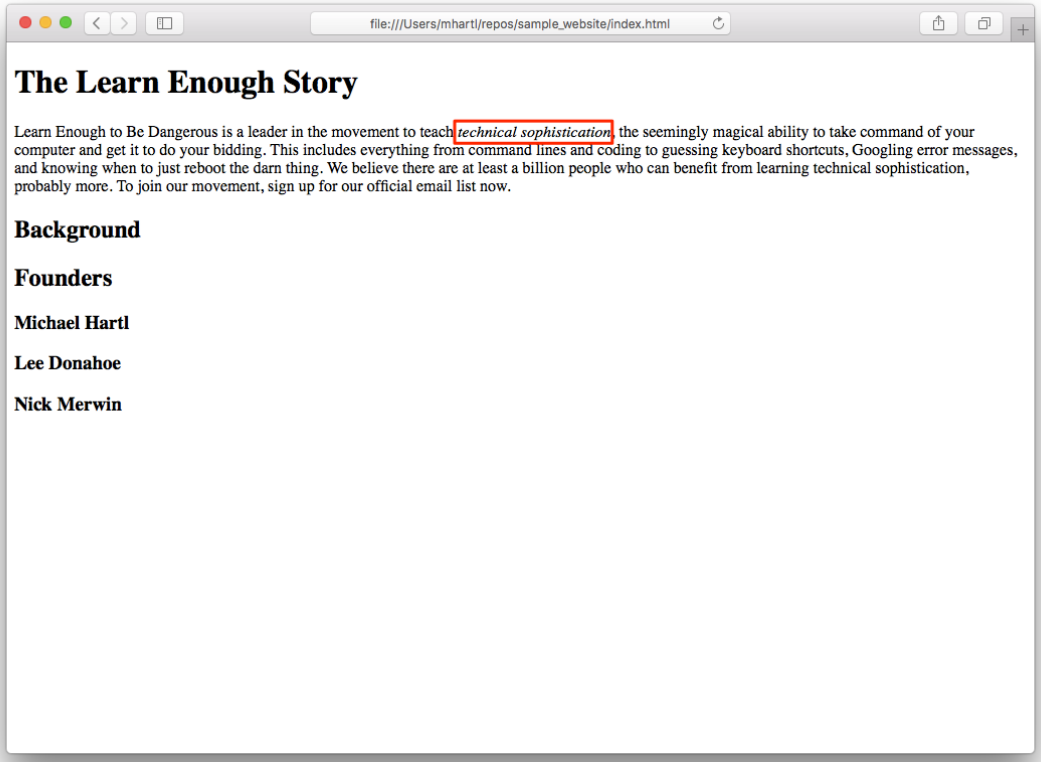


Figure 2.3: Emphasized text.

2.2.2 Strong text

As we saw briefly in Section 1.2, another possibility for drawing attention to particular text is to make it **strong** using the `strong` tag, which most browsers render as boldface text. In this case, we'd like to indicate the strength of our belief that **at least a billion people** can potentially benefit from learning technical sophistication, which we can do like this:

```
<strong>at least a billion people</strong>
```

Applying this idea to Listing 2.4 gives Listing 2.5.

Listing 2.5: Strong text.
index.html

```
.
.
.
<p>
  Learn Enough to Be Dangerous is a leader in the movement to teach
  <em>technical sophistication</em>, the seemingly magical ability to take
  command of your computer and get it to do your bidding. This includes
  everything from command lines and coding to guessing keyboard shortcuts,
  Googling error messages, and knowing when to just reboot the darn thing.
  We believe there are <strong>at least a billion people</strong> who
  can benefit from learning technical sophistication, probably more. To
  join our movement, sign up for our official email list now.
</p>
.
.
.
```

Refreshing the browser confirms that the text is now set in bold (Figure 2.4).

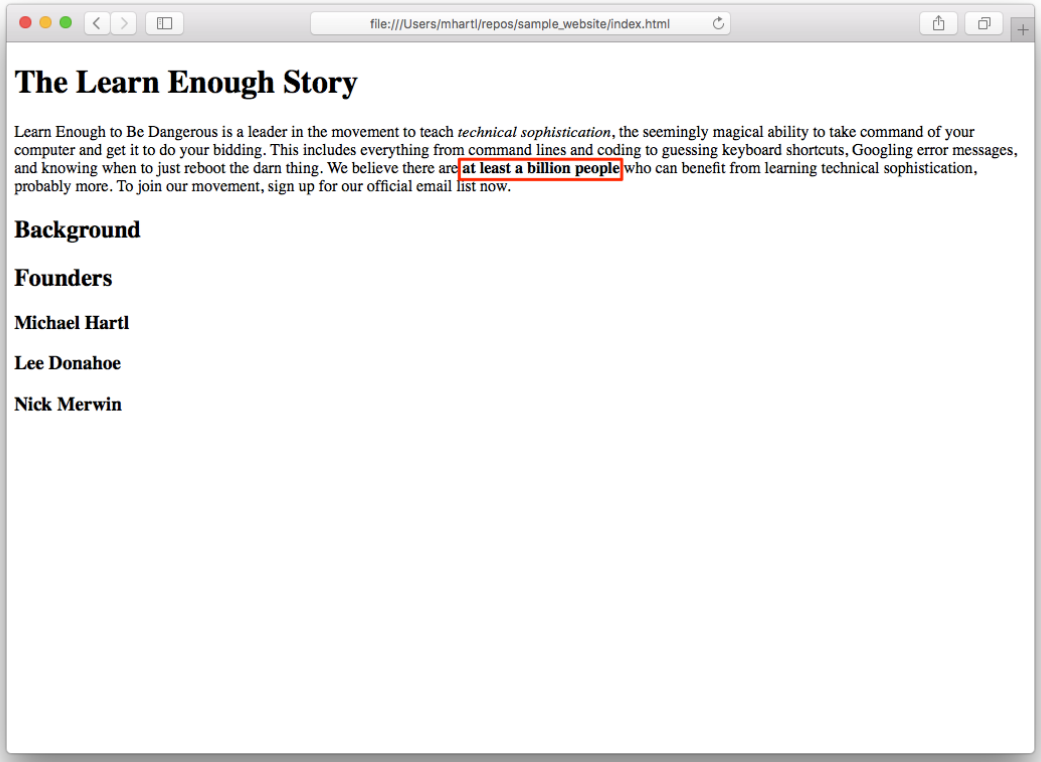


Figure 2.4: Strong text.

2.2.3 Exercises

1. Add the paragraph shown in Listing 2.6 under **founders**, then make the text **at least a billion people** bold (Figure 2.5).

2. What happens if you nest the `em` and `strong` tags? Is it possible to make something both italic ***and*** bold?

Listing 2.6: A paragraph with something to be made stronger.
index.html

```
.
.
.
<h3>Founders</h3>

<p>
  Learn Enough to Be Dangerous was founded in 2015 by Michael Hartl, Lee
  Donahoe, and Nick Merwin. We believe that the kind of technical
  sophistication taught by the Learn Enough tutorials can benefit
  at least a billion people, and probably more.
</p>
.
.
.
```

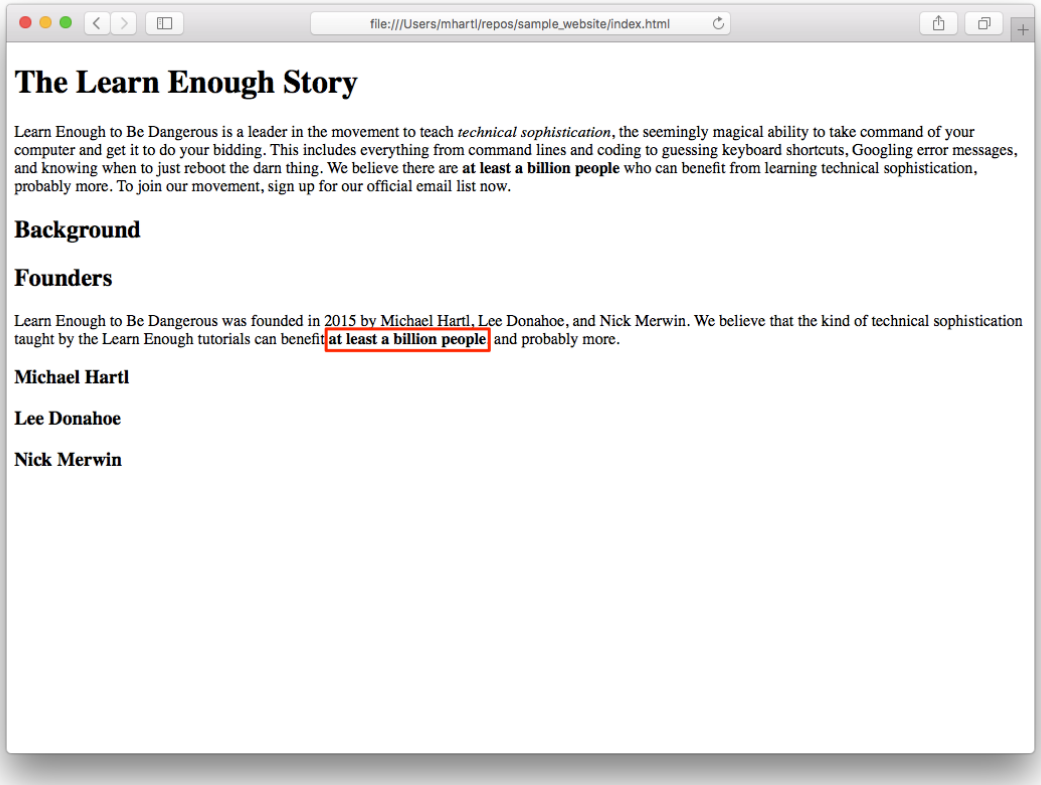


Figure 2.5: The appearance of a properly modified Listing 2.6.

2.3 Links

As mentioned in Section 1.2, much of the point of the Web is hypertext, with hyperlinks that let us move from one page to the next. The way to make such hyperlinks (or *links* for short) is with the HTML anchor tag `a`. (Why isn't it called `link`? We don't know, but at least `a` is short.)

In reading the paragraph from Listing 2.3, you may have noticed that the line

```
sign up for our official email list now
```

is practically begging for a link to a place where people can actually sign up for the email list. The exact text to use for the link is the subject of some debate, with some holding that links should be nouns, with others preferring to link a call to action if possible. We take a pragmatic approach, linking based on what seems most natural. In this case, we'll go with linking the text "sign up for our official email list", like this:

```
<a href="https://learnenough.com/email">sign up for our official email list</a>
```

This contains our first example of an *attribute*, which is a bit of text inside an HTML tag that supplies extra information about how to process it. In this case, the attribute is `href`, for "hypertext reference", and the value is the URL for the Learn Enough email list signup form.

Adding the email list link to the paragraph from Listing 2.5 gives Listing 2.7. Note how the text of the link breaks across two lines (something we saw before in Listing 1.1). Because HTML is insensitive to whitespace, this is effectively the same as having it all on one line.

Listing 2.7: Adding a link.

```
.
.
.
```

```
    <p>
      Learn Enough to Be Dangerous is a leader in the movement to teach <em>
      technical sophistication</em>, the seemingly magical ability to take
      command of your computer and get it to do your bidding. This includes
      everything from command lines and coding to guessing keyboard shortcuts,
      Googling error messages, and knowing when to just reboot the darn thing.
      We believe there are <strong>at least a billion people</strong> who
      can benefit from learning technical sophistication, probably more. To
      join our movement, <a href="https://learnenough.com/email">sign
      up for our official email list</a> now.
    </p>
    .
    .
    .
```

The result of Listing 2.7 appears in Figure 2.6.

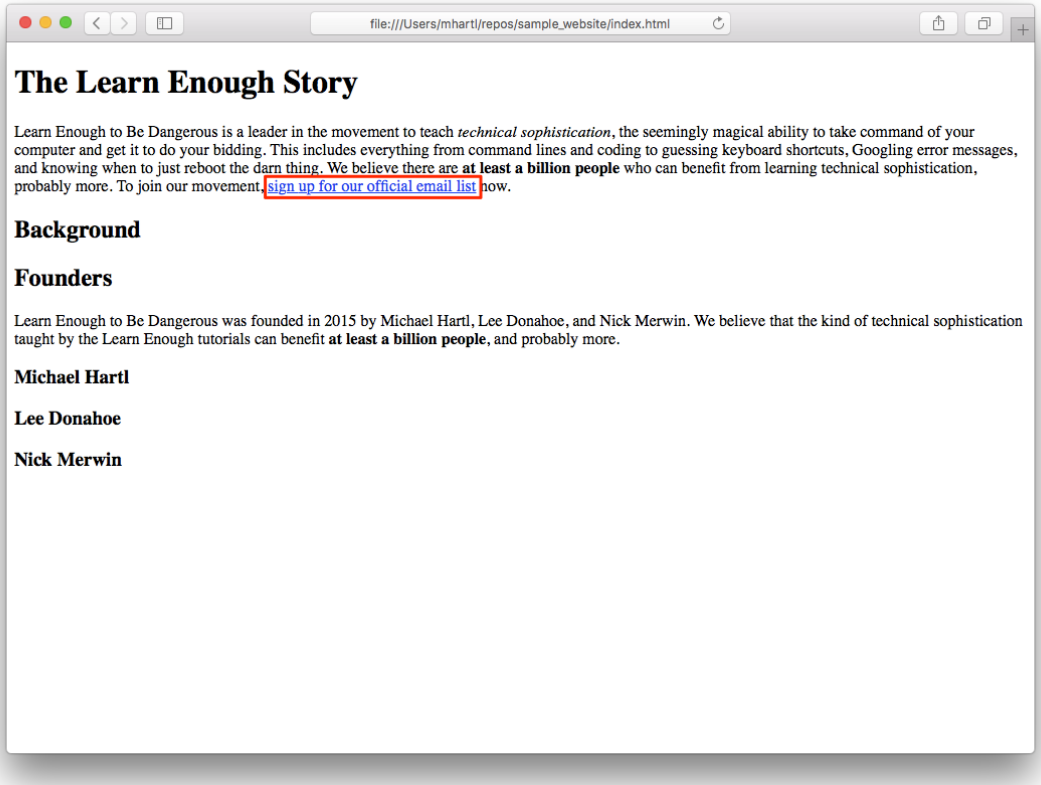


Figure 2.6: The result of adding a link.

Now that we know how to make links, we're ready to add a paragraph on the background of Learn Enough to Be Dangerous, which is both rich with links and includes further examples of the text-formatting tags shown in Section 2.2. The resulting paragraph, shown in Listing 2.8, should be placed under the second-level heading from Listing 2.2. Because it contains so many good examples of the tags we've covered so far, we recommend you type it in by hand.

Listing 2.8: Adding a paragraph using several useful tags.

```
.
.
.
<h2>Background</h2>

<p>
  Learn Enough to Be Dangerous is an outgrowth of the
  <a href="https://railstutorial.org/">Ruby on Rails Tutorial</a> and the
  <a href="https://www.softcover.io/">Softcover publishing platform</a>.
  This page is part of the sample site for
  <a href="https://www.learnenough.com/html"><em>Learn Enough HTML to
  Be Dangerous</em></a>, which teaches the basics of
  <strong>H</strong>yper<strong>T</strong>ext <strong>M</strong>arkup
  <strong>L</strong>anguage, the universal language of the World Wide Web.
  Other related tutorials can be found at
  <a href="https://www.learnenough.com/">learnenough.com</a>.
</p>
.
.
.
```

It's worth noting that Listing 2.8 contains an example of tag nesting, in the form of the link to the present tutorial:

```
<a href="..."><em>Learn Enough HTML to Be Dangerous</em></a>
```

As you might expect, this produces a link to emphasized text, as shown in Figure 2.7. You might also note that some of the links in Figure 2.7 are a different color, which is an indication that the links have been followed. (This is the default behavior for link colors, but it can be overridden by CSS, as discussed in *Learn Enough CSS & Layout to Be Dangerous* and mentioned briefly in Section 4.6.1.)

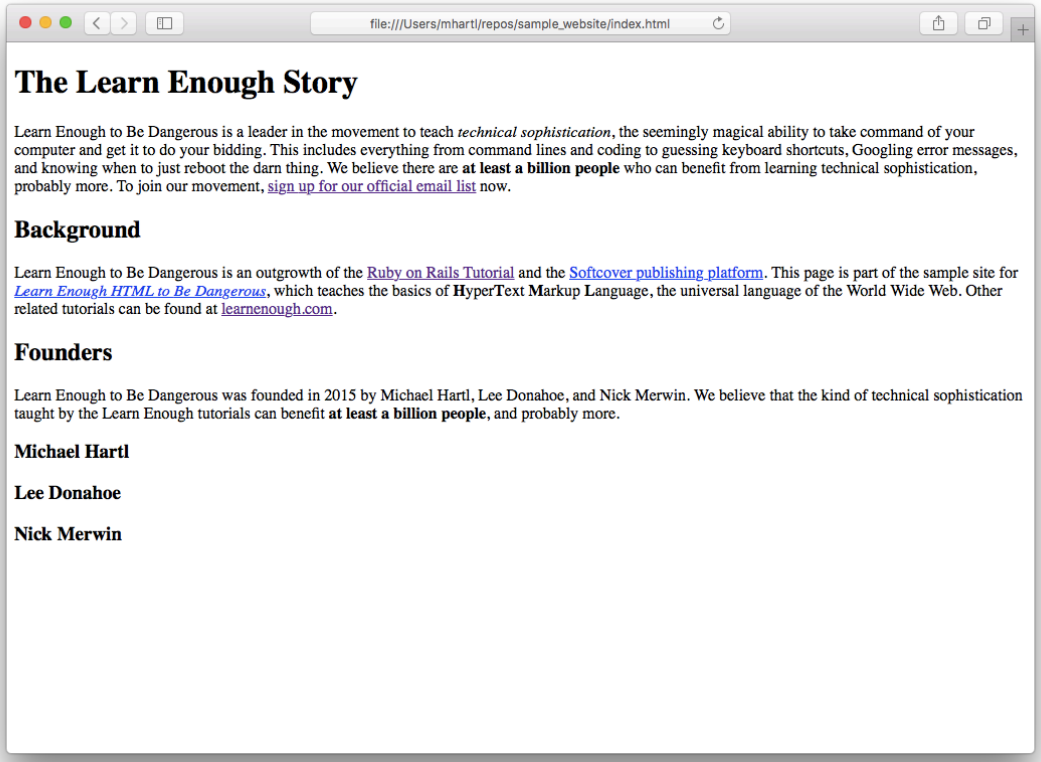


Figure 2.7: Adding a paragraph with formatting and links.

2.3.1 Exercises

1. Using the content shown in Listing 2.9, add short founder bios to the index page.
2. Now add Twitter follow links as shown in Listing 2.10. Does the content appear as shown in Figure 2.8?
3. It's sometimes convenient for external links (like those in the previous exercise) to open in a new browser tab. To do this, add the attribute `target="_blank"` to each of the Twitter links from Listing 2.10. (For technical reasons, it's important to add the rather obscure `rel="noopener"`

as well.) Does clicking on one of the links open in a new tab, as shown in Figure 2.8?

Listing 2.9: Adding short founder biographies.
index.html

```
.
.
.
<h2>Founders</h2>
.
.
.
<h3>Michael Hartl</h3>

<p>
  Michael is the creator of the <a href="https://www.railstutorial.org/">
  Ruby on Rails Tutorial</a> and principal author of the
  <a href="https://www.learnenough.com/">Learn Enough to Be Dangerous</a>
  introductory sequence. He is an advanced student of
  <a href="https://www.kravmaga.com/">Krav Maga</a> and has a three-step
  plan for world domination. Rumors that he's secretly a supervillain
  are slightly exaggerated.
</p>

<h3>Lee Donahoe</h3>

<p>
  When he's not literally swimming with sharks or hunting powder stashes
  on his snowboard, you can find Lee in front of his computer designing
  interfaces, doing front-end development, or writing some of the
  interface-related Learn Enough tutorials.
</p>

<h3>Nick Merwin</h3>

<p>
  You may have seen him shredding guitar live with Capital Cities on Jimmy
  Kimmel, Conan, or The Ellen Show, but rest assured Nick is a true nerd
  at heart. He's just as happy shredding well-spec'd lines of code from a
  tour bus as he is from his kitchen table.
</p>
.
.
.
```

Listing 2.10: Adding Twitter links to the bios.
index.html

```
.
.
.
<h3>Michael Hartl</h3>
.
.
.
<p>
  You should follow Michael on Twitter
  <a href="https://twitter.com/mhartl">here</a>.
</p>

<h3>Lee Donahoe</h3>
.
.
.
<p>
  You should follow Lee on Twitter
  <a href="https://twitter.com/leedonahoe">here</a>.
</p>

<h3>Nick Merwin</h3>
.
.
.
<p>
  You should follow Nick on Twitter
  <a href="https://twitter.com/nickmerwin">here</a>.
</p>
```

Figure 2.8: The index page after the completion of the exercises.

2.4 Adding images

At this point, our index page is coming into shape, but it's still missing a critical feature: what would a website be without images (of cats!)? Fortunately, adding images is similar to adding links, although there is an important difference in the structure of the tags. Recall from Section 2.3 that anchor tags take the form

```
<a href="https://example.com/">Example site</a>
```

We can include images in a similar way using the **`img`** tag, with a source attribute **`src`** and an alternate attribute **`alt`**:

```
<img src="images/kitten.jpg" alt="An adorable kitten">
```

Here **src** has a path to the image (either on the filesystem or on the Web), while **alt** lets developers add a bit of alternate text that describes the image in words. On some browsers, this text will display if the user's browser has a problem loading the image, but more importantly it will be read aloud (or even presented as braille) by screen readers used by the visually impaired, and is required by the HTML standard.[1]

The important difference mentioned above is that the **img** tag *doesn't* look like a typical tag, with content inside open and closing tags (Figure 1.5). If it did, it would look like this:

```
<img src="…" alt="…">content</img>
```

Instead, the **img** tag has no content and no closing tag:

```
<img src="…" alt="…">
```

That final **>** is all that's needed to close an **img** tag—like the **meta** tag discussed in Section 1.5, **img** is a void element (self-closing tag). A second variant uses **/>** at the end, like this:

```
<img src="…" alt="…" />
```

This syntax is designed to conform to XML, a markup language related to HTML, but using **/>** is not required in HTML5. We mention it mainly because you will still sometimes run across the XML-style syntax in other people's markup, and it's important to know that the two styles are exactly equivalent.

