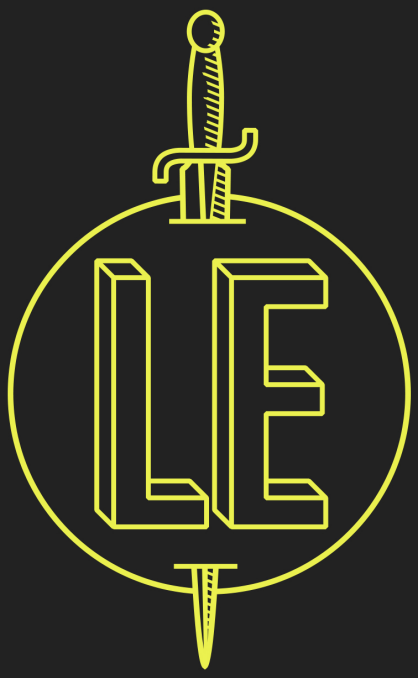

LEARN ENOUGH

TEXT EDITOR

TO BE DANGEROUS

FUNDAMENTALS **02**

TUTORIAL BY

MICHAEL HARTL

Learn Enough Text Editor to Be Dangerous

Michael Hartl

Contents

About the author

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.

Chapter 1

Introduction to text editors

Learn Enough Text Editor to Be Dangerous is designed to help you learn to use what is arguably the most important item in the aspiring computer magician's bag of tricks: a *text editor* (Figure 1.1). Learning how to use a text editor is an essential component of technical sophistication.

Unlike other text editor tutorials, which are typically tied to a specific editor, this tutorial is designed to introduce the entire *category* of application—a category many people don't even know exists. Moreover, editor-specific tutorials tend to be aimed at professional developers, and generally assume years of experience, but *Learn Enough Text Editor to Be Dangerous* doesn't even assume you know what a "text editor" is. Its only prerequisite is a basic understanding of the Unix command line, such as that provided by *Learn Enough Command Line to Be Dangerous*.[1]

Because *Learn Enough Text Editor to Be Dangerous* is part of a series of tutorials designed to teach the fundamentals of software development (with a particular focus on the prerequisites for learning web development with the Ruby on Rails Tutorial), it's ideally suited for anyone who wants to learn the skills necessary to work with developers or to become developers themselves. Finally, even if you already know how to use a text editor, following this tutorial (and doing the exercises) will help fill in any gaps in your knowledge, and you might even learn a few new tricks.

[1] This is required both because we'll be launching text editors from the command line and because some of the examples involve customizing and extending the *shell program* in which the command line runs.

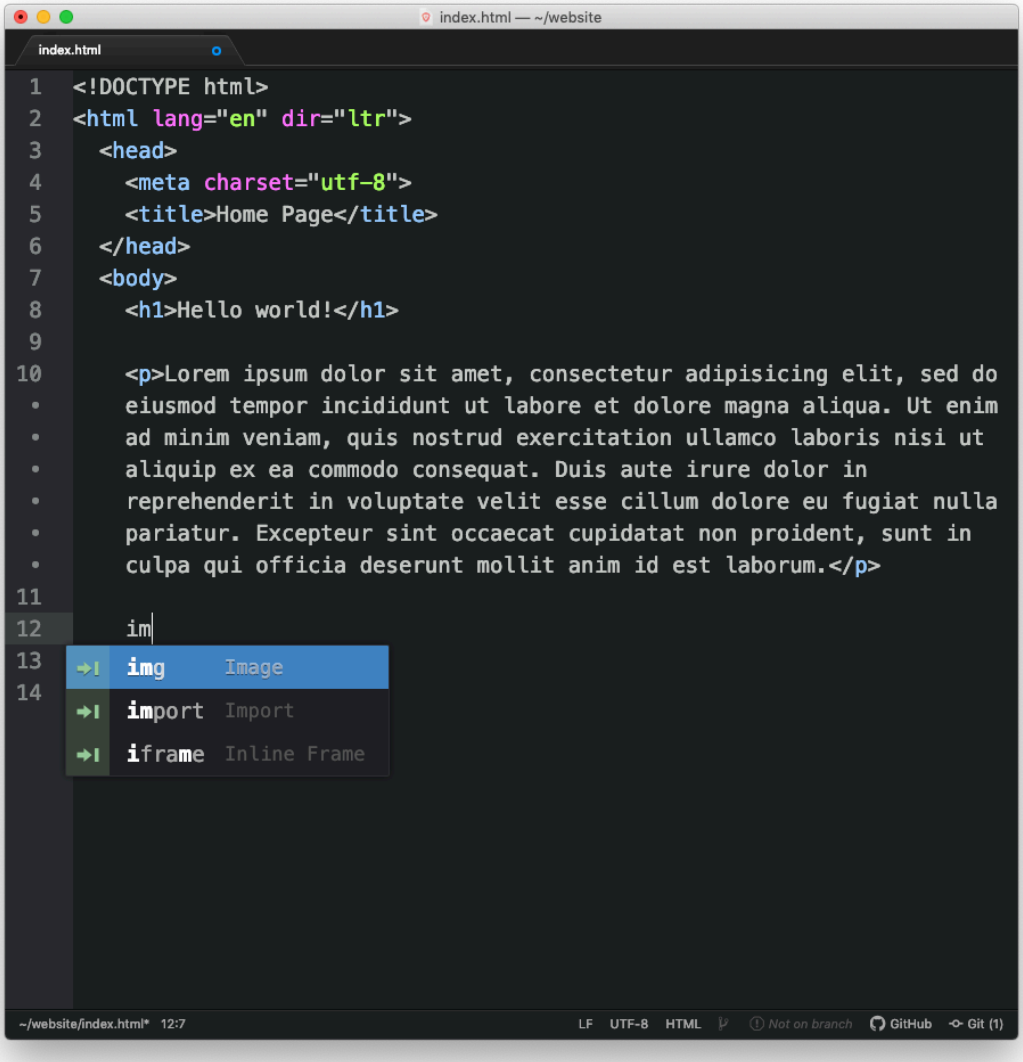

Figure 1.1: A text editor.

Unlike most programs used to produce written documents, such as word processors and email clients, a *text editor* is an application specifically designed to edit *plain text* (often called just "text" for short). Learning to use a text editor is important because text is ubiquitous in modern computing—it's used for code, markup, configuration files, and many other things.[2] (Indeed, I'm using plain text to write this very document.) Although it's surprisingly difficult to define exactly what "plain text" is, from a practical perspective it means that the text itself doesn't have any formatting, or at least none that matters. There's no notion of *emphasized* or **boldface** text, the font size and **`typeface`** don't matter, etc.—the only thing that does matter is the *content*. For example, although the previous sentence contains formatted output like *this*, its source is plain text, and appears as in Listing 1.1.[3]

Listing 1.1: The HTML source of a sentence in this tutorial.

```
There's no notion of <em>emphasized</em> or <strong>boldface</strong> text,
the <small>font size</small> and <code>typeface</code> don't matter,
etc.&mdash;the only thing that does matter is the <em>content</em>.
```

In Listing 1.1, the desired formatting options are indicated with special *tags* (such as the HTML emphasis tag **`<em>…</em>`**) rather than by changing the appearance of the text itself.[4] This is the main reason why the more familiar word processor programs such as Word aren't well-suited to editing plain text, and a different sort of tool is needed (Box 1.1).