[1] The **alt** attribute is also useful to tell the search engine "spider" programs that crawl the web which things appear in the image.

Because the World Wide Web was apparently invented to share pictures of furry felines, we'll add an image of an adorable kitten to our sample index page, as shown in (Figure 2.9).[2]

Figure 2.9: The likely inspiration for the creation of the World Wide Web.

In order to link the kitten image, we could link directly to the source of Figure 2.9 from the live Web, like this:

```
<img src="https://example.com/images/kitten.jpg" alt="An adorable kitten">
```

This practice, called *hotlinking*, is generally considered bad form, for reasons we'll explain in Section 2.4.1. Instead, we'll copy the image to the local com-

[2]Image retrieved from https://www.flickr.com/photos/deborah_s_perspective/14144861329 on 2016-01-09. Copyright © 2009 by Deborah and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

puter, which will then be uploaded automatically when we deploy to GitHub Pages.

To do this, first create a directory called **images**:

```
$ mkdir images
```

Creating a separate **images** directory isn't strictly necessary, but it's useful for keeping our main project directory tidy. Next, download the image to the local disk using **curl**:[3]

```
$ curl -o images/kitten.jpg -OL https://cdn.learnenough.com/kitten.jpg
```

Once the image is available on our local disk, we can use its location as the value of the **src** attribute. Because the image is part of the same web project as **index.html**, we can use the *relative path* to the image, like this:

```
<img src="images/kitten.jpg" alt="An adorable kitten">
```

The **src** attribute **images/kitten.jpg** will automatically be interpreted as the correct full path to the file, which locally might be

```
file:///Users/mhartl/repos/sample_website/images/kitten.jpg
```

and on the server will be something like

```
https://learnenough.github.io/sample_website/images/kitten.jpg
```

While we're at it, we'll add a paragraph giving some context about the creation of the World Wide Web (including a correction to the wildly inaccurate claims of its feline origins). The result appears in Listing 2.11.

[3] The **curl** command is covered in *Learn Enough Command Line to Be Dangerous*.

Listing 2.11: An image, with a paragraph about the original web developer.
index.html

```
<h1>The Learn Enough Story</h1>
.
.
.
<p>
  HTML was created by the original "web developer", computer scientist
  <a href="https://en.wikipedia.org/wiki/Tim_Berners-Lee">Tim
  Berners-Lee</a>. It's not true that Sir Tim invented HTML in order to
  share pictures of his cat, but it would be cool if it were.
</p>

<img src="images/kitten.jpg" alt="An adorable kitten">

<h2>Background</h2>
.
.
.
```

After adding the contents of Listing 2.11 to **index.html**, the sample index page should look something like Figure 2.10.

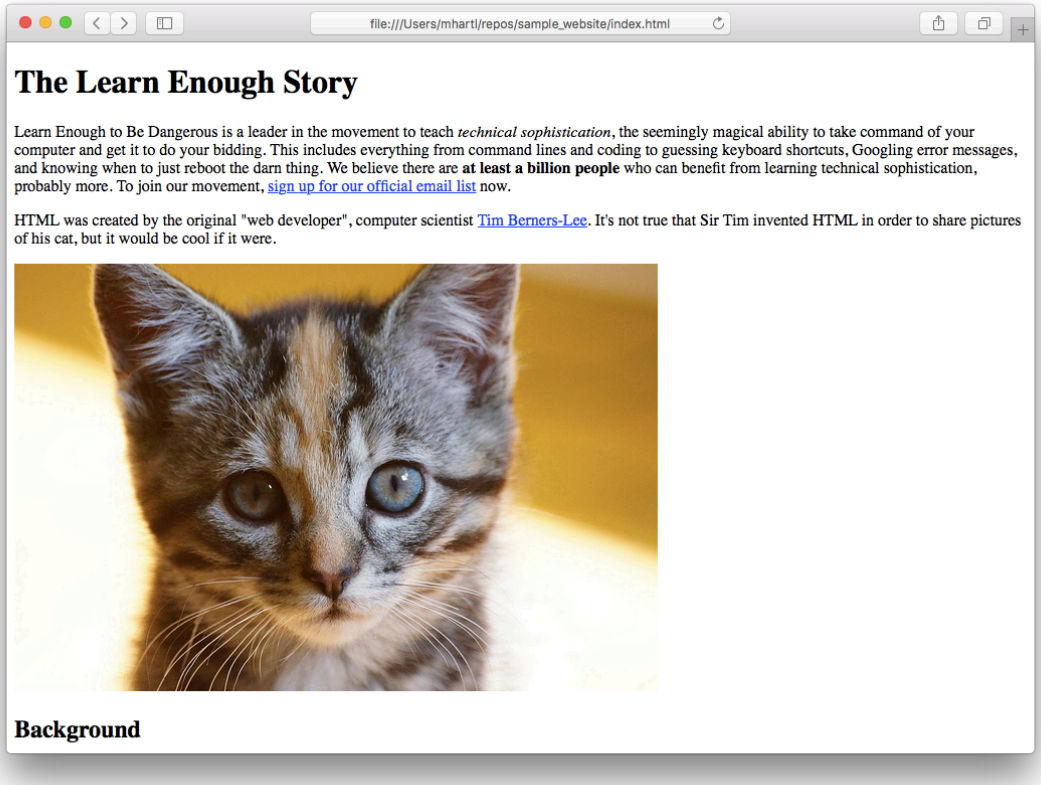


Figure 2.10: An awww-bligatory kitten image.

2.4.1 Hotlinking

We mentioned above that it's possible to link directly to images on the Web, a practice called *hotlinking*. The way to do this is to use a fully qualified URL as the `src` parameter, like this:

```
<img src="https://example.com/images/example.jpg" alt="A nonexistent example">
```

Hotlinking isn't usually a good practice, mainly because the image has to be at that exact spot on that exact site or it will not load, putting you at the mercy of the site's maintainer. The person who runs the site can also be charged for the

bandwidth used to serve the image, so hotlinking is considered inconsiderate as well. For these reasons, we generally recommend using local images for most applications.[4]

There are some important exceptions to the hotlinking rule, though, including an application known as *Gravatar*, which stands for "globally recognized avatar". Gravatar allows you to associate standard images with particular email addresses, and is used to display profile pictures on a large variety of websites, including GitHub and WordPress.[5] Gravatar images are specifically designed for hotlinking, so in this case the practice is actually encouraged. The image could still change, but in this case it's not a bug, it's a feature, because it gives the user control over their preferred profile image—if they update their picture, the change will automatically propagate to every site using the right Gravatar URL.

Gravatar URLs include a long string of hexadecimal digits (base 16, meaning 0–9 and a–f), like this:

```
https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6
```

Here `ffda7d145b83c4b118f982401f962ca6` is a unique string based on the email address associated with the Gravatar.[6] Gravatar URLs also support *query parameters*, which are additional pieces of information that come after the main URL, like `?s=150`:

```
https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6?s=150
```

Query parameters come after a question mark `?`, in this case `s=150`, which consists of a *key* `s` and a *value* `150`.[7] As you might be able to guess, `s` stands

[4] Here "local" means "local to the site" (which might be a remote server like GitHub Pages), not necessarily local to your development machine.

[5] Indeed, Gravatar was originally developed by GitHub cofounder Tom Preston-Werner, and was later acquired by Automattic, the parent company of WordPress.

[6] It's calculated using something called the MD5 message-digest algorithm, which is covered in the *Ruby on Rails Tutorial*.

[7] Multiple query parameters are separated by the ampersand symbol `&`, as in `https://example.com?foo=1&bar=2`. If you start looking at the URLs in the address bar of your browser, you'll see these query parameters everywhere.

for "size", and in this case the query parameter **`s=150`** sets the Gravatar size to 150 pixels. (By design, Gravatars are square, so a single parameter specifies the size uniquely.)

Using our newfound Gravatar knowledge, let's add avatar images to our index page under each of the Learn Enough to Be Dangerous founder bios (as added in Listing 2.9 from the exercises in Section 2.3.1). In a typical dynamic web application, such as that developed in the *Ruby on Rails Tutorial*, these URLs would be calculated on the fly based on the users' email addresses, but for convenience we'll supply you with the proper URLs. The result appears in Listing 2.12. (We've removed the leading indentation in Listing 2.12 so that the URLs fit, but in your **`index.html`** we suggest you keep the indentation as before.)

Listing 2.12: Adding Gravatar hotlinks for the Learn Enough founders.

```
.
.
.
<h2>Founders</h2>
.
.
.
<h3>Michael Hartl</h3>

<img src="https://gravatar.com/avatar/ffda7d145b83c4b118f982401f962ca6?s=150"
     alt="Michael Hartl">
.
.
.

<h3>Lee Donahoe</h3>

<img src="https://gravatar.com/avatar/b65522a6f3a6899705d119d7aa232a6d?s=150"
     alt="Lee Donahoe">
.
.
.

<h3>Nick Merwin</h3>

<img src="https://gravatar.com/avatar/e2d6ce2ba5c1b6d674ae8ff2b3b45d23?s=150"
     alt="Nick Merwin">
.
.
.
```

The result of adding the contents of Listing 2.12 should look something like Figure 2.11, although your page may not match exactly (Section 2.4.2).

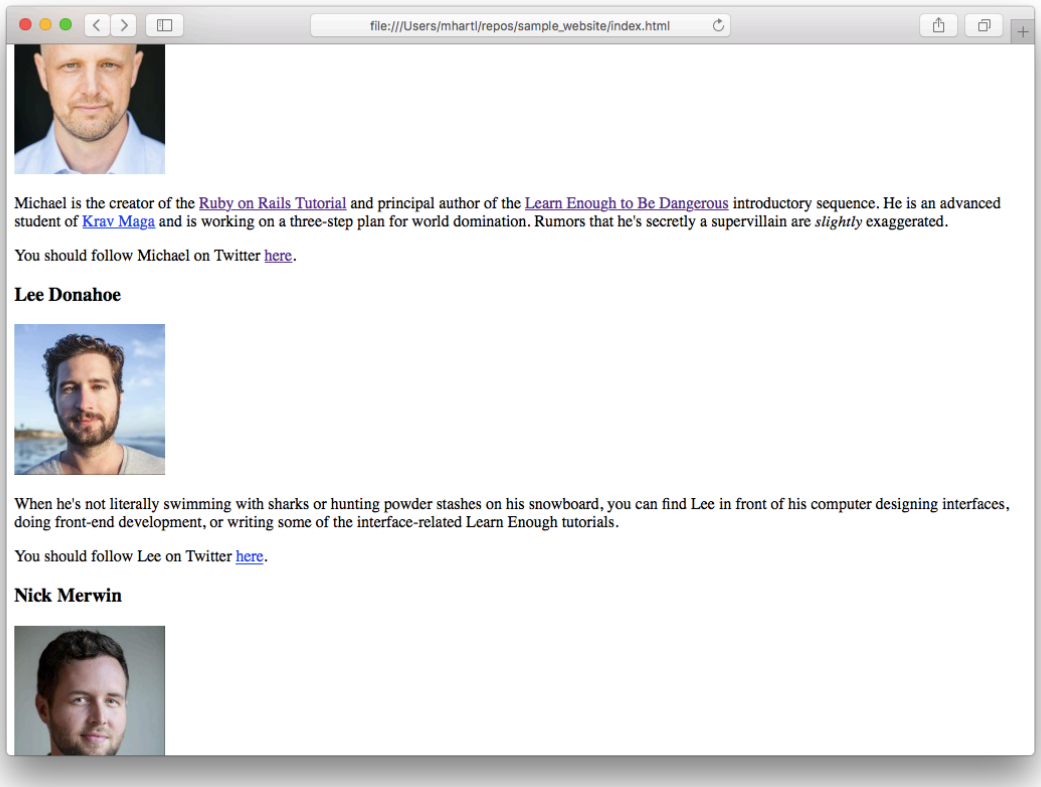


Figure 2.11: Adding Gravatar images.

With that, our sample website's index page has (nearly) taken its final form, so now is a good time to commit the changes and push to the live server at GitHub Pages:

```
$ git add -A
$ git commit -m "Add content and some images"
$ git push
```

The result should look something like Figure 2.12.

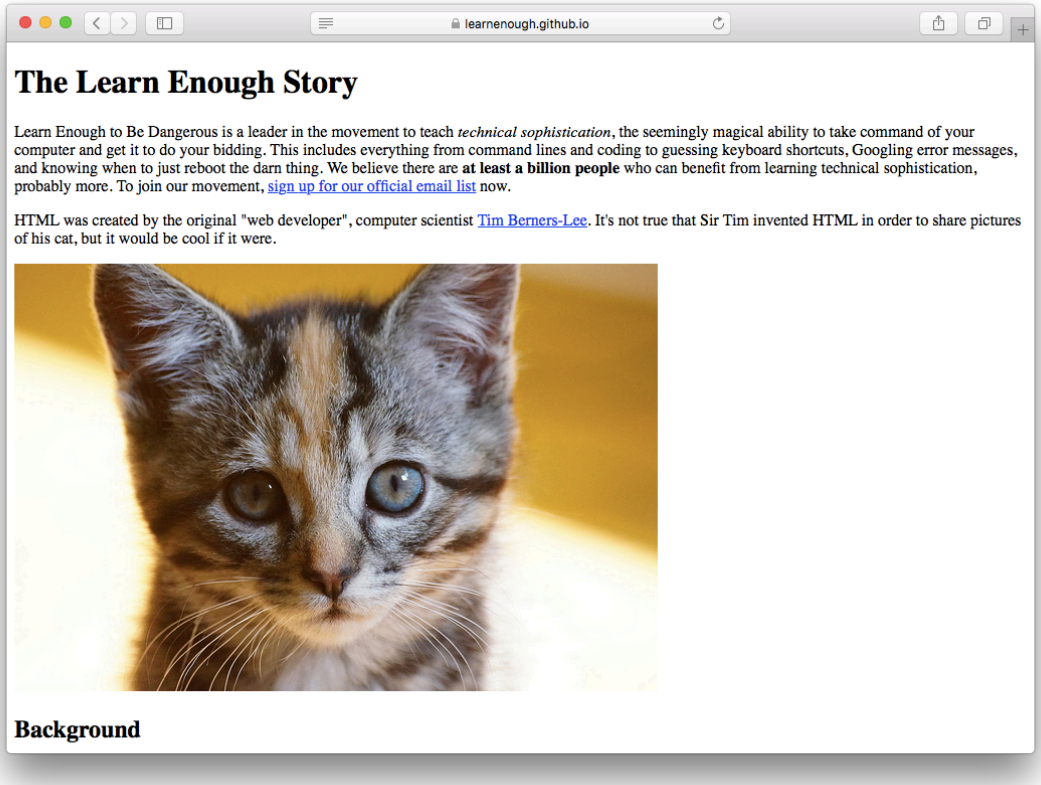


Figure 2.12: The sample index page on the live Web.

2.4.2 Exercises

1. Why might the images shown in Figure 2.11 not match your results exactly? *Hint*: It's not a bug, it's a feature.

2. The kitten image in Figure 2.9 is available under a Creative Commons license that requires attribution. We'll fulfill this requirement by linking the image on our page to the original image's URL, which involves nesting the `a` and `img` tags, as shown in Listing 2.13. (Be sure to replace `FILL_IN` with the right URL.) How does this change the page's (a) appearance and (b) behavior?

3. Under the first paragraph on **index.html**, let's add a link to the Learn Enough Twitter account. First, download the Twitter logo as shown in Listing 2.14. Then, add a link to both the page and the logo image, as shown in Listing 2.15. Be sure to replace **FILL_IN** with the right path to the image. Note that Listing 2.15 introduces *inline styling*, which is the subject of Chapter 4. *Extra credit*: Follow Learn Enough on Twitter here.

Listing 2.13: Linking an image.
index.html

```
    <p>
      HTML was created by the original "web developer", computer scientist
      <a href="https://en.wikipedia.org/wiki/Tim_Berners-Lee">Tim
      Berners-Lee</a>. It's not true that Sir Tim invented HTML in order to
      share pictures of his cat, but it would be cool if it were.
    </p>

    <a href="FILL_IN">
      <img src="images/kitten.jpg" alt="An adorable kitten">
    </a>
```

Listing 2.14: Downloading the Twitter logo.

```
$ curl -o images/small_twitter_logo.png \
>      -OL https://cdn.learnenough.com/small_twitter_logo.png
```

Note that the backslash **** should be typed, but your shell will include **>** automatically, so don't just copy and paste the whole thing.

Listing 2.15: Adding links to the Learn Enough Twitter account.
index.html

```
      .
      .
      .
      for our official email list</a> now.
    </p>
```

```
    <p>
      <a href="https://twitter.com/learnenough" target="_blank"
      rel="noopener" style="text-decoration: none;">
        <img src="FILL_IN">
      </a>
      You should follow Learn Enough on Twitter
      <a href="https://twitter.com/learnenough"
         target="_blank" rel="noopener">here</a>.
    </p>
    .
    .
    .
```

Chapter 3

More pages, more tags

Having completed a full index page in Chapter 2 (and having learned a bunch of HTML tags along the way), the time has now come to add a couple more pages to our website. This will give us a chance to learn some more useful HTML tags, while discovering some of the limitations of our purely hand-edited approach.

The first new addition will be a meta-page on HTML tags themselves, which will give us a chance to reinforce the preceding material as we build it. The second page includes a lighthearted report on *Moby-Dick*, one of the free books available on the Softcover website. This second page especially lends itself to styling, a task we'll take up in Chapter 4.

3.1 An HTML page about HTML

We'll start by adding a reference page to collect some of the HTML tags we've learned about so far. This means creating a new file, which we can do by creating a new tab (typically with ⌘N) and then saving it as **`tags.html`**. Another method, and a personal favorite, is to run **`touch tags.html`** at the command line and then use ⌘P to open it in the editor.

Once you've created **`tags.html`**, by whatever method, fill it with the contents of Listing 3.1.

Listing 3.1: Adding the beginning of a page on HTML tags.
tags.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Tags</title>
    <meta charset="utf-8">
  </head>
  <body>

    <h1>Important HTML tags</h1>

    <img src="images/storm_trooper_tagged.jpg" alt="Tagged Storm Trooper">

    <p>
      This page is designed as a quick reference for some of the common tags
      covered in <a href="https://learnenough.com/html"><em>Learn
      Enough HTML to Be Dangerous</em></a>. In the process of making it, we'll
      learn how to make HTML <em>tables</em> via <code>table</code> and
      related tags.
    </p>

    <p>
      The tables below don't include all HTML tags, but they do list many of
      the most important ones.
    </p>

  </body>
</html>
```

Note that Listing 3.1 involves repeating the HTML skeleton from Listing 1.7, which is an inconvenient duplication of effort. It also becomes increasingly annoying as the number of pages in a site grows, especially if (as is often the case) we need to make changes to the **head** of the document. We'll deal with this issue the Right Way™ in *Learn Enough CSS & Layout to Be Dangerous* (using a so-called *templating system*), but for now we'll just live with the annoyance.

Listing 3.1 introduces one new tag, the minor but occasionally useful **code** tag:

```
<code>table</code>
```

Designed to display pieces of markup or source code, the **code** tag is rendered

by most browsers in a monospace font, `like this`. (In a monospace font, all letters have the same width, which is especially convenient when lining up formatted code.)

The page defined by Listing 3.1 won't yet render as intended because of the `img` tag, which currently references a nonexistent image. To fix this, download the image (which is just a smaller version of the one in Figure 1.4) to the local disk:

```
$ curl -o images/storm_trooper_tagged.jpg \
>      -OL https://cdn.learnenough.com/storm_trooper_tagged.jpg
```

Note that the backslash `\` should be typed, but your shell will include `>` automatically, so don't just copy and paste the whole thing.

The result appears in Figure 3.1.

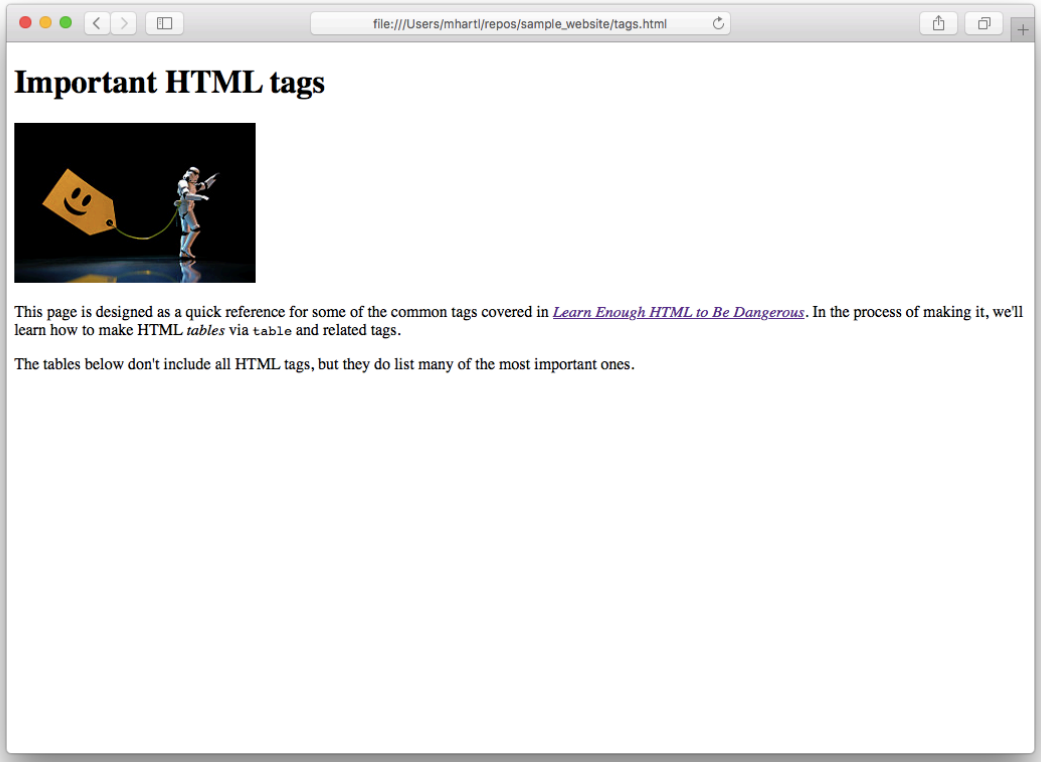


Figure 3.1: The beginning of an HTML tags page.

3.1.1 Exercises

1. Does the HTML in Listing 3.1 validate?

2. As with the kitten image, the Stormtrooper tag image is used under a Creative Commons license that requires attribution. By following the model in Listing 2.13, fulfill this requirement by linking the image in Listing 3.1 to the original image URL.

3.2 Tables

Now that we've got a basic page set up, we'll make a list of some of the tags we've learned about so far. Our plan is to indicate the exact HTML code for the tag, its name, and its purpose. The result is a *table* of information, so to display it we'll use the HTML `table` tag. Because HTML tags divide broadly into *inline elements* and *block elements* (Box 3.1), we'll make a separate table for each type of tag.

Box 3.1. Inline vs. block

All of the elements on a page of HTML either flow with the text around them or interrupt the flow by creating a box of content that is separate from the other content on the page. The first category of tags is known as *inline*; the second category is called *block* (Figure 3.2).

All of the elements that modify text, such as `<strong>` and `<em>`, are inline elements, which makes sense since we wouldn't want text to jump to a new line every time we made it bold or italic. Other common inline elements include links and (perhaps surprisingly) images. Inline elements take up only as much width on the page as is necessary to contain the content inside the tags—you can think of inline elements as being shrink-wrapped around the content inside them

In contrast, block elements always start on a new line, as if there is a line break in front of them, so one of their main purposes is to divide the page's text into different presentational groups, such as paragraphs or lists. Unlike inline elements, block elements bound the full width of the page, like an inflexible cardboard box.

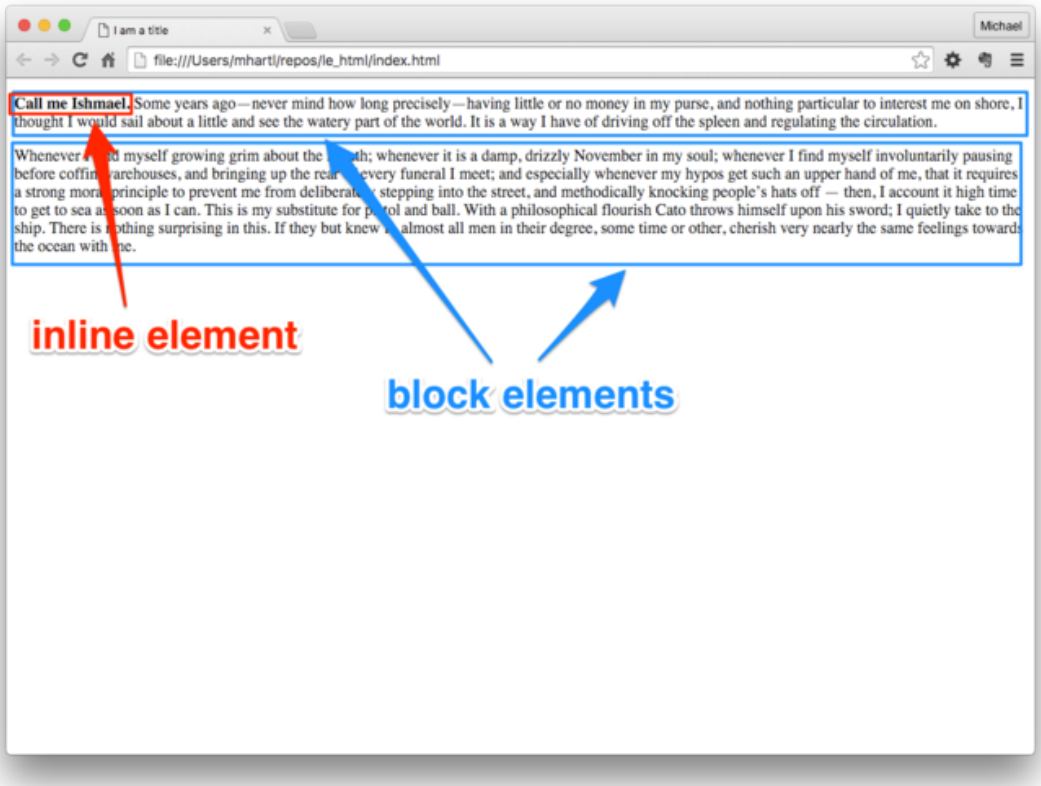


Figure 3.2: Inline vs. block elements.

3.2.1 Block elements

A table is defined by opening and closing `table` tags, with each *table row* defined by the `tr` tag. Typically, the first row includes labels for the table's columns via *table headers*, defined by the `th` tag, as shown in Listing 3.2.

Listing 3.2: Defining a table with headers.
tags.html

```
    .
    .
    .
    <p>
```

```
      The tables below don't include all HTML tags, but they do list many of
      the most important ones.
    </p>

    <h2>Block Elements</h2>

    <table>
      <tr>
        <th>Tag</th>
        <th>Name</th>
        <th>Purpose</th>
      </tr>
    </table>
    .
    .
    .
```

The result of Listing 3.2 is shown in Figure 3.3. Because the natural layout inside a file is vertical, whereas the display is horizontal, it can be challenging to mentally map table contents to the visual result, but it gets easier with practice.

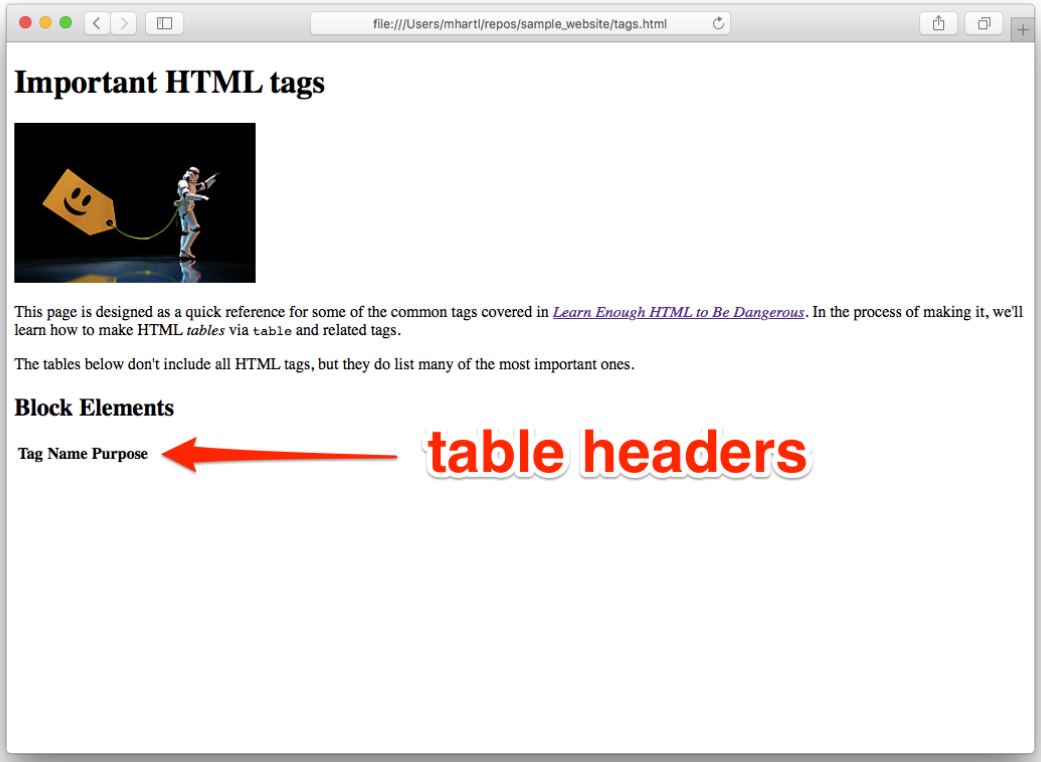


Figure 3.3: Table headers.

After optionally defining the table headers, tables generally consist of a series of *table data* cells defined by the `td` tag. We can get started with the content of the table by adding a row for the heading tags first introduced in Section 2.1. The result (which incidentally solves the exercise from Section 2.1.1) appears in Listing 3.3.

Listing 3.3: Adding a row of data.
tags.html

```
    .
    .
    .
    <p>
```

```
      The tables below don't include all HTML tags, but they do list many of
      the most important ones.
    </p>

    <h2>Block Elements</h2>

    <table>
      <tr>
        <th>Tag</th>
        <th>Name</th>
        <th>Purpose</th>
      </tr>
      <tr>
        <td><code>h1</code>&ndash;<code>h6</code></td>
        <td>headings</td>
        <td>include a heading (levels 1&ndash;6)</td>
      </tr>
    </table>
    .
    .
    .
```

Listing 3.3 uses the **code** tag we saw in Listing 3.1, and also introduces uses **–**, which is "character entity reference" for an *en dash* (a dash roughly the width of the letter "n", like this: –). Because our HTML document uses the **utf-8** character set (Listing 1.7), we could also use a literal en dash –, but the entity reference is common, and it's good to know how to use both.

This is a lot to handle at once, which makes it good practice for learning to visualize the result of HTML markup. Once you have a guess in your mind's eye, you can refresh your browser to see the result (Figure 3.4).

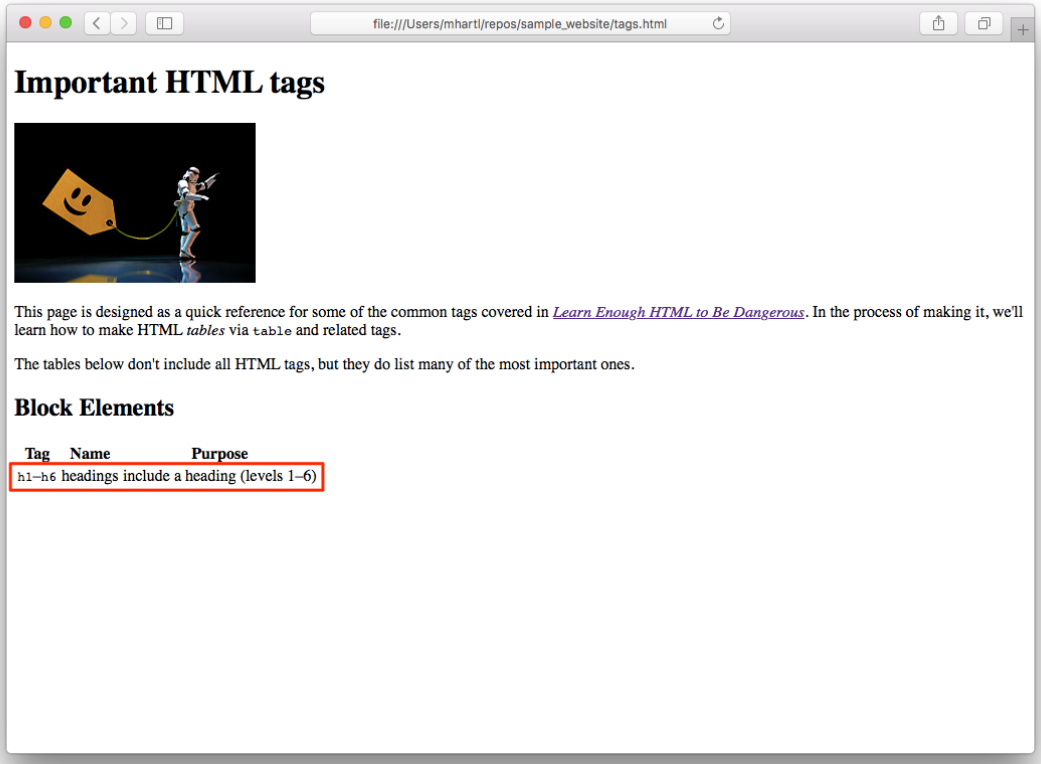


Figure 3.4: Adding a row of table data.

We've now seen the most important table tags, so we're ready to fill in the table with rows for the other block-level tags we've seen so far. These include `p` and the table tags themselves.[1] The result appears in Listing 3.4.[2]

Listing 3.4: A more complete table of HTML block elements.
tags.html

```
    .
    .
```

[1] Technically, the `td` tag is more like an "inline block", but this distinction isn't important for our purposes.

[2] Some code listings include yellow highlights for important lines, but we'll avoid the "wall of yellow" by omitting the highlights when there are as many new lines as there are in listings like Listing 3.4.

```
.
<p>
  The tables below don't include all HTML tags, but they do list many of
  the most important ones.
</p>

<h2>Block Elements</h2>

<table>
  <tr>
    <th>Tag</th>
    <th>Name</th>
    <th>Purpose</th>
  </tr>
  <tr>
    <td><code>h1</code>&ndash;<code>h6</code></td>
    <td>headings</td>
    <td>include a heading (levels 1&ndash;6)</td>
  </tr>
  <tr>
    <td><code>p</code></td>
    <td>paragraph</td>
    <td>include a paragraph of text</td>
  </tr>
  <tr>
    <td><code>table</code></td>
    <td>table</td>
    <td>include a table</td>
  </tr>
  <tr>
    <td><code>tr</code></td>
    <td>table row</td>
    <td>include a row of data</td>
  </tr>
  <tr>
    <td><code>th</code></td>
    <td>table header</td>
    <td>make a table header</td>
  </tr>
  <tr>
    <td><code>td</code></td>
    <td>table data</td>
    <td>include a table data cell</td>
  </tr>
</table>
.
.
.
```

There are a lot of new rows in Listing 3.4, so you can copy and paste if you want, but you'll learn more by typing in the tags by hand. (You'll find that

it can be quite cumbersome, which is one reason many real-world tables are generated from databases using programming languages like Ruby.) The result appears in Figure 3.5.

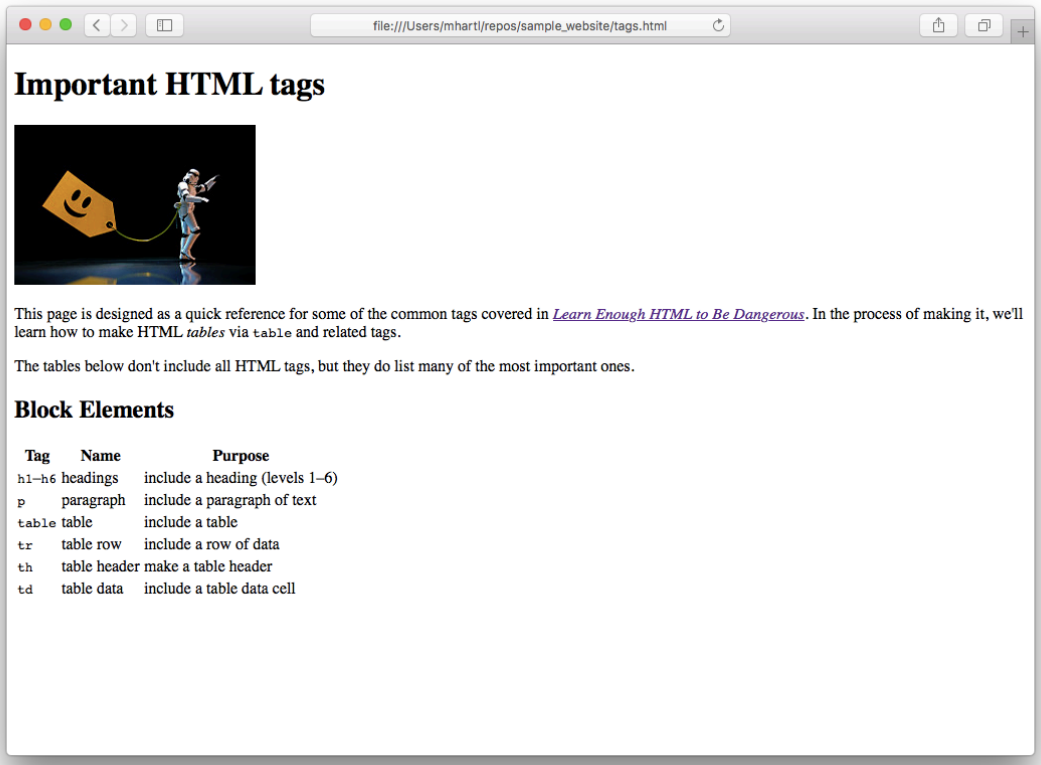


Figure 3.5: A table for some HTML block elements.

By the way, you might notice in Figure 3.5 that the spacing around the table data cells isn't ideal. This is exactly the kind of detail that is handled by Cascading Style Sheets (as covered in *Learn Enough CSS & Layout to Be Dangerous*).

3.2.2 Inline elements

Now that we know how to make a basic table, we're ready to add a new table of inline elements as well. Because inline elements by definition don't start a

new line, it's easy to include examples of the tags along with their definitions. For example, we can include a working example of the **em** tag, as shown in Listing 3.5.

Listing 3.5: A start at a table of inline tags.
tags.html

```
.
.
.
   <h2>Inline Elements</h2>

    <table>
      <tr>
        <th>Tag</th>
        <th>Name</th>
        <th>Purpose</th>
        <th>Example</th>
        <th>Result</th>
      </tr>
      <tr>
        <td><code>em</code></td>
        <td>emphasized</td>
        <td>make emphasized text</td>
        <td><code>&lt;em&gt;technical sophistication&lt;/em&gt;</code></td>
        <td><em>technical sophistication</em></td>
      </tr>
    </table>
.
.
.
```

Listing 3.5 introduces the solution to the tricky problem of displaying literal angle brackets, which arranges for the browser to display, e.g., technical sophistication rather than *technical sophistication*. The way to do it is by "escaping out" **<** and **>** with the HTML character entities **<** ("less than") and **>** ("greater than"). The result appears in Figure 3.6.

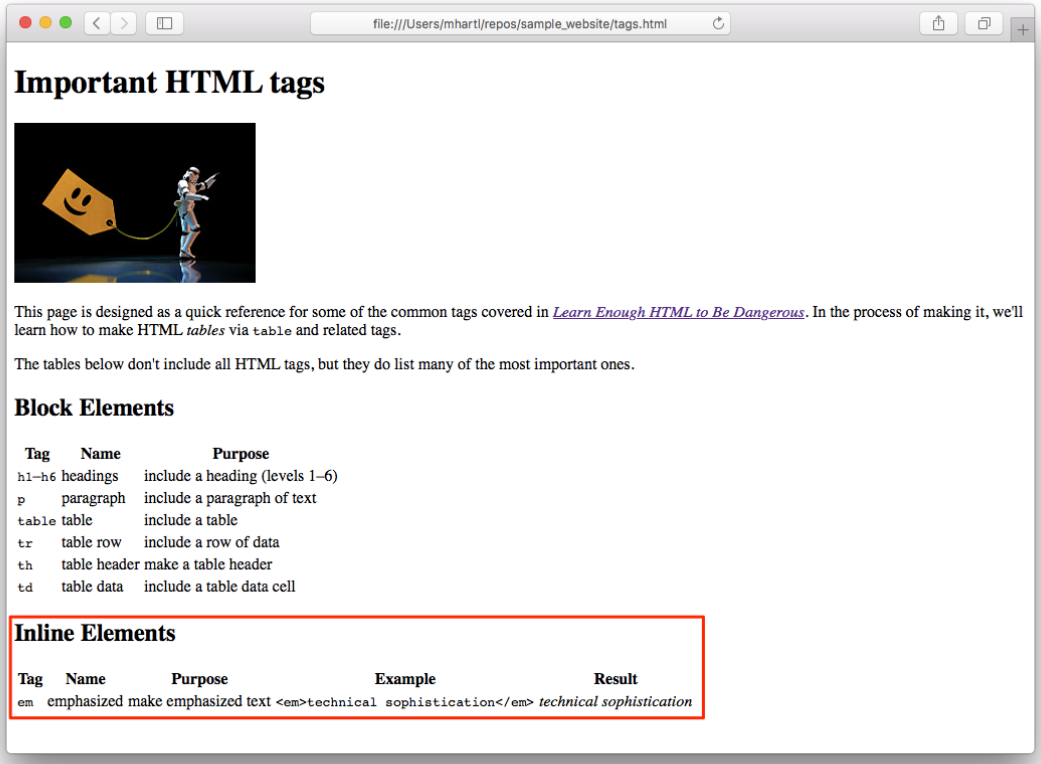


Figure 3.6: A good start at a table of inline elements.

Some other inline elements we've encountered so far are **`strong`**, **`a`**, **`img`**, and **`code`**. Adding them to the table in Listing 3.5 is left as an exercise. Either before or after doing that exercise, I suggest adding and committing the changes and then pushing up to GitHub Pages:

```
$ git add -A
$ git commit -m "Add a tags page"
$ git push
```

3.2.3 Exercises

1. Follow the template in Listing 3.6 to add information on the **strong**, **a**, and **img** tags. Why does the **img** tag example use the Bitly link shortener?

2. Add an additional row for the **code** tag. Does the page validate?

Listing 3.6: Template for adding **strong**, **a**, and **img** tags.
tags.html

```
    .
    .
    .
    <h2>Inline Elements</h2>

    <table>
      <tr>
        <th>Tag</th>
        <th>Name</th>
        <th>Purpose</th>
        <th>Example</th>
        <th>Result</th>
      </tr>
      <tr>
        <td><code>em</code></td>
        <td>emphasized</td>
        <td>make emphasized text</td>
        <td><code>&lt;em&gt;technical sophistication&lt;/em&gt;</code></td>
        <td><em>technical sophistication</em></td>
      </tr>
      <tr>
        <td><code>strong</code></td>
        <td>strong</td>
        <td>make strong text</td>
        <td>
          <code>&lt;strong&gt;at least a billion people&lt;/strong&gt;</code>
        </td>
        <td>FILL_IN</td>
      </tr>
      <tr>
        <td><code>a</code></td>
        <td>anchor</td>
        <td>make hyperlink</td>
        <td>
          <code>
            &lt;a href="https://learnenough.com/"&gt;Learn Enough&lt;/a&gt;
          </code>
        </td>
```