Box 1.1. Word processors vs. text editors

[2] For more on the power of text, see the insightful post "always bet on text".

[3] Technically, the em dash '—' appears as a raw Unicode character rather than as `&mdash;`, but the latter is equivalent and renders better in isolation. For similar reasons, Listing 1.1 uses regular quotes in place of fancier 'curly' quotes. (On most browsers, setting Unicode to display properly requires a full HTML document with the proper headers. These sorts of considerations are covered in *Learn Enough HTML to Be Dangerous*.)

[4] It is up to the individual application to determine how to display the formatting. For example, HTML is designed to be rendered and displayed by web browsers like Chrome and Safari, which typically display emphasized text using *italics*.

Even if you've never used a text editor, the chances are good that you've used a similar tool, a *word processor*. There's a lot of overlap between the features of word processors and text editors. For example, they both allow you create documents, find and replace or cut/copy/paste text, and save the results. The main difference is that word processors are generally designed to produce documents following the principle of "What You See Is What You Get" (WYSIWYG, pronounced "WIZ-ee-wig"), so that effects such as *emphasis* or **boldface** are achieved directly in the application, instead of with plain-text markup like `<em>emphasis</em>` or `**boldface**`. For the most part, word processors also save their results in proprietary formats that sometimes go bad (as many who've tried opening old Word files have learned to their chagrin).

In contrast, text editors are designed to modify plain text, one of the most universal and durable formats. Text editors also differ from word processors in having features aimed at more technical users, including syntax highlighting for source code (Section 3.2.1), automatic indentation (Section 3.2.3), support for regular expressions (Section 3.4.3), and customization via packages and snippets (Section 3.5). A good text editor is thus an essential tool in every technical person's toolkit.

Building on the material developed in *Learn Enough Command Line to Be Dangerous*, *Learn Enough Text Editor to Be Dangerous* starts by covering the important *Vim* editor (Section 1.1), which can be run at the command line directly inside a terminal window. Vim will give us a chance to see our first examples of the most important functions of a text editor, but because Vim can be forbiddingly complex for a beginner, in this tutorial we will cover only the bare minimum necessary to make basic edits. The rest of the tutorial will expand on the themes developed in Chapter 1 by describing some of the many powerful features required in any programmer-grade text editor, with examples drawn principally from *Atom*, an open-source cross-platform editor, with concepts applicable to the closely related Sublime Text and Visual Studio Code editors and to Cloud9, a cloud IDE.[5]

[5]Some developers use an *integrated development environment*, or IDE, for their day-to-day programming, but

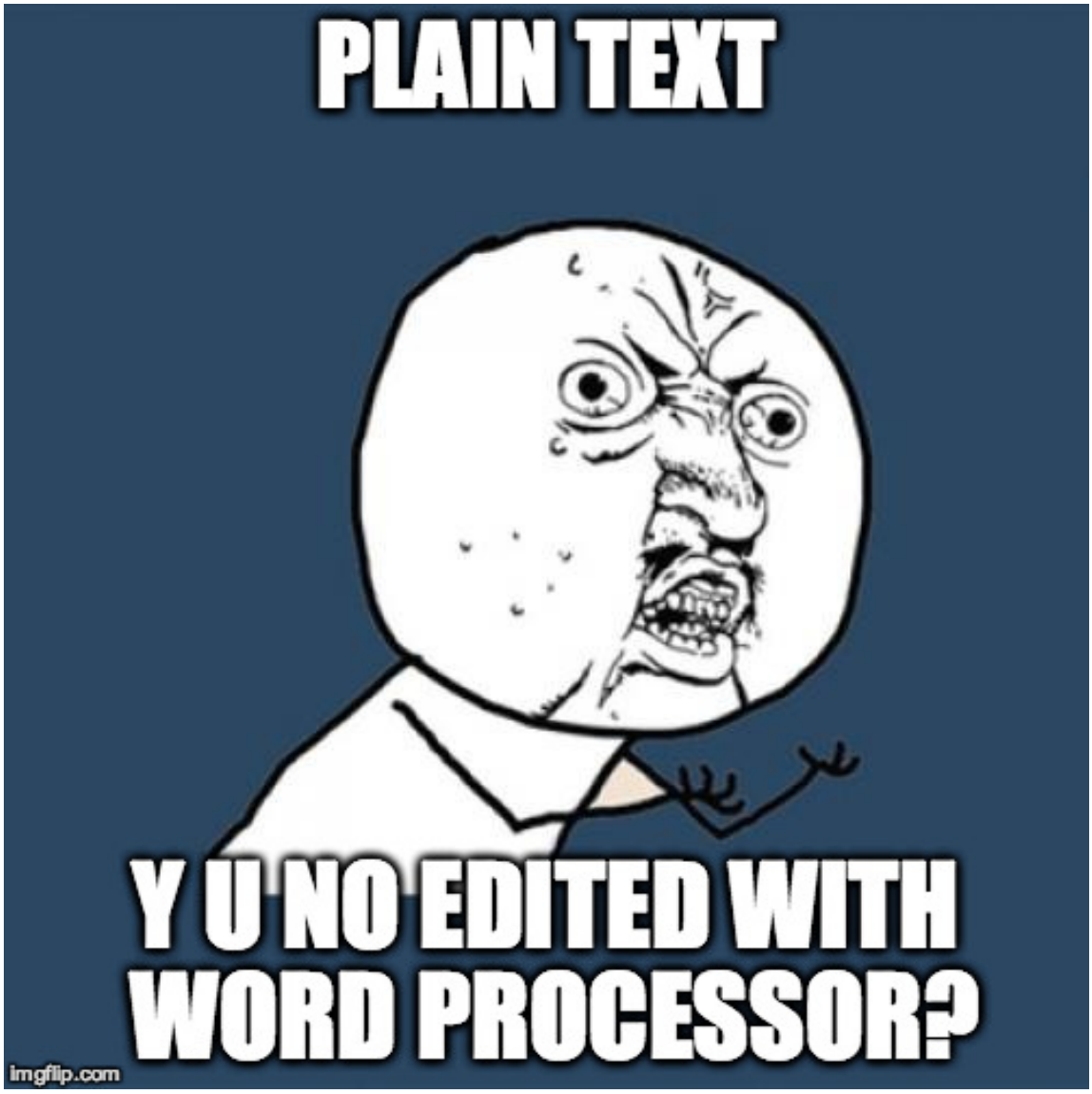

Figure 1.2: Why not edit plain text with Microsoft Word?