```
        <td>FILL_IN</td>
      </tr>
      <tr>
        <td><code>img</code></td>
        <td>image</td>
        <td>include an image</td>
        <td>
          <code>
            &lt;img src="https://bit.ly/1MZAFuQ" alt="Michael Hartl"&gt;
          </code>
        </td>
        <td>FILL_IN</td>
      </tr>
    </table>
    .
    .
    .
```

3.3 Divs and spans

Having created a new page for our HTML table experiment, we're now ready to make a third page for our sample website. We'll add some initial content while blocking out the structure with three new tags: **`header`**, **`div`** (for *division*), and **`span`**. These tags have little to no impact on the page appearance, but they help us organize the page and its contents into logical units. All three tags, and especially **`div`**s and **`span`**s, are heavily used when styling web pages using CSS (*Learn Enough CSS & Layout to Be Dangerous*). We'll get a preview of this practice when applying inline styles in Chapter 4.

The page itself will take the form of a mock book report on the classic American whaling novel *Moby-Dick*, which among other places is available in a free Softcover edition. As we'll see in Chapter 4, this content will especially lend itself to styling.

The first thing you should do is create a file called **`moby_dick.html`**. Then, because it's never wrong to include pictures of animals (even non-cats!) on web pages, we'll include a picture of sperm whales (Figure 3.7)[3] to go along

[3] Image retrieved from https://commons.wikimedia.org/wiki/File:Sperm_whale_pod.jpg on 2016-01-17. Copyright © 2013 by Gabriel Barathieu and used unaltered under the terms of the Creative Commons Attribution-ShareAlike 2.0 Generic license.

with our report. We'll also include an image for the book's cover (Figure 3.8).

Figure 3.7: A pod of sperm whales.

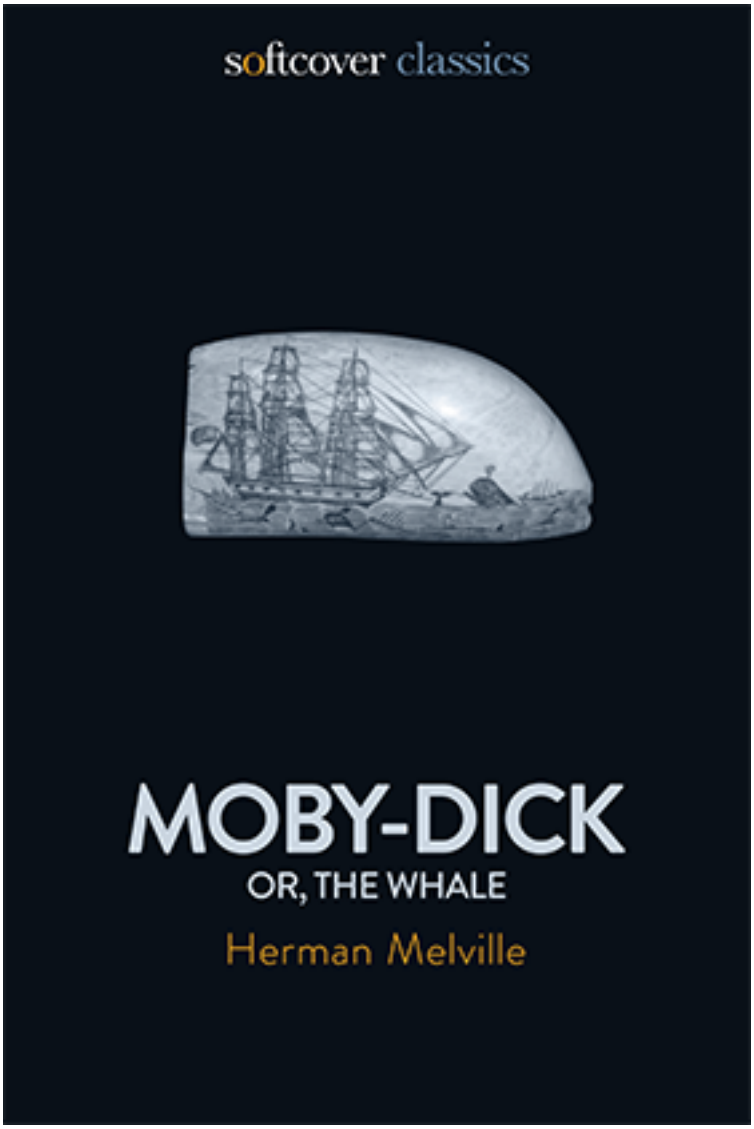


Figure 3.8: The cover image for *Moby-Dick*.

In order to include the images on our web page, we need to download them to the local machine (as we did with the kitten image in Section 2.4):

```
$ curl -o images/sperm_whales.jpg \
>      -OL https://cdn.learnenough.com/sperm_whales.jpg
$ curl -o images/moby_dick.png -OL https://cdn.learnenough.com/moby_dick.png
```

The initial book report page is fairly long, which makes it an excellent exercise in reading and writing HTML. The result, which highlights a few especially important lines, appears in Listing 3.7. Note especially the use of `target="_blank" rel="noopener"` (introduced briefly in Section 2.3.1), which arranges to open links in a new browser tab.[4] There is also one intentional error in Listing 3.7; catching and fixing it is left as an exercise (Section 3.3.1).

Listing 3.7: The initial *Moby-Dick* book report.
moby_dick.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Moby Dick</title>
    <meta charset="utf-8">
  </head>
  <body>

    <!-- much here to be styled, will do so in last section -->
    <header>
      <h1>A Softcover Book Report</h1>
      <h2>Moby-Dick (or, The Whale)</h2>
    </header>

    <div>
      <p>
        The <a href="https://www.softcover.io/">Softcover</a> publishing platform
        was designed mainly for ebooks like the
        <a href="https://railstutorial.org/book"><em>Ruby on Rails Tutorial</em>
        book</a> and <a href="https://learnenough.com/html"><em>Learn Enough
        HTML to Be Dangerous</em></a>, but it's also good for making more
        traditional books, such as the novel <em>Moby-Dick</em> by Herman
        Melville (sometimes written as <em>Moby Dick</em>). We present below a
        short and affectionately irreverent book report on this classic of
        American literature.
      </p>
    </div>

    <a href="https://commons.wikimedia.org/wiki/File:Sperm_whale_pod.jpg">
      <img src="images/sperm_whales.jpg">
    </a>

    <div>

      <h3>Moby-Dick: A classic tale of the sea</h3>
```

[4] Or a new browser window. The exact behavior depends on the user's browser settings.

```
      <a href="https://www.softcover.io/read/6070fb03/moby-dick"
         target="_blank" rel="noopener">
        <img src="images/moby_dick.png" alt="Moby Dick">
      </a>

      <p>
        <a href="https://www.softcover.io/read/6070fb03/moby-dick"
           target="_blank" rel="noopener">
          <em>Moby-Dick</em></a>
          by Herman Melville begins with these immortal words:
      </p>

      <blockquote>
        <p>
          <span>Call me Ishmael.</span> Some years ago—never mind how long
          precisely—having little or no money in my purse, and nothing
          particular to interest me on shore, I thought I would sail about a
          little and see the watery part of the world. It is a way I have of
          driving off the spleen and regulating the circulation.
        </p>
      </blockquote>

      <p>
        After driving off his spleen (which <em>can't</em> be good for you),
        Ishmael then goes on in much the same vein for approximately one
        jillion pages. The only thing bigger than Moby Dick (who&mdash;<em>spoiler
        alert!</em>&mdash;is a giant white whale) is the book itself.
      </p>
    </div>
  </body>
</html>
```

In Listing 3.7, the **header** tag contains the **h1** and **h2** tags, and its importance will become apparent only when we add inline styles in Chapter 4. For now, what's important is that it's an abstract semantic tag used to label a part of the page, and has no immediate effect on the page's appearance.

Likewise, the **div** tag sets the page divisions apart, but won't have any effect until Chapter 4. We've also wrapped the iconic first line of *Moby-Dick*, "Call me Ishmael.", in a **span** tag in anticipation of styling it in Chapter 4. (The main difference between the **div** and **span** tags is that **div** is a block element, whereas **span** is an inline element (Box 3.1).) Finally, we've introduced the **blockquote** tag for quoting blocks of text.

In addition to illustrating the **header**, **div**, and **span** tags, Listing 3.7 also

shows an HTML *comment*, which appears as follows:[5]

```
<!-- much here to be styled, will do so in last section -->
```

This line, which foreshadows the styling steps in Chapter 4, is ignored by the browser, and is *not* visible on the rendered page, as (not) seen in Figure 3.9.[6]

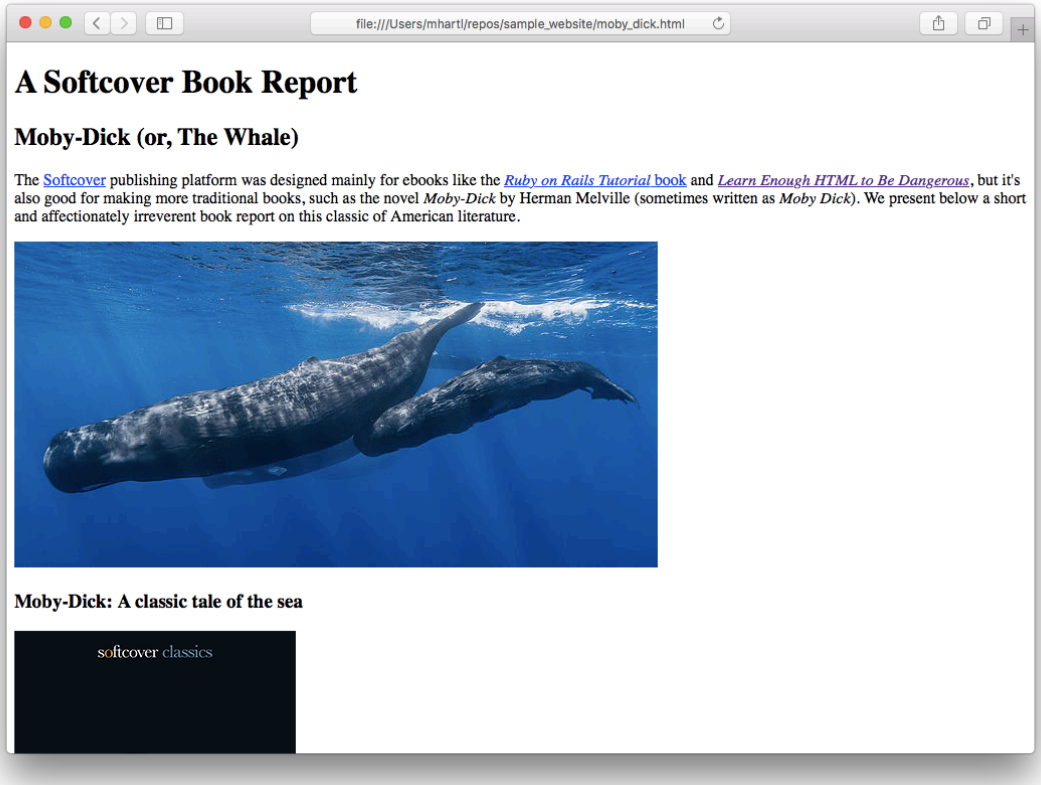


Figure 3.9: The initial Softcover book report on *Moby-Dick*.

[5] In Atom, you can toggle HTML comments using ⌘/ ("Command-slash"), which (as noted in *Learn Enough Text Editor to Be Dangerous*) works for source code as well. Because the text editor infers the file type from the filename extension (or from the shebang line), we can use ⌘/ to comment things out without having to remember the exact syntax for the file type we happen to be editing.

[6] The comment is sent over the wire to the browser, though, and is visible to anyone inspecting the HTML source of our page.

3.3.1 Exercises

1. Validate the HTML in Listing 3.7 and confirm that there's one warning and one error. Apply the fixes suggested by the validator and confirm that the new page validates with no warnings.

2. In the previous exercise, you should have found a warning that suggested setting the language of the page to English. Update the site's other pages with the same change. (This repetition of effort is inconvenient, and would be handled automatically by a templating system.)

3. Add **`header`**, **`div`**, and **`span`** to the tables in **`tags.html`**. *Hint*: Like **`div`**, **`header`** is a block element.

3.4 Lists

As part of our mini-report on *Moby-Dick*, we'd like to include a couple of lists with some of our observations about the book. As we'll see, HTML lists come in two basic types.

The first list will highlight our top three favorite things about *Moby-Dick*. Because we want these to be in rank-order, we'll use the *ordered list* tag **`ol`**:

```
<h4>Our top 3 favorite things about Moby Dick</h4>

<ol>
  <li>Vengeful whale</li>
  <li>Salty sailors</li>
  <li>The names "Queequeg" and "Starbuck"</li>
</ol>
```

Here the **`li`** tag indicates a **li**st element, and the result will be numbered in sequence:

```
1. …
2. …
3. …
```

The second list will contain some other miscellaneous musings. Because the order isn't important, we'll use the *unordered list* tag **ul**, together with the same **li** list element tag used for ordered lists:

```
<h4>Other things about Moby Dick</h4>

<ul>
  <li>
    Chapter after chapter (after chapter) of meticulous detail about whaling
  </li>
  <li>
    The story pretty much
    <a href="https://en.wikipedia.org/wiki/Essex_(whaleship)"
       target="_blank" rel="noopener">happened in real life</a>
  </li>
  <li>Mad sea captains are fun</li>
</ul>
```

As we'll see in a moment, unordered list elements are styled as bullet points by default:

```
• …
• …
• …
```

They are much more useful than this, though, and in practice unordered lists are used for significantly more than just making bullet points (as we'll see in *Learn Enough CSS & Layout to Be Dangerous*).

Putting these two lists together and adding them to the end of **moby_dick.html** gives Listing 3.8.

Listing 3.8: Adding lists to our *Moby-Dick* book report.
moby_dick.html

```
      .
      .
      .
      <h4>My top 3 favorite things about Moby Dick</h4>

      <ol>
        <li>Vengeful whale</li>
```

```
        <li>Salty sailors</li>
        <li>The names "Queequeg" and "Starbuck"</li>
      </ol>

      <h4>Other things about Moby Dick</h4>

      <ul>
        <li>
          Chapter after chapter (after chapter) of meticulous detail about
          whaling
        </li>
        <li>
          The story pretty much
          <a href="https://en.wikipedia.org/wiki/Essex_(whaleship)"
             target="_blank" rel="noopener">happened in real life</a>
        </li>
        <li>Mad sea captains are fun</li>
      </ul>
    </div>
  </body>
</html>
```

The result appears in Figure 3.10, which incidentally also shows the rendered **`blockquote`** that's not quite visible in Figure 3.9.

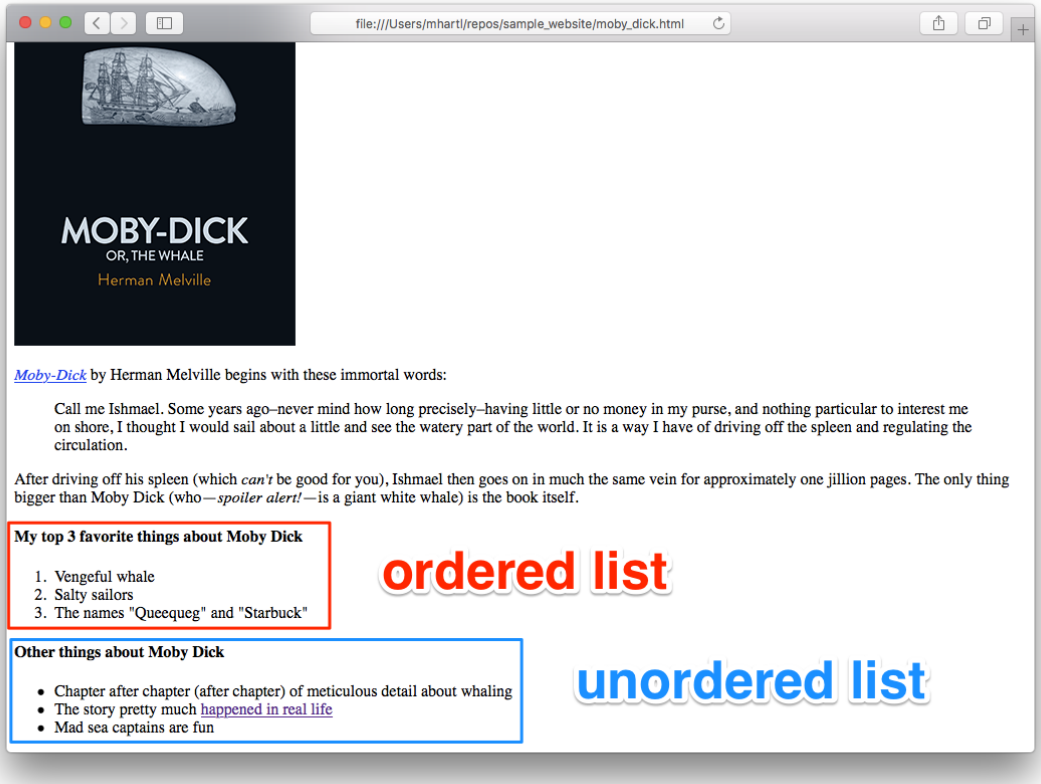


Figure 3.10: Ordered and unordered lists.

3.4.1 Exercises

1. Add `ol`, `ul`, and `li` tags to `tags.html`. Which are block elements and which are inline?

3.5 A navigation menu

Before moving on to styling our book report page in Chapter 4, we'll add a component common to most sites on the Web, yet one whose origins are mysterious: a navigation menu with links to all the pages on the site (Box 3.2). In

the process, we'll learn how to make links to pages on the current site instead of always linking to external websites. The navigation menu will also give us another chance to see how cumbersome it is to make websites without a templating system.

Box 3.2. Hacked together with Perl

Having watched the Web evolve from the start, I (Michael) knew how to make web pages from an early date, but I always wondered how it was that so many sites had the same menu on each page. I figured it must be some property of HTML I didn't know about, but even an early book on web design didn't cover it.

I remember thinking, you don't just hard-code the same menu on every page, do you? That's seems like an awful lot of duplicated effort. And what if you want to change it?

It turns out that no, well-made sites don't require you to hard-code the menu everywhere. In fact, as I realized much later, most such sites at the time were just "hacked together with Perl" (as the comic strip xkcd once humorously put it), but at the time it was a genuine mystery. (Although Perl is still in use, nowadays it's probably more common to stitch websites together using PHP, Python, JavaScript, or Ruby. The principle, though, remains the same.)

In the present tutorial, we're not in a position to solve this problem the Right Way™, so we *will* have to write everything by hand. But this is a feature, not a bug, because solving the problem by hand helps us appreciate why it should be solved by a computer instead.

We'll reveal the solution to this mystery, called a *templating system*, in *Learn Enough CSS & Layout to Be Dangerous*, and it's covered thoroughly in the *Ruby on Rails Tutorial* as well.

We'll start by adding a `div` containing three links to the index page (Listing 3.9). Note that we've adopted the common convention of referring to the index page as "Home".

Listing 3.9: Adding navigation links.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="utf-8">
  </head>
  <body>

    <div>
      <a href="index.html">Home</a>
      <a href="moby_dick.html">Moby Dick</a>
      <a href="tags.html">HTML Tags</a>
    </div>

    <h1>The Learn Enough Story</h1>
    .
    .
    .
```

We see from Listing 3.9 that the way to link to a local page simply involves setting the **`href`** attribute equal to the path to the file, which works exactly the same way as an **`img`** tag's **`src`** attribute (Section 2.4):

```
<a href="tags.html">HTML Tags</a>
```

The result appears in Figure 3.11.

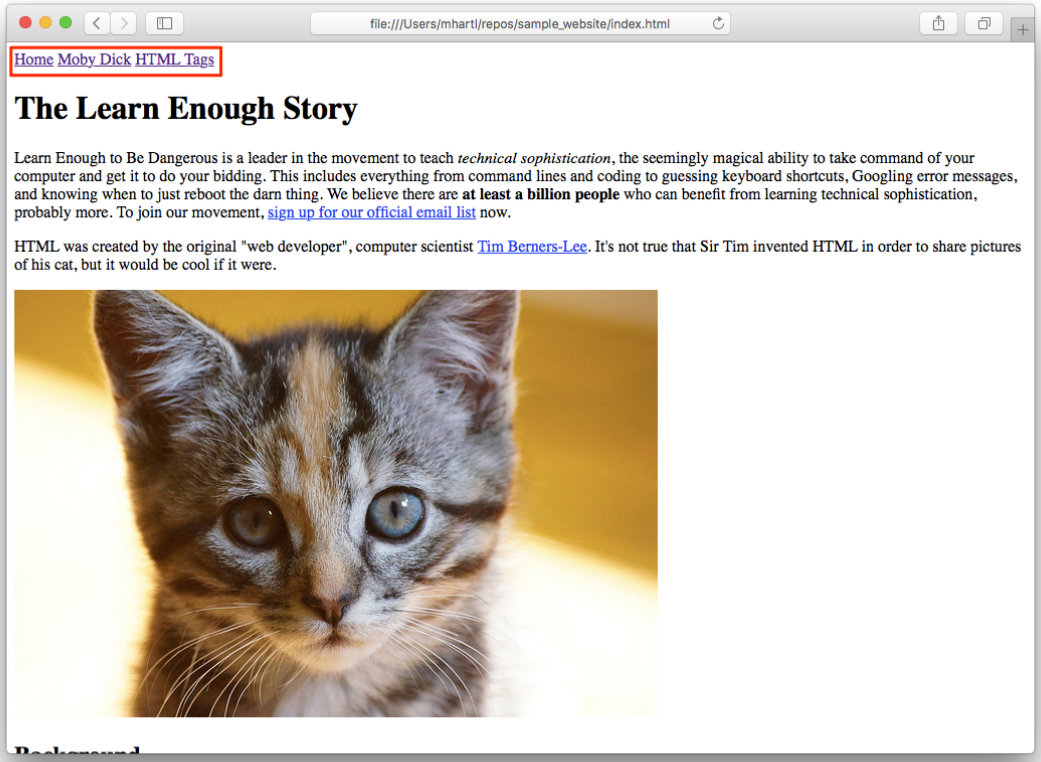


Figure 3.11: The navigation menu on the index page.

We can add the same menu to the HTML tags page with the markup in Listing 3.10. The result appears in Figure 3.12.

Listing 3.10: Adding the navigation menu to the HTML tags page.
tags.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>HTML Tags</title>
    <meta charset="utf-8">
  </head>
  <body>
```

```
    <div>
      <a href="index.html">Home</a>
      <a href="moby_dick.html">Moby Dick</a>
      <a href="tags.html">HTML Tags</a>
    </div>

    <h1>Important HTML tags</h1>
    .
    .
    .
```

Adding the menu to the page on Moby Dick is left as an exercise.

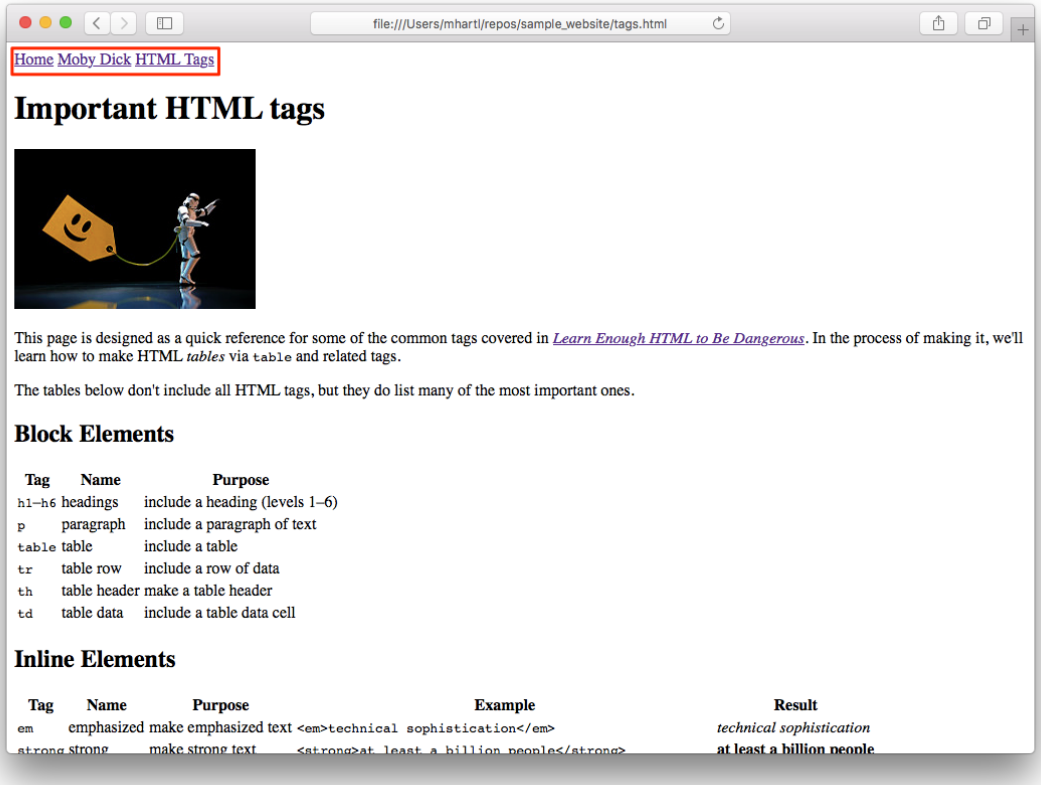


Figure 3.12: The navigation menu on the HTML tags page.

By the way, you may have noticed that it would probably be more natural to order the navigation in order of the pages' introduction, i.e., Home, Tags,

Moby Dick. Changing this order is also left as an exercise, mainly so that you feel the pain of making the same change on three different pages—a pain that is alleviated by the templating systems covered in *Learn Enough CSS & Layout to Be Dangerous*. Either before or after doing that exercise, I suggest adding and committing the changes and then pushing up to GitHub Pages:

```
$ git add -A
$ git commit -m "Add a Moby Dick page and a menu"
$ git push
```

3.5.1 Exercises

1. Add the menu links to the Moby Dick page.
2. In the menu links, change the order of the links so that the tags page comes second. How many files do you have to edit?

Chapter 4

Inline styling and CSS

Now that we've added the content for our pages and blocked out the basic structure, we're ready to add some styling. Our basic approach involves adding *inline styles*, which means putting styling commands directly inside the site's HTML tags. This approach will allow us to understand exactly what our styling is doing on a per-element basis, giving us immediate results.

Inline styles are generally considered a bad practice, though (Box 4.1). En route to fixing this issue, we'll get just a *tiny* taste of Cascading Style Sheets—the design language of the Web—by converting inline styles for one of our pages to CSS. This in turn will provide an essential foundation for front-end design and development (*Learn Enough CSS & Layout to Be Dangerous*) and for writing interactive websites using JavaScript (*Learn Enough JavaScript to Be Dangerous*).

Box 4.1. Separating style and content

If adding styles directly to elements gives you the results that you want, why is it considered bad practice to do it?

One reason is that when your styling is kept in a separate file from your content and layout, the HTML files are cleaner and easier to maintain. This doesn't make a huge difference when one developer is working on a single page, but when you have multiple developers all making changes to multiple pages, it quickly becomes

a nightmare to make changes efficiently and consistently. Imagine if you decided that you didn't like the size of a font on your site, and had to go around to every place on every page that needed a new style. If this were the only way, no one would ever do it.

Another reason for separating style from content is that it allows for much greater flexibility and efficiency when applying style rules to multiple elements. Instead of having to style each individual tag, we can apply styling to all elements on the entire site, or just to certain elements that we choose.

For instance, it's possible to make a table span the full width of the page using the rule `width: 100%`. If we wanted *every* table on the site to have the same styling, we would have to copy and paste that into every `<table>` tag on every page:

```
<table style="width: 100%;">
<table style="width: 100%;">
<table style="width: 100%;">
```

With CSS, we can tell the browser to style every table with a small bit of code.

```
table {
  width: 100%;
}
```

For an extensive website with lots of different pages, elements, and style rules, this can represent a massive gain in simplicity and efficiency.

Finally, every page on the Internet is sitting on a remote server somewhere sending data to the users visiting the site. Every word or line of code that you add to a page is something extra that needs to be downloaded over the network. Cutting out repetitive elements on a page makes them smaller, and thus helps sites load faster as well.

4.1 Text styling

As mentioned in Section 3.3, our design efforts will focus on the *Moby-Dick* book report page, which stands to gain the most from changing the default HTML styling. We'll begin by adding a little styling to the quote of the first paragraph from *Moby-Dick*. Recall from Listing 3.7 that the quote uses the `blockquote` tag, which is set apart from the surrounding text by extra space and indentation (Figure 4.1).

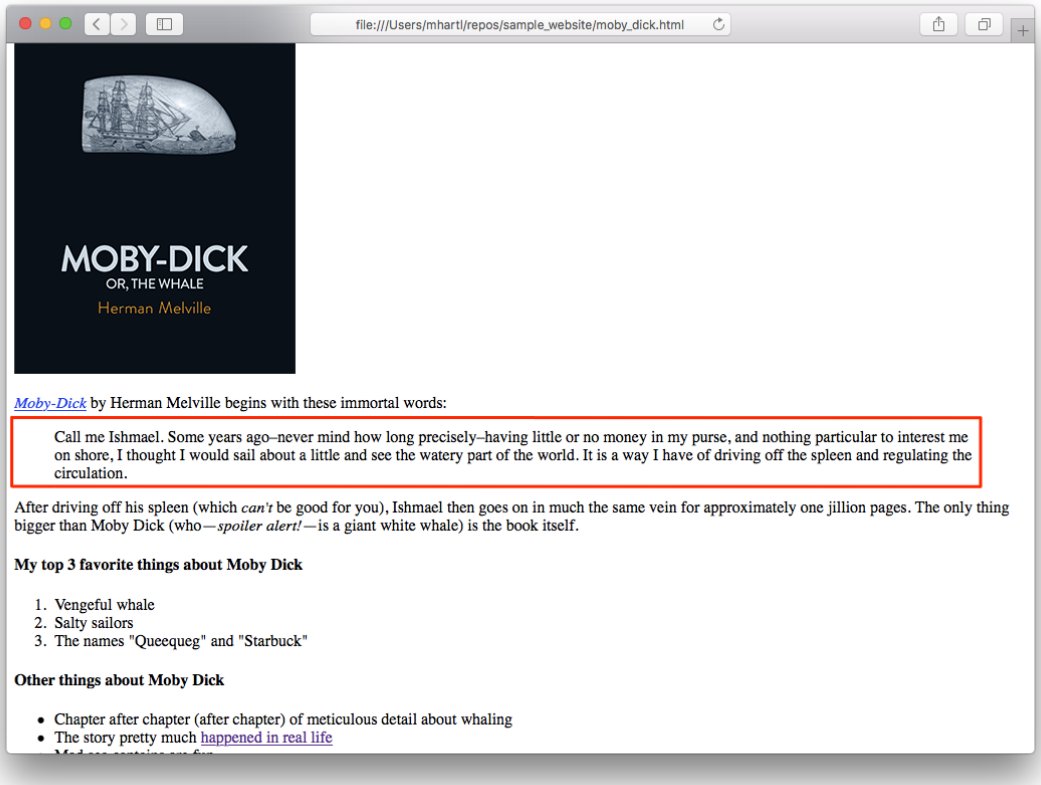


Figure 4.1: The default blockquote styling.

To make the quote stand out even more, let's change the font style to *italics* while increasing the font size to 20 pixels (`20px`). The way to do this is with

the **style** attribute, which can be added to virtually any HTML tag. In this case, we'll change the **font-style** and the **font-size** as follows:

```
<blockquote style="font-style: italic; font-size: 20px;">
```

Note that the styling rules after **style=** is a single string of characters, with each individual style separated from the others by a semicolon **;**. (The final semicolon isn't technically necessary, but including it is a good practice since it lets us add additional styles later on without having to remember to add a semicolon.)

Taking this idea and editing **moby_dick.html** leads to the HTML shown in Listing 4.1. The result appears in Figure 4.2.

Listing 4.1: Styling the blockquote.
moby_dick.html

```
      .
      .
      .
      <blockquote style="font-style: italic; font-size: 20px;">
        .
        .
        .
      </blockquote>
      .
      .
      .
```

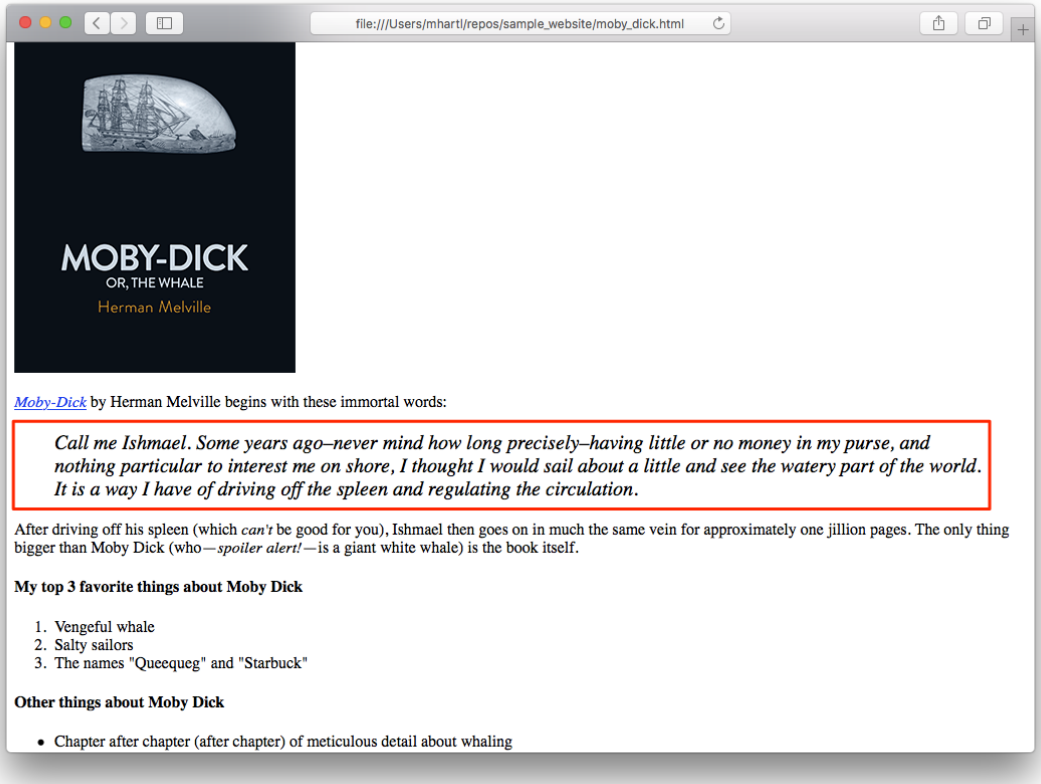


Figure 4.2: A styled blockquote.

Next, we'll add some styling to the famous first line "Call me Ishmael." We'll first revert the font style back to normal (which is how you emphasize text when the surrounding text is already in italics). Then we'll make the font even bigger that the `20px` used for the rest of the quote, while also making it **bold**. Finally, we'll change the color to an attention-grabbing **red**.

Ordinarily, there would be no way to style just the line "Call me Ishmael." If we had written it as

```
Call me Ishmael. Some years ago…
```

at this point we'd be out of luck. Recall from Listing 3.7, though, that the opening line is wrapped in a `span` tag:

```
<span>Call me Ishmael.</span> Some years ago…
```

The reason for this is that we correctly anticipated wanting to style it later on. (In practice we can't always anticipate such things, but of course we can wrap the relevant text in a **span** tag as needed.)

Although the **span** tag doesn't really do anything by itself, we *can* add styles to it, as follows:

```
<span style="font-style: normal; font-size: 24px; font-weight: bold;
color: #ff0000;">Call me Ishmael.</span>
```

Here we've combined **font-style**, **font-size**, **font-weight**, and **color** to obtain the results outlined above. These are but a few of the many style properties available, which you can learn about at sites like w3schools. Meanwhile, the color **ff0000** is a hexadecimal code for red, as discussed in Box 4.2.

Box 4.2. HTML and hexadecimal color

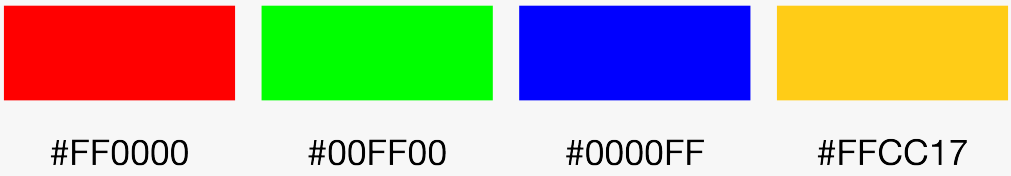
HTML colors are typically indicated using a system known as *hexadecimal RGB* (for "red, green, blue"), which gives us a flexible way to specify colors with fine-grained precision.

Hexadecimal refers to the use of *base 16*, which uses 16 symbols from `0` to `F` to specify the decimal numbers 0 through 15:

0	1	2	3	4	5	6	7	8	9	10	11	12	13	14	15
0	1	2	3	4	5	6	7	8	9	A	B	C	D	E	F

Just as base 10 lets us count from 0 to $10^2 - 1 = 99$ using only two digits, hexadecimal lets us count from 0 to $16^2 - 1 = 255$ using `00` = 0 up to `FF` = 255.

A computer monitor displays colors by combining red, green, and blue pixels together, which represent the three additive primary colors. If all three colors are turned on, which would look like `#FFFFFF`, the screen will look white. Conversely, if all three colors are turned off, which is `#000000`, the screen will look black. Intermediate combinations of the three colors combine to produce virtually all the colors you can see:



For convenience, HTML also supports a shorthand that can stand in for the full hex string in some cases. For example, if the numbers are repeated, as in `#222222`, `#bbbbbb`, or `#aa22ff`, you can shorten the whole number to just three digits, like this: `#222`, `#bbb`, or `#a2f`. (Note that we've switched to lowercase letters, which is more common in modern HTML code, but is exactly equivalent the upper-case versions shown above.) When the browser sees only three digits, it fills in the missing ones automatically.

The hexadecimal color system might seem confusing at first, but you'll find that you quickly come to understand how the three values work together to make different colors (and different shades of those colors). To learn more, we suggest playing around with a color picker to see what kinds of colors you can make.

Including the styles above in **`moby_dick.html`** gives us the code in Listing 4.2, with the result shown in Figure 4.3.

Listing 4.2: Adding style to the opening span.
moby_dick.html

```
.
.
.
<blockquote style="font-style: italic; font-size: 20px;">
  <p>
    <span style="font-style: normal; font-size: 24px; font-weight: bold;
    color: #ff0000;">Call me Ishmael.</span>
    Some years ago—never mind how long precisely—having little or
    no money in my purse, and nothing particular to interest me on shore,
    I thought I would sail about a little and see the watery part of the
    world. It is a way I have of driving off the spleen and regulating the
    circulation.
  </p>
</blockquote>
.
.
.
```

As a final change, we'll align the text of the book report headers so that it's centered in the page. The way to do this is with the **`text-align`** property, as shown in Listing 4.3. (Note that Listing 4.3 also removes the comment from Listing 3.7 since it's now obsolete.)

Listing 4.3: Centering the headings.
moby_dick.html

```
.
.
.

<header>
  <h1 style="text-align: center;">A Softcover Book Report</h1>
  <h2 style="text-align: center;">Moby-Dick (or, The Whale)</h2>
</header>
.
.
.
```

The result appears in Figure 4.4.

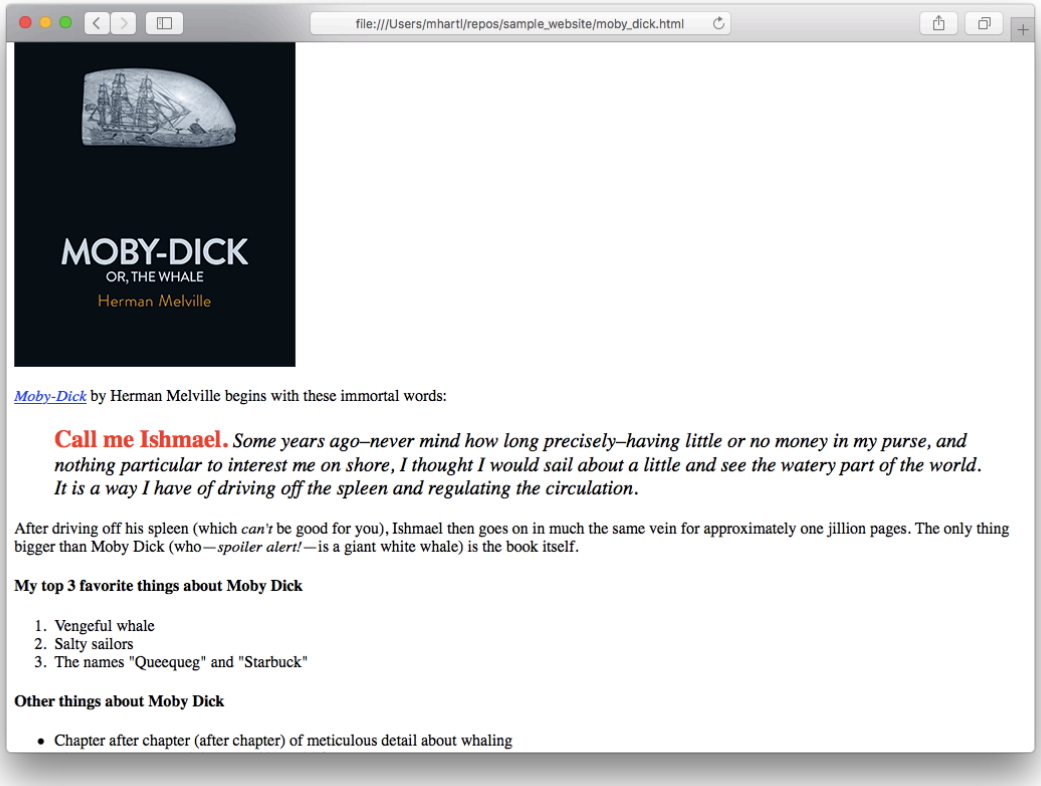


Figure 4.3: Making the opening line really pop.