As with *Learn Enough Command Line to Be Dangerous*, this tutorial is part of the *Unix tradition*, which includes virtually every operating system you've ever heard of (macOS, iOS, Android, Linux, etc.) except Microsoft Windows. Although all the editors we'll discuss do run under Windows, using a non-Unix OS introduces friction into the process, so Windows users are encouraged to set up a Linux-compatible development environment by installing a *virtual machine* (Box 1.2) or following the Windows steps in *Learn Enough Dev Environment to Be Dangerous*. Another good option is to use a cloud IDE, which comes with a built-in command line and text editor; this option is also covered in *Learn Enough Dev Environment to Be Dangerous*.

Box 1.2. Running a virtual machine

One option for Windows users is to install a couple of free programs to run a *virtual machine* (VM) that allows Windows to host a version of the Linux operating system. It should be noted that Windows has improved Linux support in recent years, so I suggest trying the Windows steps in *Learn Enough Dev Environment to Be Dangerous* first to see if you can get those to work.

The steps to install a VM appear as follows:

1. Install the right version of VirtualBox for your system.
2. Download the Learn Enough Virtual Machine (large file).
3. Once the download is complete, double-click the resulting "OVA" file and follow the instructions to install the Virtual Machine (VM).
4. Double-click the VM itself and log in using the default user's password, which is "`foobar!`".

The result will be a Linux desktop environment (including a command-line terminal program and text editor) pre-configured for this tutorial (Figure 1.3).

every IDE includes an integrated text editor, so the lessons in this tutorial still apply.

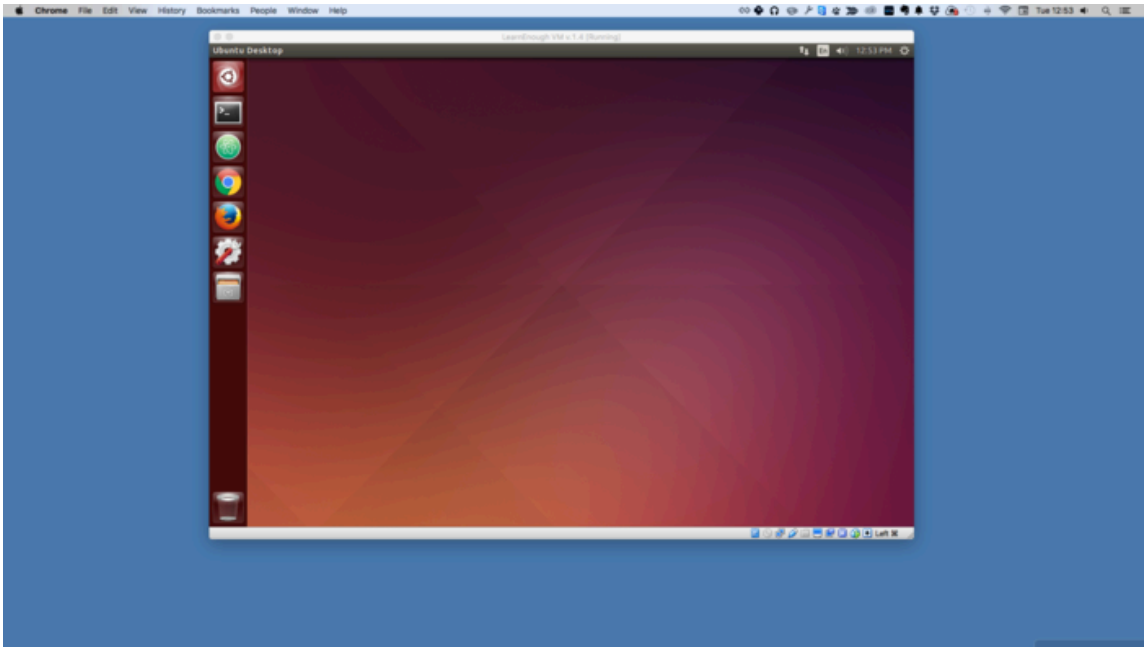

Figure 1.3: A Linux virtual machine running inside a host OS.

The focus throughout *Learn Enough Text Editor to Be Dangerous* is on general principles, so no matter which editor you end up using, you will have a good mental checklist of the kinds of tasks you should rely on your editor to perform. In addition, because the details vary by particular text editor and by system, this tutorial presents an ideal opportunity to continue developing your *technical sophistication* (Box 1.3). Finally, don't feel any pressure to master everything at once. You can be productive with even a small subset of what's included in this tutorial. Because technically sophisticated people use text editors practically every day, you'll keep learning new tricks in perpetuity.

Box 1.3. Technical sophistication

The phrase *technical sophistication*, mentioned before in *Learn Enough Command Line to Be Dangerous*, refers to the general ability to use computers and other technical things. This includes both existing knowledge (such as familiarity with

text editors and the Unix command line) and the ability to acquire *new* knowledge, as illustrated in "Tech Support Cheat Sheet" from xkcd. Unlike "hard skills" like coding and version control, this latter aspect of technical sophistication is a "soft skill"—difficult to teach directly, but essential to develop if you want to work with computer programmers or to become a programmer yourself.

In the context of text editors, technical sophistication includes things like reading menu items to figure out what they do, using the Help menu to discover new commands, learning keyboard shortcuts by reading menu items or Googling around, etc. It also involves a tolerance for ambiguity: technically sophisticated readers won't panic if a tutorial says to use ⌘Z to Undo something when it's actually ^Z on their system. They also won't panic if they see ⌘Z but don't know what ⌘ means, because they know they can skim ahead to find something like Table 2.1, or simply Google for it. Perhaps the most important aspect of technical sophistication is an *attitude*—a confidence and can-do spirit in the face of confusion that is well worth cultivating.

Throughout the rest of *Learn Enough Text Editor to Be Dangerous*, we'll refer back to this box whenever we encounter examples of issues that require a little technical sophistication to solve. With experience, you too will become one of the "computer people" from "Tech Support Cheat Sheet" who seem to have the magical ability to figure out technical things. (*Warning*: You might need a new shirt (Figure 1.4).)

1.1 Minimum Viable Vim

The vi (pronounced "vee-eye") editor dates back to the earliest days of Unix, and Vim (pronounced "vim") is an updated version that stands for "Vi IMproved". Vim is absolutely a full-strength text editor, and many developers use it for their daily editing needs, but the barrier to Vim mastery is high, and it requires substantial customization and technical sophistication (Box 1.3) to reach its full potential. Vim also has a large and often obscure set of commands,

Figure 1.4: A T-shirt for the technically sophisticated.

which rarely correspond to native keybindings (keyboard shortcuts), making Vim challenging to learn and remember. As a result, I generally recommend beginners learn a "modern" editor (Chapter 2) for everyday use. Nevertheless, I consider a minimal proficiency with Vim to be essential, simply because Vim is utterly ubiquitous—it's present on virtually every Unix-like system in the known universe, which means that if you ssh into some random server halfway 'round the globe, Vim will probably be there.

This chapter includes only Minimum Viable Vim—just enough to use Vim to do things like edit small configuration files or Git commits.[6] It's not even really enough to be *dangerous*. But it's worth noting that even mastering Minimum Viable Vim puts you in elite company—because Vim is so difficult, even a little Vim knowledge is the sort of thing that can impress your friends (or a job interviewer).

Note: If you're using macOS, you should follow the instructions in Box 1.4 at this time.

Box 1.4. Switching macOS to Bash

If you're using macOS, at this point you should make sure you're using the right shell program for this tutorial. The default shell as of macOS Catalina is *Z shell* (Zsh), but to get results consistent with this tutorial you should switch to the shell known as *Bash*.

The first step is to determine which shell your system is running, which you can do using the `echo` command:

```
$ echo $SHELL
/bin/bash
```

This prints out the `$SHELL` environment variable. If you see the result shown above, indicating that you're already using Bash, you're done and can proceed with the rest of the tutorial. (In rare cases, `$SHELL` may differ from the current

[6] See *Learn Enough Git to Be Dangerous* for more details.

shell, but the procedure below will still correctly change from one shell to another.) For more information, including how to switch to and use Z shell with this tutorial, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

The other possible result of `echo` is this:

```
$ echo $SHELL
/bin/zsh
```

If that's the result you get, you should use the `chsh` ("change shell") command as follows:

```
$ chsh -s /bin/bash
```

You'll almost certainly be prompted to type your system password at this point, which you should do. Then completely exit your shell program using Command-Q and relaunch it.

You can confirm that the change succeeded using `echo`:

```
$ echo $SHELL
/bin/bash
```

At this point, you will probably start seeing the following alert, which you should ignore:

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/
For more details, please visit https://support.apple.com/kb/

[~]$
```

Note that the procedure above is entirely reversible, so there is no need to be concerned about damaging your system. See "Using Z Shell on Macs with the Learn Enough Tutorials" for more information.

1.2 Starting Vim

Unlike most of the modern editors discussed starting in Chapter 2, Vim can be run directly inside a terminal window, and requires no graphical interface.[7] All you do is type **vim** at the prompt:

```
$ vim
```

Typical results of running the **vim** command appear in Listing 1.2 and Figure 1.5. In both cases, the tildes (~) are not characters in the file but rather represent lines that have yet to be defined.

Listing 1.2: A textual representation of a Vim window (message and versions may differ).

```
~
~
~
~
~
~                          VIM - Vi IMproved
~
~                             version 7.3
~                        by Bram Moolenaar et al.
~                 Vim is open source and freely distributable
~
~                        Help poor children in Uganda!
~                type  :help iccf<Enter>       for information
~
~                type  :q<Enter>               to exit
~                type  :help<Enter>  or  <F1>  for on-line help
~                type  :help version7<Enter>   for version info
```

If starting Vim is easy, learning to use it, at least at first, can be incredibly hard. This is mostly due to Vim being a *modal* editor, which is probably unlike anything you have used before (Box 1.5). Vim has two principal *modes*, known as *normal mode* and *insertion mode*. Normal mode is for doing things like moving around the file, deleting content, or finding and replacing text, whereas insertion mode is for inserting text.

[7] Vim thus dovetails nicely with a command-line tutorial like *Learn Enough Command Line to Be Dangerous*.

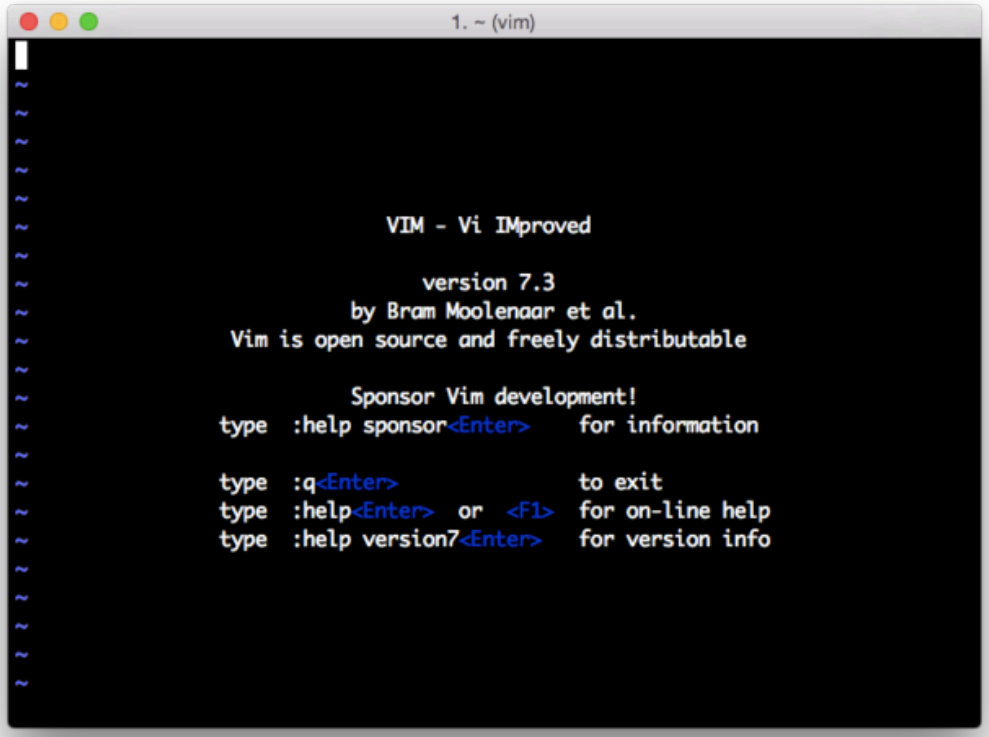

Figure 1.5: Vim running in a terminal window.

Box 1.5. Modal Vim

When I first started to learn programming in the Unix tradition (as opposed to my childhood experience with Microsoft DOS, BASIC, and Pascal), I distinctly recall being absolutely mortified at the unbelievably primitive editor I was expected to use. At the time, I was a first-year undergraduate at Harvard University, working in a research group at the Harvard-Smithsonian Center for Astrophysics. The tool I had been handed was vi. To say that it seemed like a downgrade from word processors is a gross understatement (Figure 1.2).

What shocked me most about vi was modal editing: unlike word processors, vi didn't let me just click in the window and start typing. Instead, there were a profusion of options (`i`, `a`, and `o` among them) for switching to *insertion mode*, and all it took was a few wrong keystrokes for all hell to break loose. Although the intervening years have seen a proliferation of more modern text editors, whose design is much more like the click-and-type interface I expected from my experience with word processors, the enduring popularity of vi (via Vim) means that learning the basics of modal editing is a valuable skill, even if it might at first seem ridiculously foreign.

Going back and forth between these two modes can cause a lot of confusion, especially since virtually all other programs that edit text (including word processors, email clients, and most text editors) have only insertion mode. Part of what makes Vim particularly confusing is that it *starts* in normal mode, which means that, if you try entering text immediately after starting Vim (as in Listing 1.2), the result will be chaos.

Because confusion is the likeliest result if you're not used to Vim's modal editing, we're going to start our study of Vim with the *Most Important Vim Command*™. One of my college friends, who was a huge partisan of vi's (and hence Vim's) historical rival Emacs (Box 1.6), claimed the Most Important Vim Command™ was the only one he ever wanted to learn. Here it is:

```
ESC:q!<return>
```

This command means "Press the Escape key, then type 'colon q exclamation-point', then press the return key." We'll learn in a moment what this does and why, but for now we'll start by practicing it a couple of times in the exercises.

Box 1.6. Holy wars: vi vs. Emacs

The Jargon File defines *holy wars* as follows:

> **holy wars**: n.
>
> [from *Usenet*, but may predate it; common] *flame war*s over *religious issues*. The paper by Danny Cohen that popularized the terms *big-endian* and *little-endian* in connection with the LSB-first/MSB-first controversy was entitled *On Holy Wars and a Plea for Peace*.
>
> Great holy wars of the past have included *ITS* vs. *Unix*, *Unix* vs. *VMS*, *BSD* Unix vs. System V, *C* vs. *Pascal*, *C* vs. FORTRAN, etc. In the year 2003, popular favorites of the day are KDE vs. GNOME, vim vs. elvis, Linux vs. [Free|Net|Open]BSD. Hardy perennials include *EMACS* vs. *vi*, my personal computer vs. everyone else's personal computer, ad nauseam. The characteristic that distinguishes holy wars from normal technical disputes is that in a holy war most of the participants spend their time trying to pass off personal value choices and cultural attachments as objective technical evaluations. This happens precisely because in a true holy war, the actual substantive differences between the sides are relatively minor. See also *theology*.

As noted in the Jargon File entry, one of the longest-raging holy wars is fought between proponents of vi and its arch-rival Emacs (sometimes written "EMACS"), both of which have played important roles in the Unix computing tradition. Both also retain much popular support, although my guess is that, with the popularity of

Figure 1.6: Watching a holy war play out can be entertaining.

Vim, vi has taken a decisive lead in recent years. Of course, this is just the sort of statement that serves to perpetuate a holy war—likely prompting Emacs partisans to, say, make claims about the superior power and customizability of their favorite editor.

If you wanted to start a *new* holy war, you might try something like, "Happily, the vi vs. Emacs holy war is now mostly a historical curiosity, as anyone who's anyone has switched to a modern editor like Atom or Sublime." It's going to be quite a show—better bring some popcorn (Figure 1.6).

1.2.1 Exercises

1. Start Vim in a terminal, then run the Most Important Vim Command™.
2. Restart Vim in a terminal. Before typing anything else, type the string "This is a Vim document." What happened? Confusing, right?
3. Use the Most Important Vim Command™ to recover from the previous exercise and return to the normal command-line prompt.

1.3 Editing small files

Now that we know the Most Important Vim Command™ (Section 1.2), we'll start learning how to use Vim for real by opening and editing a small file. In Section 1.2, we started by running **`vim`** by itself, but it's more common to use a filename as an argument. Let's navigate to the home directory of our system and then open the file indicated:

```
$ cd
$ vim .bash_profile
```

This file, which probably already exists on your system (but will be created automatically by Vim if necessary), is used to configure the *shell*, which is the program that supplies a command line. The default shell on most systems is *Bash*, a pseudo-acronym that stands for *Bourne Again SHell*.[8] As is common on Unix-based systems, the profile configuration file for Bash begins with a dot, indicating (as noted in *Learn Enough Command Line to Be Dangerous*) that the file is *hidden*, i.e., it doesn't show up by default when listing directory contents with **`ls`** (or even when viewing the directory using a graphical file browser).

We'll learn in Section 1.4 how to save changes to this file, but for now we're just going to add some dummy content so that we can practice moving around. In Section 1.2, we learned that Vim starts in normal mode, which means that we

[8] The first program in the sequence was the Bourne shell; in line with the Unix tradition of terrible puns, its successor is called the "Bourne again" (as in "born again") shell.

can change location, delete text, etc. Let's go into insertion mode to add some content. The first step is to press the **i** key to *insert* text. Then, type a few lines (separated by returns), as shown in Listing 1.3.[9] (There may be other existing content, which you should simply ignore.)

Listing 1.3: Adding some text after typing **i** to insert.
~/.bash_profile