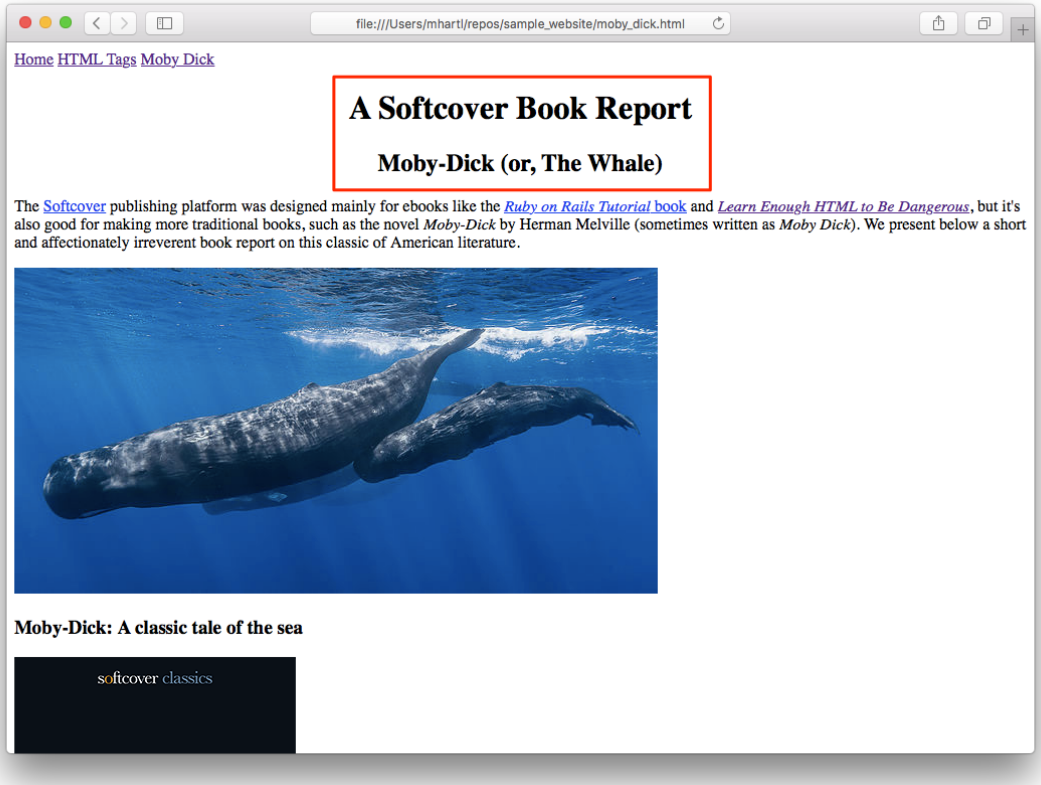


Figure 4.4: Centered headings.

4.1.1 Exercises

1. Verify that `color: red;` has the same effect as `color: #ff0000;`. What are the advantages of each approach?

2. What would you guess the color of `#cccccc` is? Temporarily modify the color of the `span` in Listing 4.2 to check your guess. How does it differ from `#ccc`?

4.2 Floats

Now that we've learned how to move text around, let's look at how to adjust how other elements on the page can be moved. We'll start by shrinking the size of the cover image down a little, and then we'll arrange for the text to flow around it as if it were part of the paragraphs and blockquote.

We'll start by using the **height** attribute to the **img** tag to restrict the height to 200 pixels:

```
<img src="images/moby_dick.png" alt="Moby Dick" height="200px">
```

Several caveats are in order here. First, although inline resizing is still fairly common, using CSS is the best practice. Second, resizing the image this way (whether inline or with CSS) affects only the image *display*, and the entire image still needs to be downloaded from the web server, so this technique should be used only for fairly minor resizings.[1] (If you've ever visited a web page where a seemingly tiny image takes *forever* to download, this is probably the reason why.) Finally, if you go this route you should use *either* **height** *or* **width**, but not both, as the combination forces the browser to attempt to respect both numbers, which can result in weird image resizing effects (Figure 4.5).

[1] If you need to resize the image itself and don't have access to Photoshop, you can use the free Skitch utility for resizing, cropping, and simple annotation.

Figure 4.5: Bad image resizing is bad.

In order to get the text to flow around the image, we need to use a style technique called *floating*. The idea is that when you set an element to "float" to the left or right (there is no center), all the inline content around it will flow around the floated element. To see this in action, all we need to do is add **`style="float: left;"`** to the image attributes:

```
<img src="images/moby_dick.png" alt="Moby Dick" height="200px"
style="float: left;">
```

Inserting this into the book report page gives Listing 4.4, with the result shown in Figure 4.6.

Listing 4.4: Resizing and floating an image.
moby_dick.html

```
      .
      .
      .
      <h3>Moby-Dick: A classic tale of the sea</h3>

      <a href="https://www.softcover.io/read/6070fb03/moby-dick"
         target="_blank" rel="noopener">
        <img src="images/moby_dick.png" alt="Moby Dick" height="200px"
```

```
        style="float: left;">
      </a>
      .
      .
      .
```

In Figure 4.6, the image is effectively treated like text, with the normal text now flowing up and to its side. We'll see in Section 4.5 that the **`float`** attribute also arranges for the text to flow past the image as well.

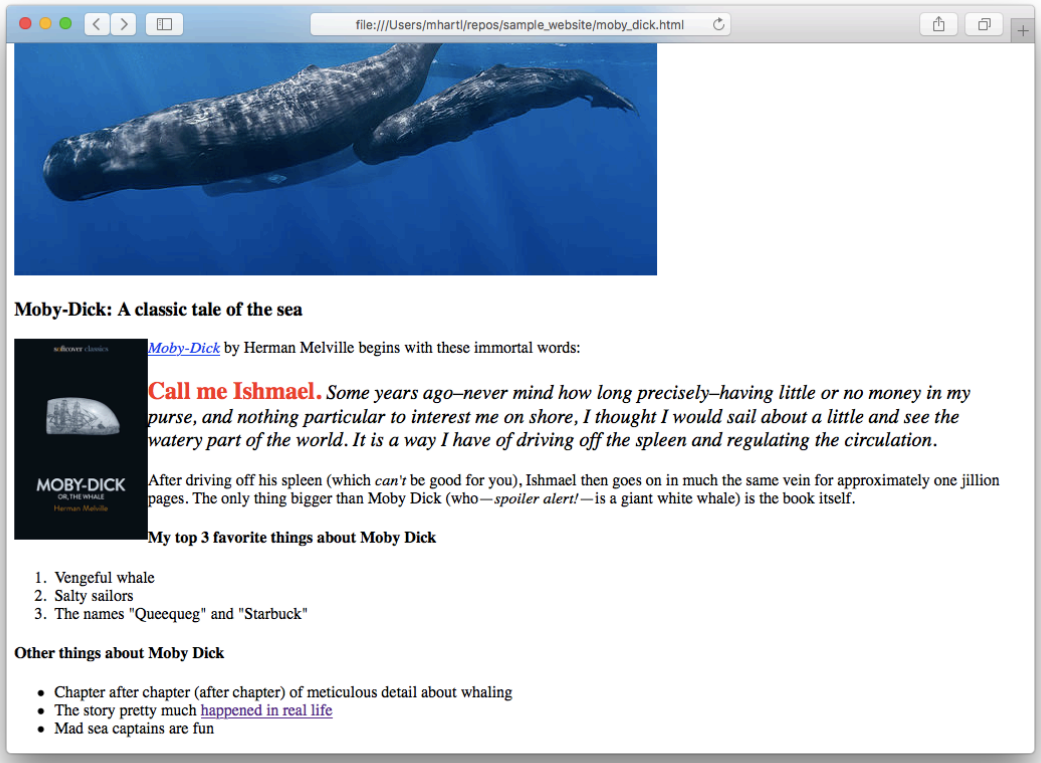


Figure 4.6: A nicely resized & floated image.

4.2.1 Exercises

1. What happens if you change `float: left` to `float: right` in Listing 4.4?

4.3 Applying a margin

Even though we've floated the image, the page still looks a little weird, with the text smashed up against the book cover. To make it look better, we'll add a *margin* of empty space to the right of the image.

Margins are one of three styles that can be applied to the imaginary boxes that contain HTML content, the others being *padding* (empty space inside the box) and *borders* (a line around the box). We'll talk more about these styles in the context of the *box model* covered in *Learn Enough CSS & Layout to Be Dangerous*, but we can accomplish our immediate goal by applying a margin using inline styling. (We'll see an example of padding in Section 4.5.)

We'll start with the simplest kind of margin declaration, which looks like `margin: 40px;`:

```
<img src="images/moby_dick.png" alt="Moby Dick" height="200px"
style="float: left; margin: 40px;">
```

If we add this to the `img` tag (Listing 4.5), everything surrounding the image moves 40 pixels in each direction, as shown in Figure 4.7.

Listing 4.5: Adding an image margin.
moby_dick.html

```
      .
      .
      .
      <h3>Moby-Dick: A classic tale of the sea</h3>

      <a href="https://www.softcover.io/read/6070fb03/moby-dick"
         target="_blank" rel="noopener">
        <img src="images/moby_dick.png" alt="Moby Dick" height="200px"
        style="float: left; margin: 40px;">
```

```
        </a>
        .
        .
        .
```

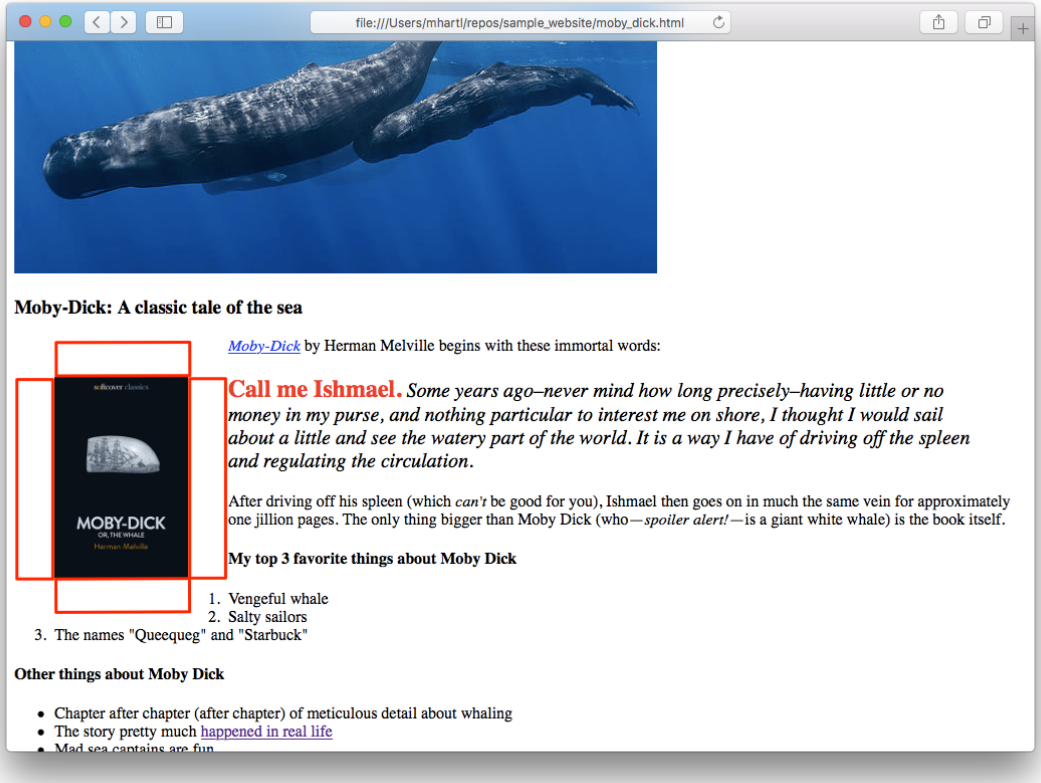


Figure 4.7: Close, but not quite!

Figure 4.7 shows that we've gotten closer to our goal by putting some space between the text and the image, but styling still isn't quite what we want. The reason is that writing `margin: 40px;` applies the margin in *all* directions, but we really only want the margin on the right side of the image, to separate it from the text.

The most general way to control where the margin goes is to give the `margin`

attribute *four* values, corresponding to the top, right, bottom, and left of the box (Figure 4.8).

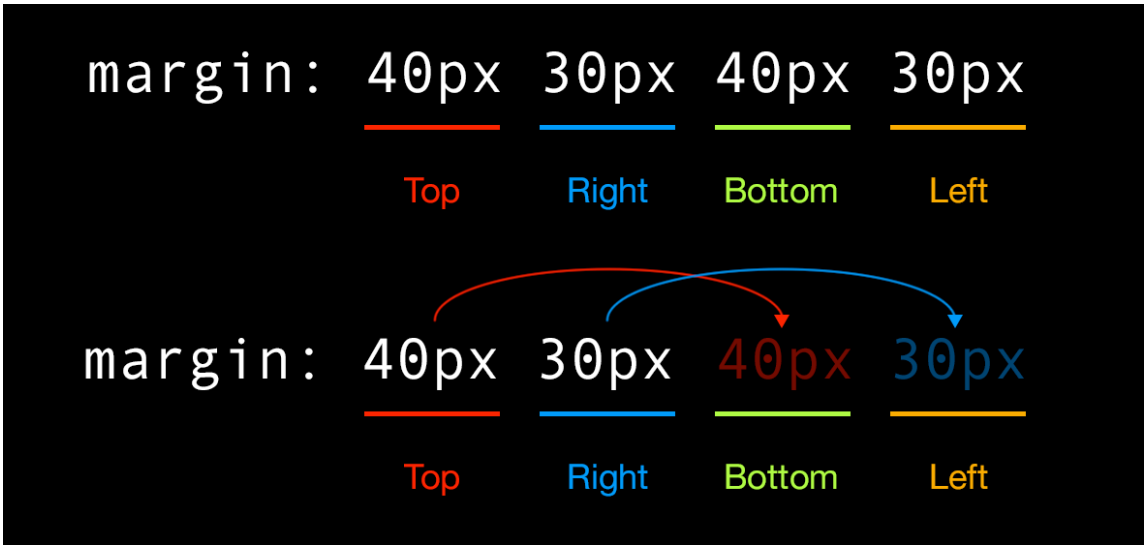


Figure 4.8: Think of the four values as going clockwise from the top.

For example, to get margins of 40, 30, 20, and 10 pixels going around an image (clockwise from the top), we could use this **`style`** attribute:

```
<img src="…" style="margin: 40px 30px 20px 10px;">
```

In the present case, we want only a right margin, so we can set the other three sides to **`0px`** (or just **`0`** for short):[2]

```
<img src="images/moby_dick.png" alt="Moby Dick" height="200px"
style="float: left; margin: 0 40px 0 0;">
```

Applying this to the full source of the book report page gives Listing 4.6.

[2] We could use the attribute **`style="margin-right: 40px;"`** to achieve the same effect, but specifying all four margins is the dominant convention and so is worth learning.

Listing 4.6: Adding only a right margin.
moby_dick.html

```
      .
      .
      .
      <h3>Moby-Dick: A classic tale of the sea</h3>

      <a href="https://www.softcover.io/read/6070fb03/moby-dick"
         target="_blank" rel="noopener">
        <img src="images/moby_dick.png" alt="Moby Dick" height="200px"
        style="float: left; margin: 0 40px 0 0;">
      </a>
      .
      .
      .
```

The result of Listing 4.6 shows exactly the result we want, with a margin applied only on the right.

4.3.1 Exercises

1. How does the margin in Listing 4.6 change if we replace the margin with `margin-right: 40px`?

2. Add the style rule `padding: 10px;` to the `td` elements for the first two block elements in the table in `tags.html`. How annoying would it be to add them to every `td`? How annoying is changing from `10px` to `20px` everywhere? (This is one reason why in real life you always use CSS.)

4.4 More margin tricks

There are a couple of more margin tricks worth mentioning, both of which we can immediately put to good use. First, in addition to the shorthand `margin: 40px` (with only one value), it's possible to include only two values:

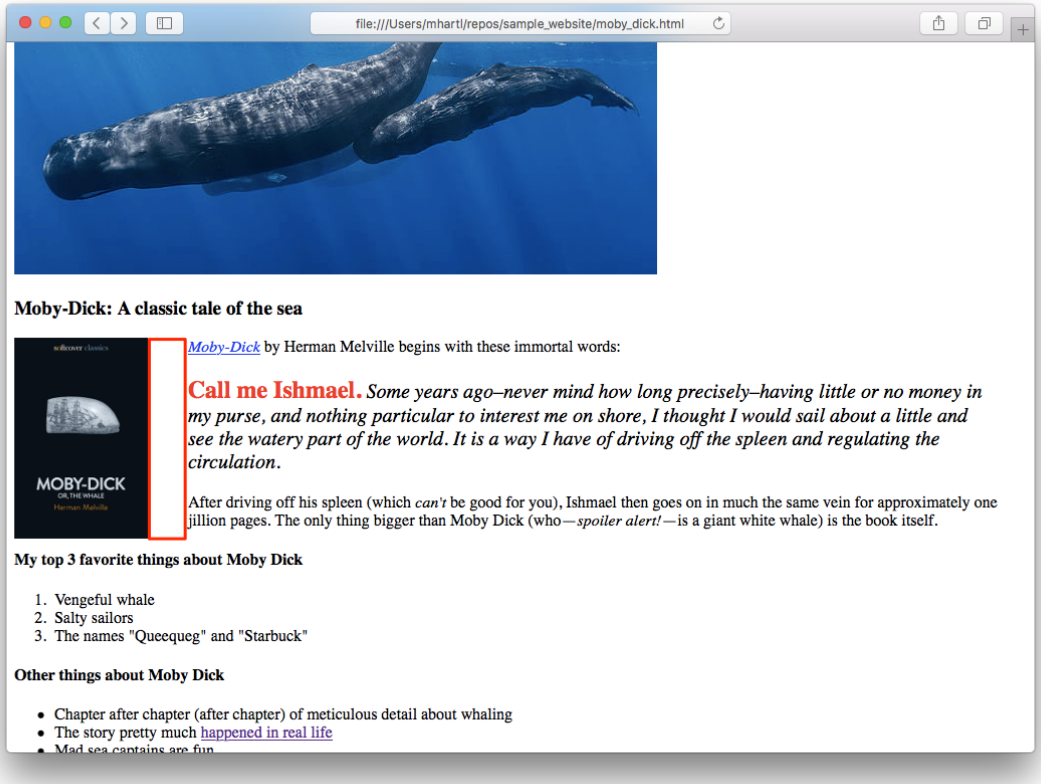


Figure 4.9: Much better!

```
margin: 20px 40px;
```

As illustrated in Figure 4.8, this syntax sets the top and bottom margins to 20px and the left and right margins to 40px, so it is equivalent to

```
margin: 20px 40px 20px 40px;
```

This shorthand also works with just three values, like this: **`margin: 20px 10px 40px;`**. This is missing the last value, which (as seen in Figure 4.8) is the left margin. In this case, it will be filled in automatically from its opposite across the box (in this case, **`10px`**).

We can apply this to the header for the book report page by adding a bottom margin via **`margin: 0 0 80px`** as shown in Listing 4.7.

Listing 4.7: Centering the book report headers.
moby_dick.html

```
    .
    .
    .
    <header style="text-align: center; margin: 0 0 80px;">
      <h1>A Softcover Book Report</h1>
      <h2>Moby-Dick (or, The Whale)</h2>
    </header>
    .
    .
    .
```

Note that we've also taken this opportunity to hoist the style **`text-align: center`** into the **`header`** tag. The result appears in Figure 4.10.

The second margin trick is the use of **`margin: auto;`**, which inserts a margin with a size that is automatically the same on all sides. Its most common application is probably in the rule

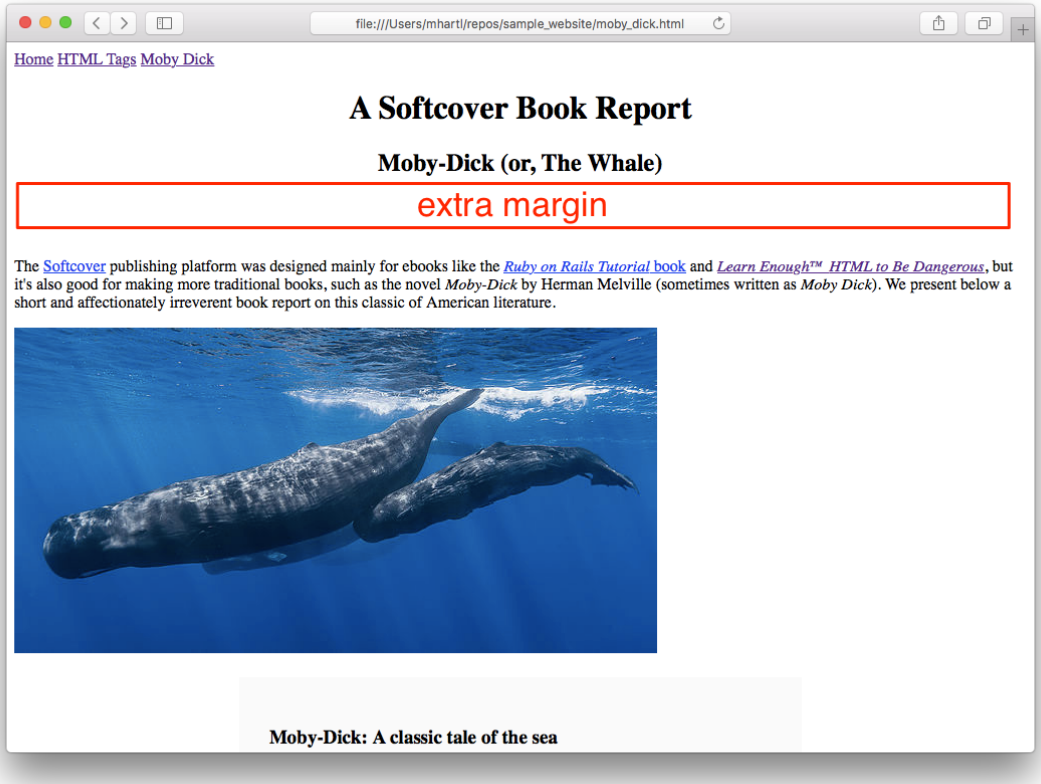


Figure 4.10: Adding a margin under the header.

```
margin: 0 auto;
```

which arranges for no top or bottom margin and automatic margins on the left and right. The result of equal left and right margins is that the element is *centered*, which is especially useful for elements like images that can't be centered using the `text-align: center;` rule we saw in Listing 4.3.

One restriction of `margin: 0 auto` is that it works only on block elements, but recall from Box 3.1 that the `img` tag is an inline element. We can fix this with the style `display: block;`, which overrides the default. Putting this together with the margin rule leads to Listing 4.8, with the result shown in Figure 4.11.

Listing 4.8: A centered image.
moby_dick.html

```
    .
    .
    .
    <a href="https://commons.wikimedia.org/wiki/File:Sperm_whale_pod.jpg">
      <img src="images/sperm_whales.jpg"
      style="display: block; margin: 0 auto;">
    </a>
    .
    .
    .
```

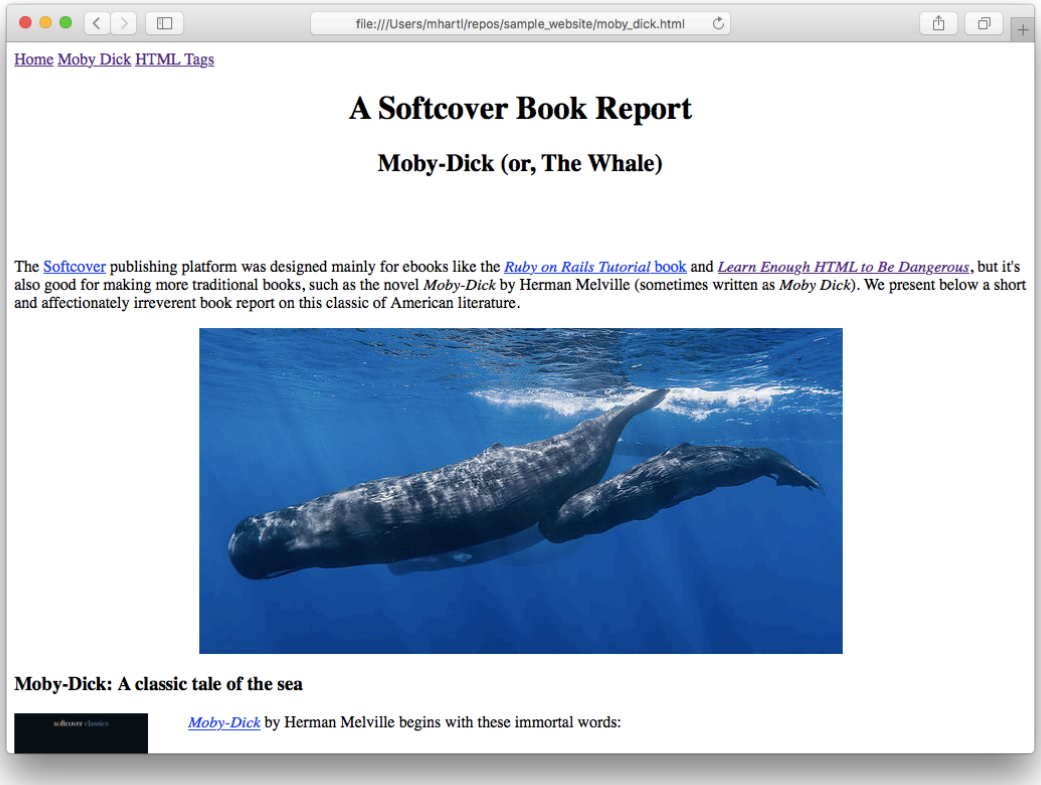


Figure 4.11: A centered image.

4.4.1 Exercises

1. What happens if you use `margin: 0 auto` for the book cover image (together with `display: block`) without changing the float rule? What does this tell you about the precedence of the two rules?

4.5 Box styling

So far, the changes we've made have had a relatively minor impact on the appearance of the book report page. In this section, we'll see how a set of only

four style rules can make a surprisingly big difference.

Recall from Listing 3.7 that we wrapped the bulk of the report in a **`div`** tag, which defines a block element that doesn't get any default styling from the browser, thereby making it perfect to use as a wrapper for styling other content. In this case, we'll use the **`width`** style to restrict the size of the main report to 500 pixels, and it turns out this lets us use the automatic margin trick from Section 4.4 to center it using **`margin: 20px auto`** (thereby also putting a 20 pixel margin on the top and bottom). Finally, we'll combine a **`padding`** rule with a change in the **`background-color`** using the hexadecimal color convention covered in Box 4.2. The resulting style rules appear in Listing 4.9.

Listing 4.9: Adding styling to the book report box.
moby_dick.html

```
    .
    .
    .
    <div style="width: 500px; margin: 20px auto; padding: 30px;
    background-color: #fafafa;">

      <h3>Moby-Dick: A classic tale of the sea</h3>
    .
    .
    .
```

Comparing the before (Figure 4.12) and after (Figure 4.13) shows what a difference a few style rules can make.

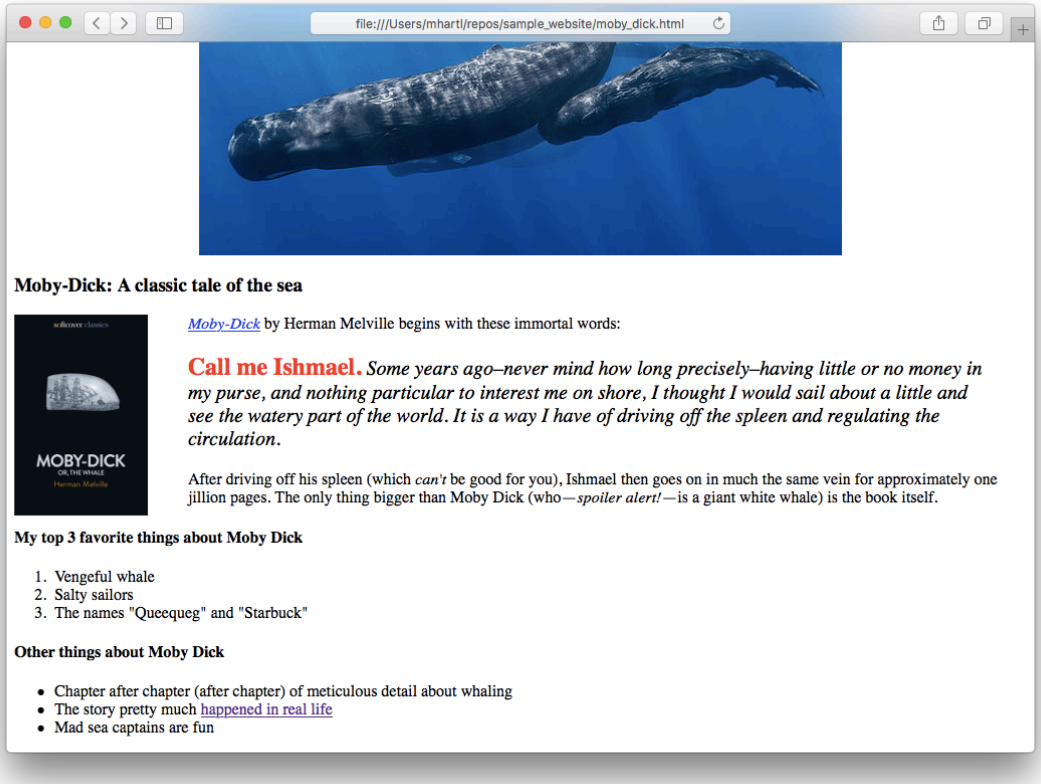


Figure 4.12: The report box before adding styling.

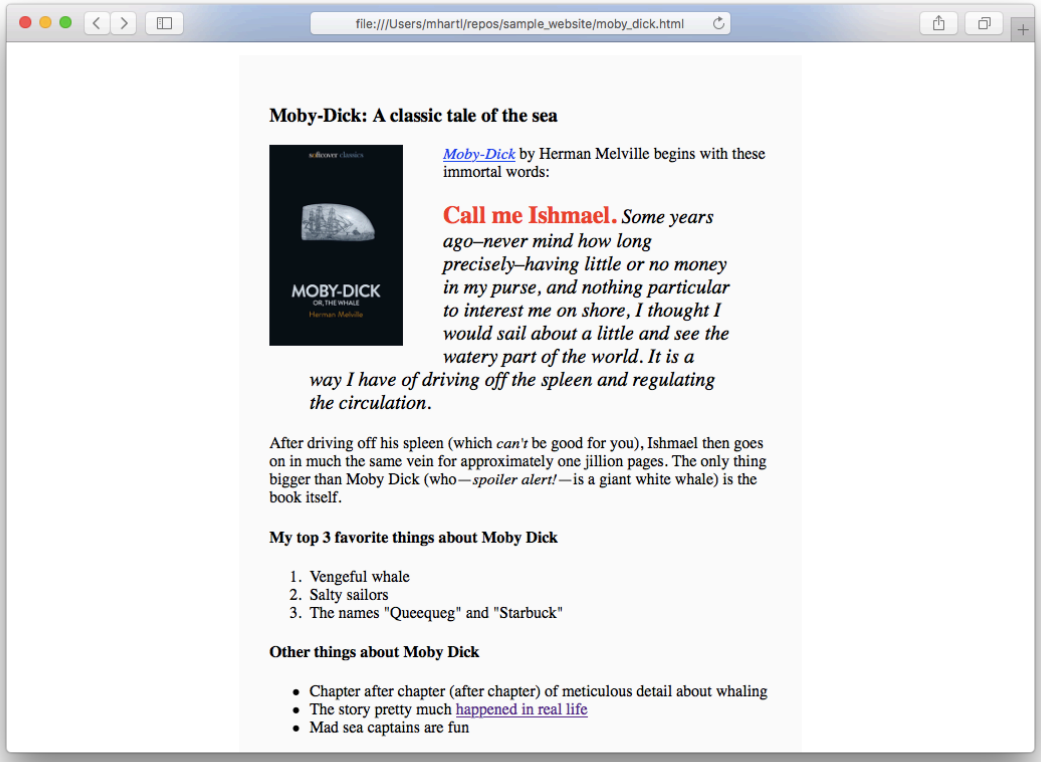


Figure 4.13: The report box after adding styling.

As we see in Figure 4.13, the report content has now been set apart from the rest of the page in a styled box.

To recap what happened, we set a width for the box, and because of this we were able to set the left and right margins to **`auto`**. Then we added padding to the box, which pushed the content inside away from the edges. (Investigating the difference between padding and margins is left as an exercise (Section 4.5.1).) We also added a light gray background color with the hexadecimal code **`#fafafa`** (Box 4.2). (Don't worry about trying to visualize the color corresponding to a hex code; that's what color pickers are for.) Finally, because of the narrower width, the text of the *Moby-Dick* quote now flows around the floated cover image, thereby fulfilling the promise made at the end of Sec-

tion 4.2.

4.5.1 Exercises

1. Temporarily change **`padding`** to **`margin`** in Listing 4.9. What difference does this make in the appearance?

2. Add and style a blockquote with padding and background color as shown in Listing 4.10. Fill in **`TAG`** with the right level tag for that location in the document, and replace **`FILL_IN`** with a reasonable color of your choice. The result for one color choice appears in Figure 4.14.

Listing 4.10: Styling a famous quotation.
index.html

```
    .
    .
    .
    <h1>The Learn Enough Story</h1>
    .
    .
    .
    <img src="images/kitten.jpg" alt="An adorable kitten">

    <TAG>Quotations</TAG>

    <p>
      In addition to hosting most of the world's supply of kitten videos, the
      Web is also full of inspiring quotes, perhaps none more so than this one:
    </p>

    <blockquote style="padding: 2px 20px; background: #FILL_IN;">
      <p>
        <em>Don't believe every quote you read on the Internet.</em>
        <br>
        &mdash;Abraham Lincoln
      </p>
    </blockquote>

    <h2>Background</h2>
    .
    .
    .
```

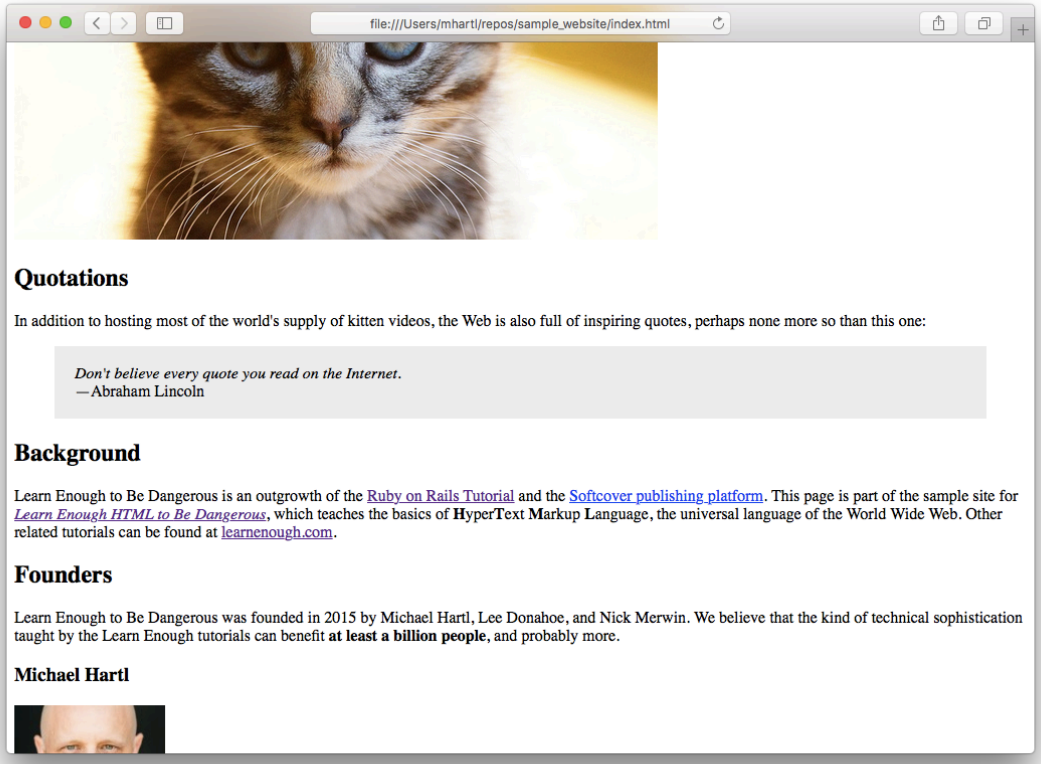


Figure 4.14: Styling a famous quote from an American president.

4.6 Navigation styling

As a final change to our sample website, we'll make a change across all three pages by adding styling to the navigation menu added in Section 3.5. In the process, we'll yet again feel the pain of having to make the same change in several places, further preparing us to appreciate the value of the template system developed in *Learn Enough CSS & Layout to Be Dangerous*.

To style the nav menu, we'll first move it from the top left of the page to the more conventional top right. This will involve adding a style rule to the `div` tag that wraps the whole menu (Listing 3.9). At the same time, we'll add a left

margin to the second and third nav links in order to improve the spacing. The changes to the book report page appear in Listing 4.11, and the result appears in Figure 4.15.

Listing 4.11: Styling the navigation menu on the book report page.
moby_dick.html

```
.
.
.
<div style="text-align: right;">
  <a href="index.html">Home</a>
  <a href="tags.html" style="margin: 0 0 0 10px;">HTML Tags</a>
  <a href="moby_dick.html" style="margin: 0 0 0 10px;">Moby Dick</a>
</div>
.
.
.
```

Of course, we're not done yet, as we need to make the same edits to the nav menu on the Home page (Listing 4.12) and HTML tags page (Listing 4.13).

Listing 4.12: Styling the navigation menu on the Home page.
index.html

```
.
.
.
<div style="text-align: right;">
  <a href="index.html">Home</a>
  <a href="tags.html" style="margin: 0 0 0 10px;">HTML Tags</a>
  <a href="moby_dick.html" style="margin: 0 0 0 10px;">Moby Dick</a>
</div>
.
.
.
```

Listing 4.13: Styling the navigation menu on the HTML tags page.
tags.html

```
.
.
```

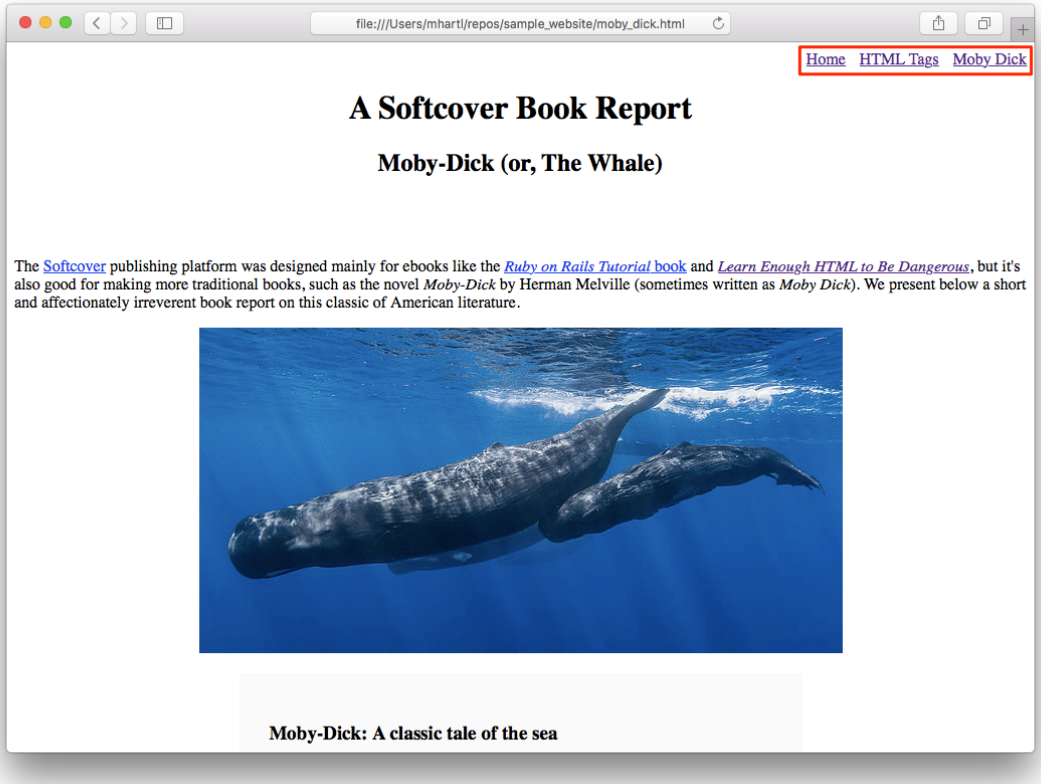


Figure 4.15: The styled navigation on the book report page.

```
.
<div style="text-align: right;">
  <a href="index.html">Home</a>
  <a href="tags.html" style="margin: 0 0 0 10px;">HTML Tags</a>
  <a href="moby_dick.html" style="margin: 0 0 0 10px;">Moby Dick</a>
</div>
.
.
.
```

The results on the nav menu are exactly the same as shown in Figure 4.15. This is not surprising given that the changes represented by Listing 4.11, Listing 4.12, and Listing 4.13 are *all identical*. This sort of repetition is cumbersome and error-prone—definitely a Bad Thing. As mentioned several times before, we'll solve this problem with a templating system in *Learn Enough CSS & Layout to Be Dangerous*.

4.6.1 Exercises

1. Change **`margin: 0 0 0 10px;`** to **`margin-left: 10px`** in Listing 4.13. What if anything changes in the appearance?

2. It's conventional for navigation links not to change color after being followed, and they also look better if they're not underlined like normal links. Using your Google-fu and the w3schools reference, guess the style rules for making these changes, and apply them to each element in the menu. *Hint*: The property you'll have to modify to remove underlining is **`text-decoration`**. The result should look something like Figure 4.16.

3. Add a new table for the "document tags" that define the properties of the document. Include **`html`**, **`head`**, **`body`**, **`title`**, and **`meta`**.

4. Add any missing tags to **`tags.html`**. (By my count there are five after including the tags from the previous exercise.)

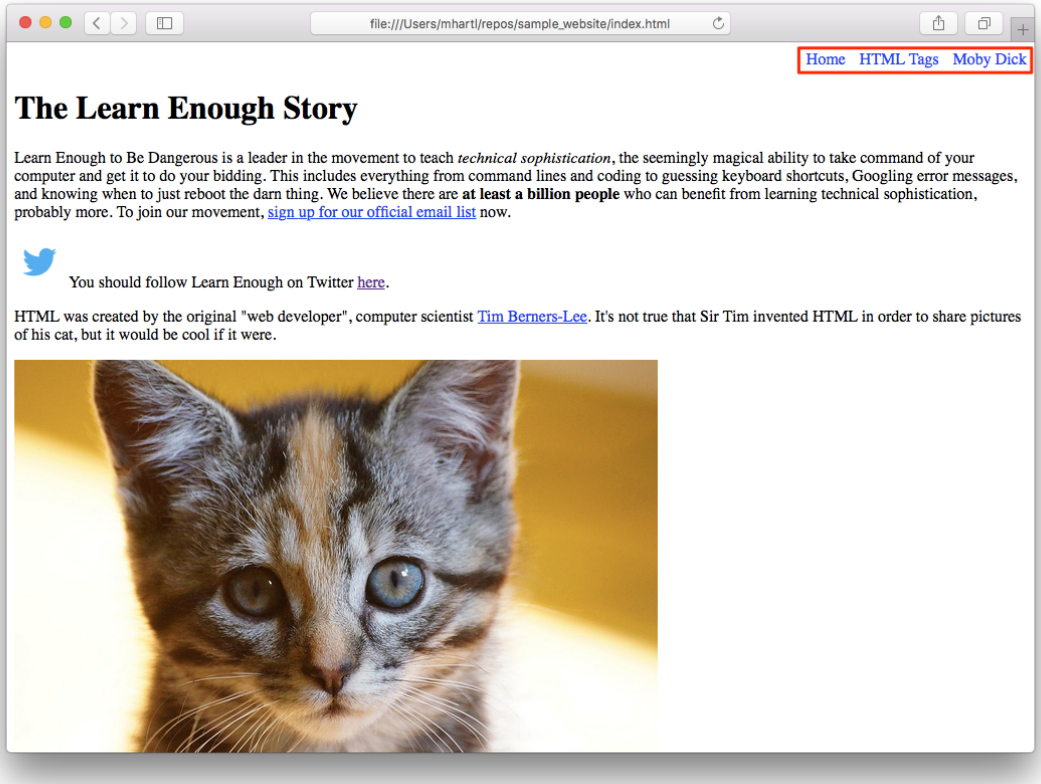


Figure 4.16: Styling the menu links.

4.7 A taste of CSS

In this section, we'll get a first taste of Cascading Style Sheets (CSS),[3] using the inline styles from **index.html** as our example (Figure 4.17).[4] We'll break the transition from inline styles to CSS into two steps: first, we'll move the inline styles into an *internal stylesheet* (Section 4.7.1); second, we'll move the internal style rules into an *external stylesheet* (Section 4.7.2). The result will be a self-contained and centralized location for our page's styling.