```
lorem ipsum
dolor sit amet
foo bar baz
I've made this longer than usual because I haven't had time to make it shorter.
```

After entering the text in Listing 1.3, press **ESC** (the escape key) to switch from insertion mode back to normal mode.

Now that we have some text on a few lines, we can learn some commands for moving around small files. (We'll cover some commands for navigating large files in Section 1.6.) The easiest way to move around is to use the arrow keys—up, down, left, right—which is what I recommend.[10] Vim has literally jillions of ways of moving around, and if you decide to use Vim as your primary text editor I recommend learning them, but for our purposes the arrow keys are fine. The only two additional commands I feel are essential are the ones to move to the beginning and end of the line, which are **0** and **$**, respectively.[11]

1.3.1 Exercises

1. Use the arrow keys to navigate to Line 4 in the file from Listing 1.3.
2. Use the arrow keys to go to the end and then the beginning of Line 4. Cumbersome, eh?

[9] Recall from *Learn Enough Command Line to Be Dangerous* that a tilde ~ is used to indicate the home directory, so **~/.bash_profile** in the Listing 1.3 caption refers to the Bash profile file in the home directory.

[10] Vim purists will tell you that there's a better way, namely, to use the **h**, **j**, **k**, **l** to move around, but it takes a lot of practice for this to become intuitive, and it's certainly not necessary to be *dangerous*.

[11] These are not native keybindings (on macOS they would be Command–left arrow and Command–right arrow), which as noted in the introduction to Chapter 1 makes it harder to learn them. This is just one of many reasons I don't generally recommend beginners use Vim as their primary editor.

3. Go to the beginning of Line 4 by using the command mentioned in the text.

4. Go to the end of Line 4 using the command mentioned in the text.

1.4 Saving and quitting files

Having learned a little about moving around and inserting text, now we're going to learn how to save a file. Our specific example will involve making a useful new Bash command, but first we have to deal with the current state of the Bash profile file. The text we added in Listing 1.3 is gibberish (at least from Bash's perspective), so what we'd like to do is quit the file without saving any changes. For historical reasons, some Vim commands (especially those having to do with file manipulation) start with a colon `:`, and the normal way to quit a file is with `:q<return>`, but that only works when there are no changes to save. In the present case, we get the error message "No write since last change (add ! to override)", as shown in Figure 1.7.

Following the message's advice, we can type `:q!<return>` to force Vim to quit without saving any changes (Figure 1.8), which returns us to the command line.

You may have noticed that we're now in a position to understand the Most Important Vim Command™ introduced in Section 1.2: no matter what terrible things you might have done to a file, as long as you type `ESC` (to get out of insertion mode if necessary)[12] followed by `:q!<return>` (to force-quit) you are guaranteed not to do any damage.

Of course, Vim is only really useful if we can save our edits, so let's add some useful text and then write out the result. As in Section 1.3, we'll work on the `.bash_profile` file, and the edit we'll make will add an *alias* to our shell. In a computing context, an alias is simply a synonym for a command or set of commands. The main use for Bash aliases is defining shorter commands for commonly used combinations.[13]

[12] Hitting `ESC` while in normal mode does no harm, so it's a good idea to include this step in any case.

[13] To learn how to write aliases using Zsh, see "Using Z Shell on Macs with the Learn Enough Tutorials". TL;DR:

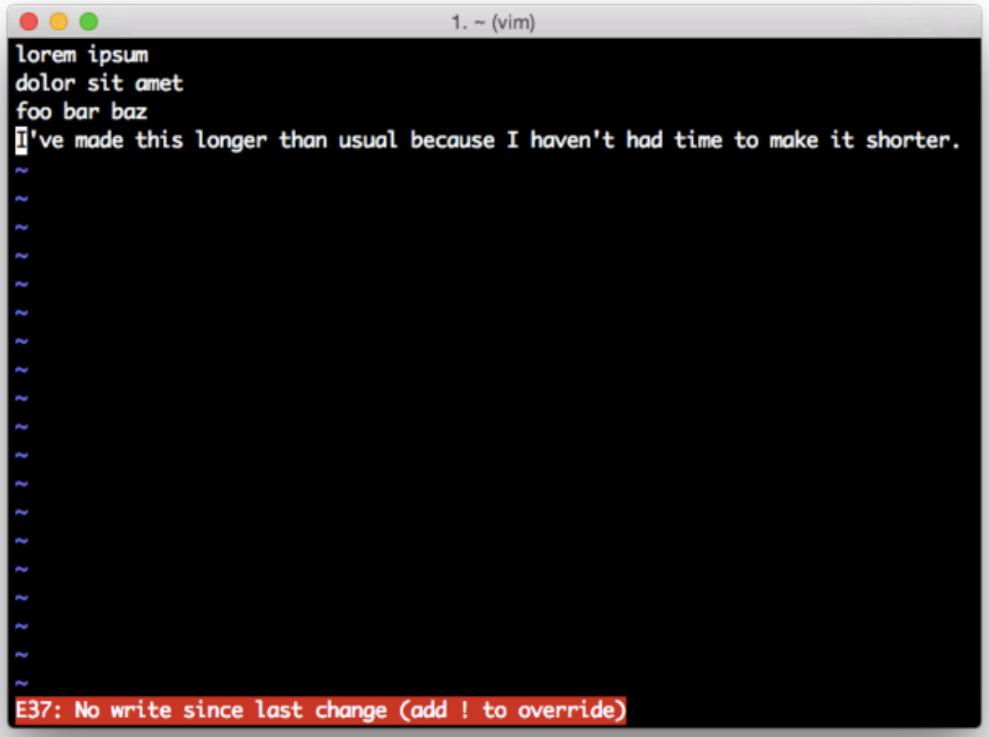

Figure 1.7: Trying to quit a file with unsaved changes.

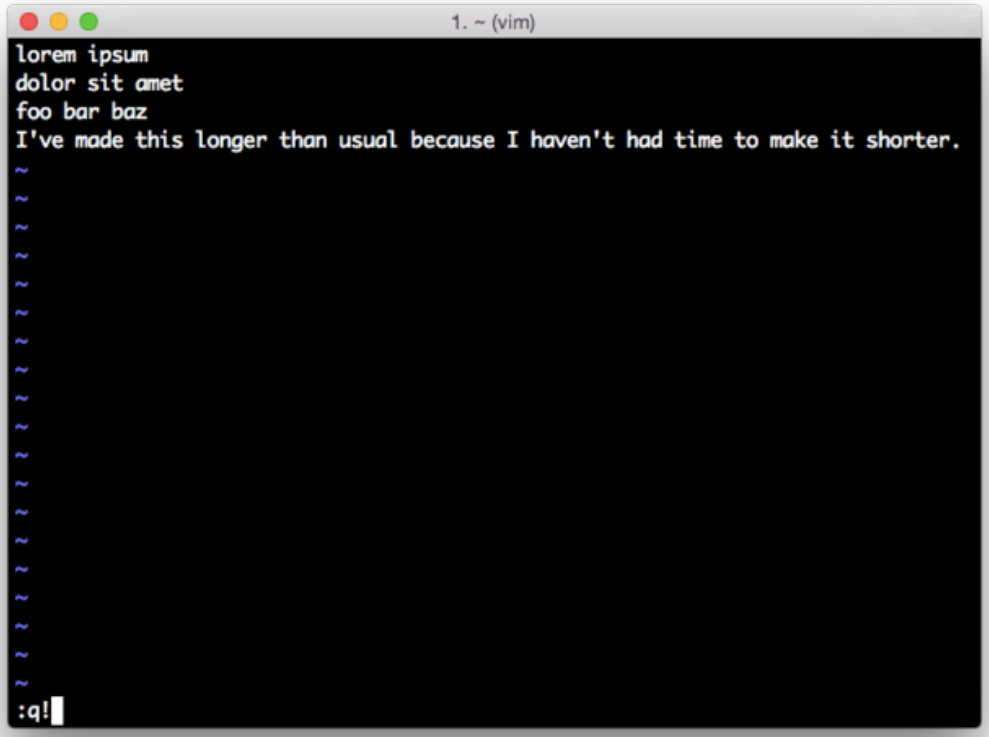

Figure 1.8: Forcing Vim to quit.

In this case, we'll define the command `lr` (short for "list reverse") as an alias for `ls -hartl`, which is the command to list files and directories using **h**uman-readable values for the sizes (e.g., 29K instead of 29592 for a 29-kilobyte file), including **a**ll of them (even hidden ones), ordered by **r**everse **t**ime, **l**ong form. This command, which as you may recognize from an exercise in *Learn Enough Command Line to Be Dangerous*, is useful for seeing which files and directories have recently changed (as well as being, for obvious reasons, one of my personal favorites). After defining the alias, we'll be able to replace the more verbose

```
$ ls -hartl
```

with the pithier

```
$ lr
```

The steps appear as follows:

1. Press `i` to enter insertion mode.
2. Enter the contents shown in Listing 1.4. (On some systems, the `.bash_profile` file may include some pre-existing content, which you can simply leave in place.)
3. Press `ESC` to exit insertion mode.
4. Write the file using `:w<return>`.
5. Quit Vim by typing `:q<return>`.

Note: If you make any mistakes, you can type `ESC` followed by `u` to *undo* any of the previous steps. (Most programs use Command-Z or Ctrl-Z to undo things, yet another example of the non-native keybindings used by Vim. In contrast, the editors discussed starting in Chapter 2 all support native keybindings.)

The syntax is identical; the only difference is that you edit a file called `.zshrc` instead of `.bash_profile`. (Indeed, in Bash you can actually edit a file called `.bashrc` instead, which makes the parallel even clearer.)

Listing 1.4: Defining a Bash alias.
~/.bash_profile

```
alias lr='ls -hartl'
```

After adding the `lr` alias to `.bash_profile`, writing the file, and quitting, you may be surprised to find that the command doesn't yet work:

```
$ lr
-bash: lr: command not found
```

This is because we need to tell the shell about the updated Bash profile file by "sourcing" it using the `source` command, as shown in Listing 1.5.[14]

Listing 1.5: Activating the alias by sourcing the Bash profile.

```
$ source .bash_profile
```

With that, the `lr` command should work as advertised:

```
$ lr
.
.
.
drwx------+  15 mhartl  staff   510B Sep  4 18:58 Desktop
-rw-------    1 mhartl  staff    13K Sep  4 19:13 .viminfo
-rw-r--r--    1 mhartl  staff    46B Sep  4 19:14 .bash_profile
drwxr-xr-x+ 117 mhartl  staff   3.9K Sep  4 19:14 .
```

By the way, the `.bash_profile` file is sourced automatically when we open a new terminal tab or window, so explicit sourcing is necessary only when we want a change to be reflected in the *current* terminal.

[14] Incidentally, a bare "dot" is a shorthand for `source`, so in fact you can type `. .bash_profile` to obtain the same result. (This usage is unrelated to the use of a dot to refer to the current directory.)

1.4.1 Exercises

1. Define an alias `g` for the commonly used *case-insensitive grep* `grep -i`. What happens if, after making your changes and hitting `ESC`, you issue the command `:wq` instead of `:w` and `:q` separately?

2. You may recall the `curl` command from *Learn Enough Command Line to Be Dangerous*, which lets us interact with URLs via the command line. Define `get` as an alias for `curl -OL`, which is the command to download a file to the local disk (while following any redirects encountered along the way).

3. Use the alias from the previous exercise to execute the command shown in Listing 1.6, which downloads a longer text file for use in Section 1.6.

Listing 1.6: Downloading a longer text file for use in a future section.

```
$ get cdn.learnenough.com/sonnets.txt
```

1.5 Deleting content

As with every category of text manipulation, Vim has an enormous number of commands for deleting content, but in this section we're just going to cover the absolute minimum. We'll start with deleting single characters, which we can do in normal mode using the `x` command:

1. Open `.bash_profile` and insert the misspelled word `aliaes`

2. Get back to normal mode by pressing `ESC`

3. Move the cursor over the `e` in `aliaes` (Figure 1.9) and press `x`

There are lots of fancy ways to delete text, but by repeatedly pressing `x` it's easy (if a bit cumbersome) to delete entire words or even entire lines. On the other hand, deleting lines is enough of a special case to merit inclusion. Let's

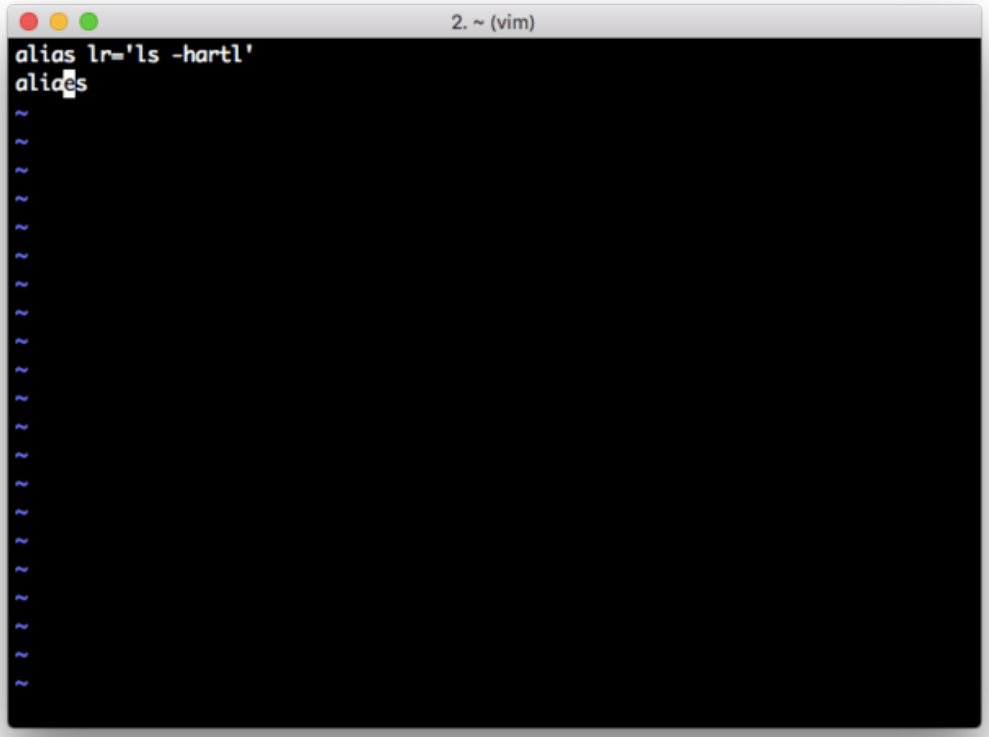

Figure 1.9: Preparing to delete a letter using `x`.

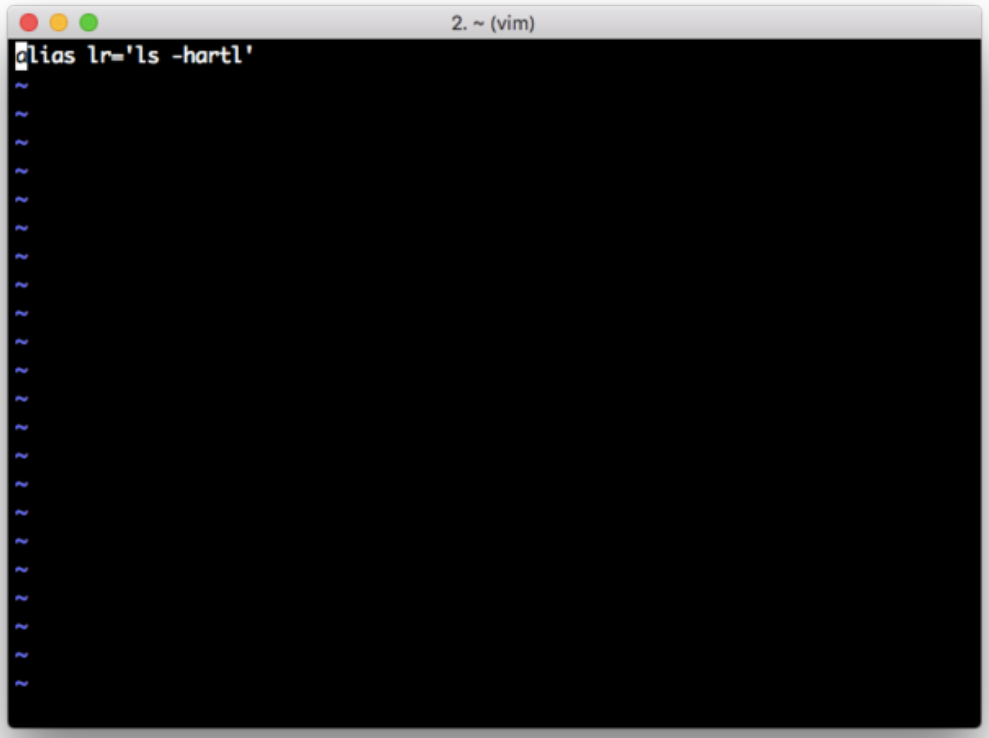

Figure 1.10: The result of deleting a line with **dd**.

get rid of the extra **alias** we added by pressing **dd** to delete the line. Voilà! It should be gone (Figure 1.10). To get it back, you can press **p** to "put" the line, which also allows you to simulate copying and pasting one line at a time. (Again, this is a minimal subset of Vim; if you decide to get good at it, you'll learn lots of better ways to do things.)

1.5.1 Exercises

1. Using Vim, open a new file called **foo.txt**.

Figure 1.11: This animal's spot-changing abilities are frequently questioned.

2. Insert the string "A leopard can't change it's spots." (Figure 1.11).[15]

3. Using the **x** key, delete the character necessary to correct the mistake in the line you just entered. (If you can't find the error, refer to Table 1.1.)

4. Use **dd** to delete the line, then use **p** to paste it repeatedly into the document.

5. Save the document and quit using a single command. *Hint*: See the first exercise in Section 1.4.1.

[15] Image retrieved from https://www.flickr.com/photos/tambako/4703806355 on 2015-11-12 and used unaltered under the terms of the Creative Commons Attribution-NoDerivs 2.0 Generic license.

1.6 Editing large files

The final skills needed for your Minimum Viable Vim involve navigating large files. If you didn't download **`sonnets.txt`** in the exercises from Section 1.4, you should so do now (Listing 1.7).[16]

Listing 1.7: Downloading Shakespeare's *Sonnets*.