4.7.1 Internal stylesheets

We'll start by *refactoring* the inline styles into an internal stylesheet. Refactoring involves changing the form of code or markup without changing its function. In this case, we'll be moving the style rules on the elements in **index.html** into the **head** of our document, using a special **style** tag designed for exactly this purpose. In preparation for this, we'll add the opening and closing **style** tags with empty space for the style rules, as shown in Listing 4.14.

Listing 4.14: Adding an empty **style** tag (to be filled in in this section).
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="UTF-8">

    <style>

    </style>
  </head>
  .
  .
  .
```

[3] The words "style" and "sheet" are conventionally written as two words—Style Sheet—only in the context of spelling out the meaning of "CSS". Otherwise, they are generally combined into one word, "stylesheet".

[4] The appearance of your site may differ if you followed the exercises in Section 4.6.1.

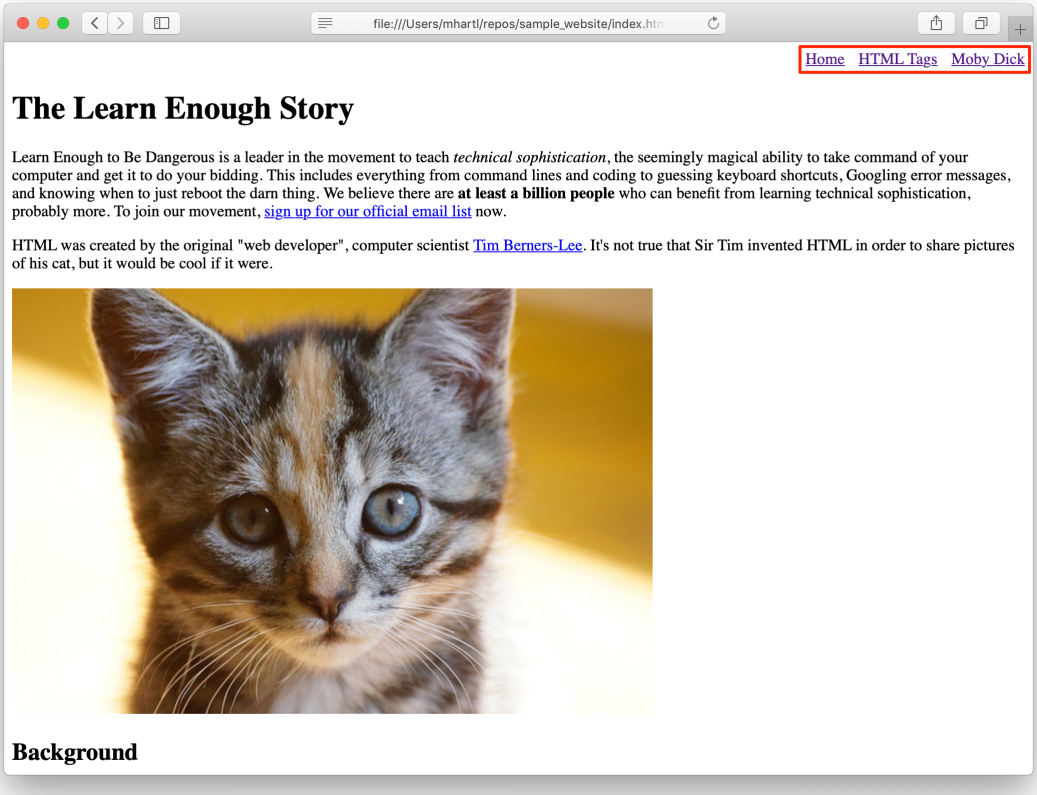


Figure 4.17: The current state of the Home page.

Our first step is to move the navigation styling developed in Section 4.6 (Listing 4.12). As shown in Listing 4.15, two of the lines contain identical margin styles (which control the spacing between links).

Listing 4.15: The current state of the menu styling code.
index.html

```
    .
    .
    .
    <div style="text-align: right;">
      <a href="index.html">Home</a>
      <a href="tags.html" style="margin: 0 0 0 10px;">HTML Tags</a>
      <a href="moby_dick.html" style="margin: 0 0 0 10px;">Moby Dick</a>
    </div>
    .
    .
    .
```

To eliminate this repetition, we'll start by replacing each of the inline styles with one of the most important ideas in web design, a CSS *class*, as shown in Listing 4.16.

Listing 4.16: Replacing the inline style with a CSS class.
index.html

```
    .
    .
    .
    <div style="text-align: right;">
      <a href="index.html">Home</a>
      <a href="tags.html" class="nav-spacing">HTML Tags</a>
      <a href="moby_dick.html" class="nav-spacing">Moby Dick</a>
    </div>
    .
    .
    .
```

A CSS class acts as a named label for the element, and allows us to simultaneously style all elements with the given class. The code for doing involves the name of the class with a leading dot (`.nav-spacing`) and a list of style rules (in this case, only one) inside curly braces:

```
.nav-spacing {
  margin: 0 0 0 10px;
}
```

In order to apply the styling to the page, this code should be placed inside the `style` tag, as shown in Listing 4.17.

Listing 4.17: Moving the inline margin rule into an internal stylesheet.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="UTF-8">

    <style>
      .nav-spacing {
        margin: 0 0 0 10px;
      }
    </style>
  </head>
  .
  .
  .
```

Note that the `margin` rule in Listing 4.17 is identical to the one in Listing 4.15; this is a general pattern for inline styles and CSS.

Next, we'll replace the inline style for the navigation `div` (Listing 4.18) with another CSS class (Listing 4.19) and a second CSS rule (Listing 4.20).

Listing 4.18: The inline style for the nav div.
index.html

```
    .
    .
    .
    <div style="text-align: right;">
      <a href="index.html">Home</a>
      <a href="tags.html" class="nav-spacing">HTML Tags</a>
      <a href="moby_dick.html" class="nav-spacing">Moby Dick</a>
    </div>
```

```
    .
    .
    .
```

Listing 4.19: Replacing the inline style with a class.
index.html

```
    .
    .
    .
    <div class="nav-menu">
      <a href="index.html">Home</a>
      <a href="tags.html" class="nav-spacing">HTML Tags</a>
      <a href="moby_dick.html" class="nav-spacing">Moby Dick</a>
    </div>
    .
    .
    .
```

Listing 4.20: Adding the nav menu rule to the internal stylesheet.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="UTF-8">

    <style>
      .nav-menu {
        text-align: right;
      }
      .nav-spacing {
        margin: 0 0 0 10px;
      }
    </style>
  </head>
  .
  .
  .
```

Here we've placed the `.nav-menu` rule above the `.nav-spacing` rule, but in this case the order doesn't matter. (The order of CSS rules often does matter, though; the details are covered in *Learn Enough CSS & Layout to Be Dangerous*.)

As a final step, we'll add the counterpart to a CSS class, called a CSS "identifier", or *id* (read as separate letters: "eye-dee"). Adding an id is certainly not necessary, and nor is it considered a good practice to use ids for styling (for reasons covered in *Learn Enough CSS & Layout to Be Dangerous*); we include it here mainly for purposes of illustration. But ids can be quite useful for *deep linking* (discussed in Section 4.7.3), and they are essential for many JavaScript applications (as covered in *Learn Enough JavaScript to Be Dangerous*).

In this case, we'll call the id **`main-nav`** (to indicate the main navigational element) and add it to the main navigational **`div`**, as shown in Listing 4.21. This has no effect on the page's appearance, but serves to illustrate the syntax. Note that, unlike classes, which can be used multiple times on a page, a given CSS id can be used only once.

Listing 4.21: Adding a CSS id.
index.html

```
    .
    .
    .
    <div id="main-nav" class="nav-menu">
      <a href="index.html">Home</a>
      <a href="tags.html" class="nav-spacing">HTML Tags</a>
      <a href="moby_dick.html" class="nav-spacing">Moby Dick</a>
    </div>
    .
    .
    .
```

At this point, if you've done everything right, the styles should be exactly the same as before, so upon refreshing the browser the appearance of the page should be also be the same (Figure 4.18).

4.7.2 External stylesheets

Now that we've refactored the inline styles into an internal stylesheet, putting them in an *external* stylesheet is easy. All we need to do is create a CSS file (which we'll call **`main.css`**), move the styles there, and then link to the file in the **`head`** of our document.

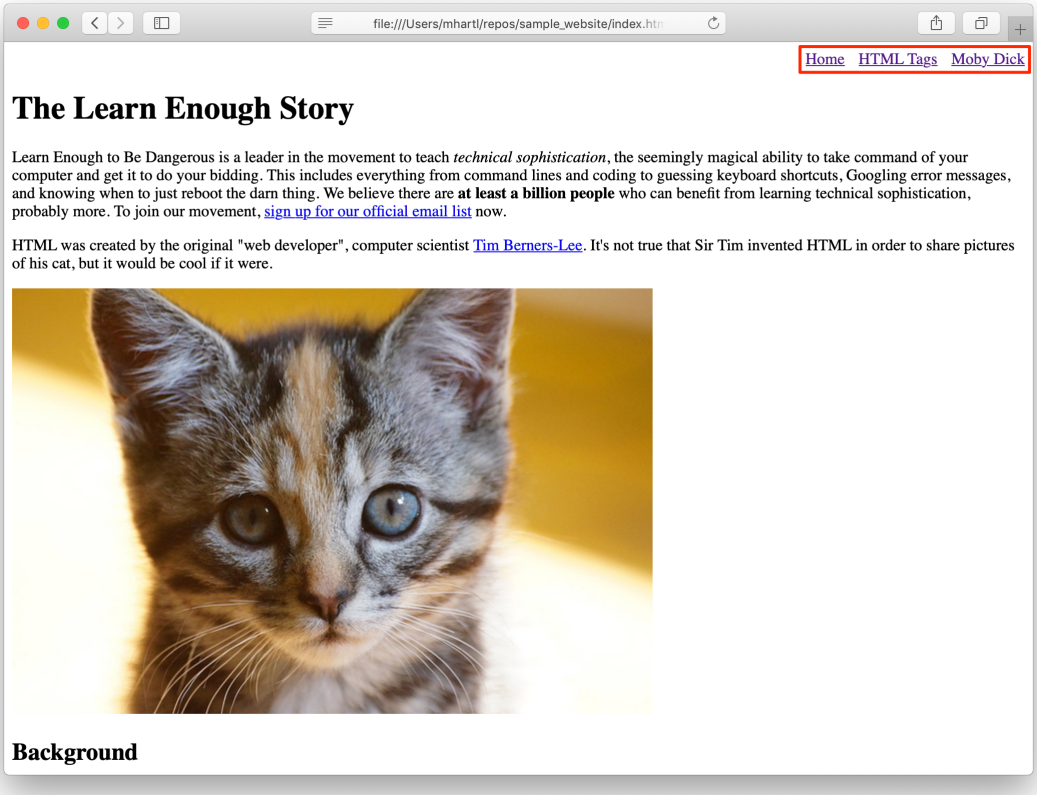


Figure 4.18: The menu links styled with CSS.

By convention, CSS files are usually located in a directory called **`css`** or **`stylesheets`**; we'll pick the former for brevity:

```
$ mkdir css
$ touch css/main.css
```

All that's needed now is to cut the CSS rules from **`index.html`** (Listing 4.22) and paste them into **`main.css`** (Listing 4.23).

Listing 4.22: Cutting the style rules.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="UTF-8">

    <style>

    </style>
  </head>
```

Listing 4.23: The style rules in an external stylesheet.
css/main.css

```
.nav-menu {
  text-align: right;
}
.nav-spacing {
  margin: 0 0 0 10px;
}
```

Finally, we'll delete the style tags and replace them with a **`link`** tag that includes the stylesheet into the page, as shown in Listing 4.24. (The **`rel`** in Listing 4.24 stands for "relationship"; in my experience, it has never been important to know this.)

Listing 4.24: Including the external stylesheet.
index.html

```
<!DOCTYPE html>
<html>
  <head>
    <title>Learn Enough to Be Dangerous</title>
    <meta charset="UTF-8">
    <link rel="stylesheet" href="css/main.css">
  </head>
  .
  .
  .
```

Putting styles in an external stylesheet is the technique that is generally used in real websites.

As before, moving the CSS into an external stylesheet should have changed nothing about the rules being applied, so the resulting page should look exactly the same (Figure 4.19).

The final step is to make a Git commit (taking care to run **`git add`** since we've added a new file and directory):

```
$ git add -A
$ git commit -m "Refactor inline styles into a stylesheet"
```

With that, we're done with our brief overview of CSS. This background is (barely) sufficient to skip to a beginning programming tutorial like *Learn Enough JavaScript to Be Dangerous*, although we recommend solidifying your knowledge by following at least some of *Learn Enough CSS & Layout to Be Dangerous*, which also covers how to use a professional-grade static site generator with a proper templating system to prevent repeated use of code.

4.7.3 Exercises

1. If you followed the exercises in Section 2.4.2, there is an image link to the Learn Enough Twitter account that has an inline style to remove underlining, as shown in Listing 2.15. Add a sensible class to this element

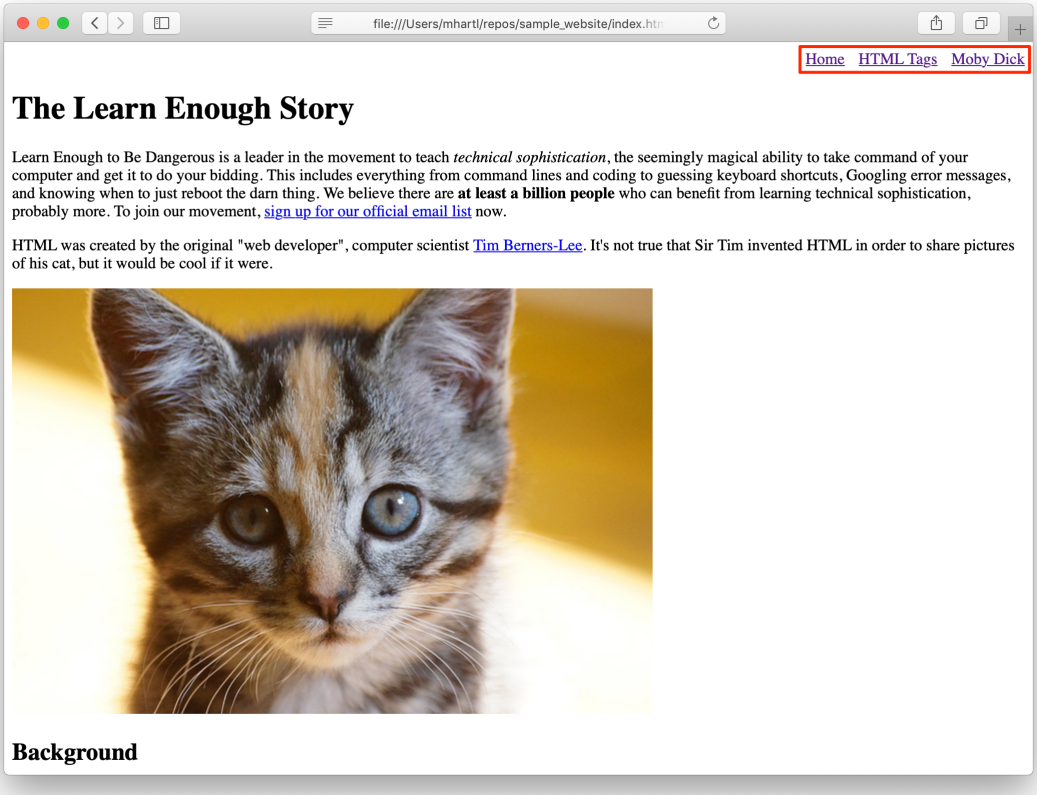


Figure 4.19: The menu links styled with an external stylesheet.

(such as `"image-link"`) and move the corresponding style rule into `main.css`. Does the page remain unchanged as required?

2. The updated menu shown in Listing 4.19 appears only on the Home page; to fix this, propagate the changes to the other two pages. (Appreciate once again how cumbersome this is, and how nice it would be to have a proper templating system.)

3. Move the style rules for the HTML Tags page and the Moby Dick page into `main.css` and confirm that the appearance of the pages stays the same.

4. One of the simplest and most useful applications of CSS ids is for creating links to arbitrary HTML elements using a convenient hash symbol (#) syntax. By adding the id `founders` to the "Founders" `h2` tag, show that you can visit that section of the page directly by pasting the URL sample_website/index.html#founders into your browser's address bar.

Listing 4.25: Adding a CSS id for deep linking.
index.html

```
    .
    .
    .
    <h2 id="founders">Founders</h2>
    .
    .
    .
```

4.8 Conclusion

Congratulations! You now know enough HTML to be *dangerous*. All that's left is to commit (in case you've made changes in Section 4.7.3) and deploy the final sample website:

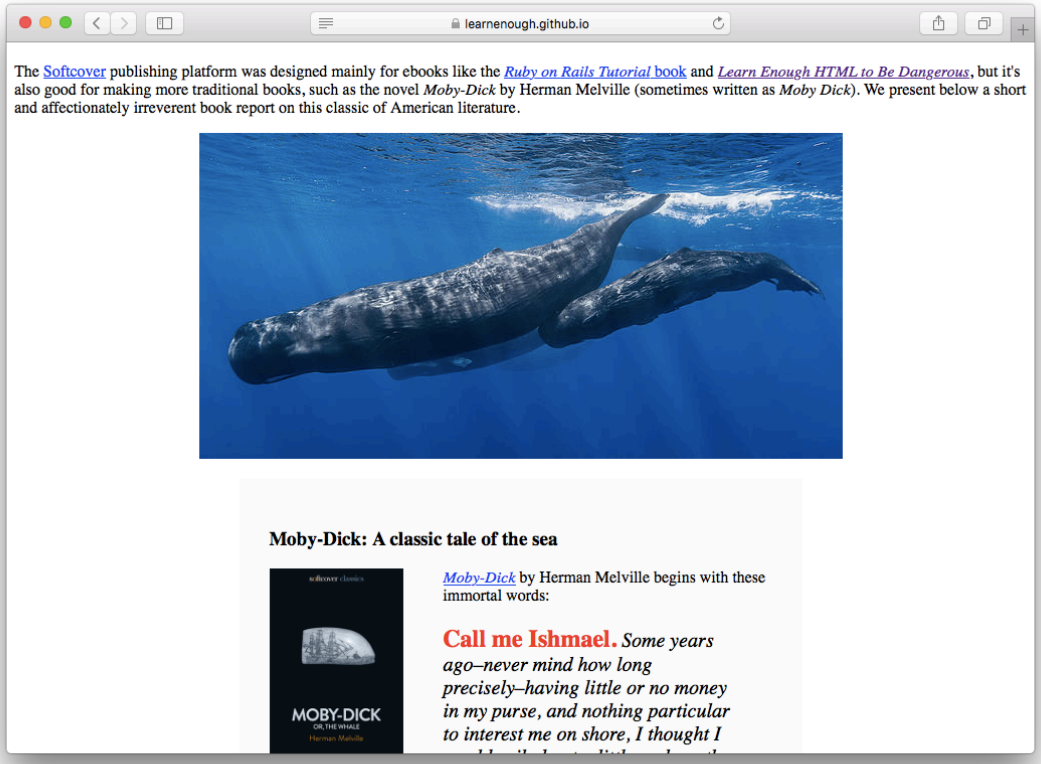


Figure 4.20: A production website.

```
$ git commit -am "Finish the sample website"
$ git push
```

The result is a full website running in a production environment (Figure 4.20). (To take things to an even more impressive level, see the free tutorial *Learn Enough Custom Domains to Be Dangerous* to learn how serve your GitHub Pages website from a custom domain.)

For reference, summary tables of all the block-level tags and inline tags appear in Table 4.2 and Table 4.3, respectively.

At this point, you've got a solid foundation in the basics of HTML. To learn how to make industrial-strength websites, though, you'll need to take the one

Tag	Name	Purpose
`h1`–`h6`	headings	include a heading (levels 1–6)
`p`	paragraph	include a paragraph of text
`table`	table	include a table
`tr`	table row	include a row of data
`th`	table header	make a table header
`td`	table data	include a table data cell
`div`	division	define block-level section in document
`header`	header	label the page header
`ol`	ordered list	list elements in numerical order
`ul`	unordered list	list elements whose order doesn't matter
`li`	list item	include a list item (ordered or unordered)
`blockquote`	block quotation	show formatted quotation
`br`	break	enter line break

Table 4.2: The block-level tags covered in this tutorial.

Tag	Name	Purpose	Example	Result
`em`	emphasized	make emphasized text	`<em>technical sophistication</em>`	*technical sophistication*
`strong`	strong	make strong text	`<strong>at least a billion people</strong>`	**at least a billion people**
`a`	anchor	make hyperlink	`<a href="#">Learn Enough</a>`	Learn Enough
`img`	image	include an image	`<img src="mhartl.jpg" alt="Michael Hartl">`	
`code`	code	format as source code	`<code>table</code>`	`table`
`span`	span	define inline section in document	`<span>Call me Ishmael.</span>`	Call me Ishmael.

Table 4.3: The inline tags covered in this tutorial.

additional step of reading *Learn Enough CSS & Layout to Be Dangerous*. And if you *really* want to take things as far as they can go, we recommend the full Learn Enough sequence:

1. **Developer Fundamentals**
    (a) *Learn Enough Command Line to Be Dangerous*
    (b) *Learn Enough Text Editor to Be Dangerous*
    (c) *Learn Enough Git to Be Dangerous*
2. **Web Basics**
    (a) *Learn Enough HTML to Be Dangerous*
    (b) *Learn Enough CSS & Layout to Be Dangerous*
    (c) *Learn Enough JavaScript to Be Dangerous*
3. **Application Development**
    (a) *Learn Enough Ruby to Be Dangerous*
    (b) *The Ruby on Rails Tutorial*
    (c) *Learn Enough Action Cable to Be Dangerous* (optional)

Good luck!

Learn Enough HTML to Be Dangerous. Copyright © 2016 by Michael Hartl and Lee Donahoe.