```
$ curl -OL https://cdn.learnenough.com/sonnets.txt
```

The resulting file contains the full text of Shakespeare's *Sonnets*, which is 2620 lines, 17670 words, and 95635 characters long, which we can verify using the word count command **`wc`** covered in *Learn Enough Command Line to Be Dangerous*:

```
$ wc sonnets.txt
    2620   17670   95635 sonnets.txt
```

On many systems, Vim shows some of the same basic stats upon opening the file:

```
$ vim sonnets.txt
```

The result on my system appears in Figure 1.12. Because of its length, this file is far too long to navigate conveniently by hand.

As before, there are lots of commands for moving around, but I find that the most useful ones involve moving a screen at a time, moving to the beginning or end, or searching. The commands to move one screen at a time are Ctrl-F (Forward) and Ctrl-B (Backward). To move to the end of the file, we can use **`G`**, and to move to the beginning we can use **`1G`**. Finally, perhaps the most powerful navigation command is *search*, which involves typing slash **`/`** followed by the

[16] If you completed the exercises in Section 2.7.1, you can use your own custom **`get`** alias in place of **`curl -OL`**.

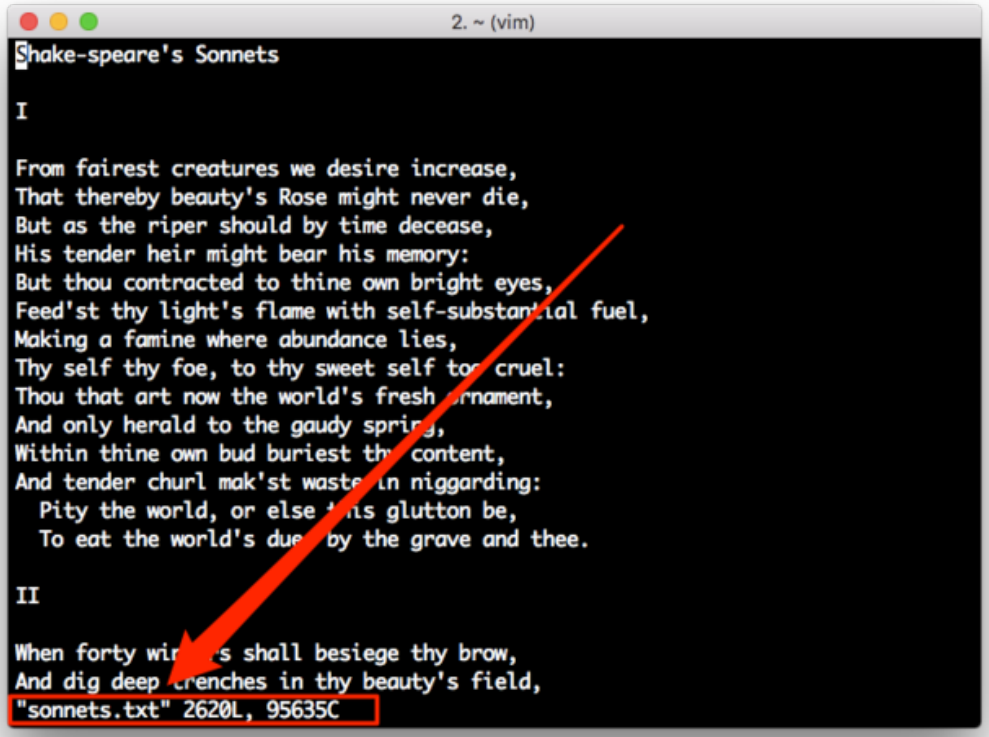

Figure 1.12: Some file stats displayed upon starting Vim.

string you want to find. The trick is to type `/<string>` followed by return, and then press `n` to go to the next match (if any).

This might all sound a little familiar, because it's the same as the interface to the `less` program covered in *Learn Enough Command Line to Be Dangerous*.[17] This is one of the advantages of learning basic Unix commands: many of the patterns recur in many different contexts.

1.6.1 Exercises

1. With `sonnets.txt` open in Vim, move down three screens and then back up three screens.

2. Go to the end of the file. What is the last line of the final sonnet?

3. Navigate back to the top to change the old-style name "Shake-speare" on Line 1 of `sonnets.txt` to the more modern "Shakespeare", and save the result.

4. Use Vim's search feature to discover which sonnet contains references to Cupid, the Roman god of love.

5. Confirm that `18G` goes to the final line of the first sonnet. What do you suppose that command does? *Hint*: Recall that `1G` goes to the beginning of the file, i.e., Line 1.

1.7 Summary

Important commands from this section are summarized in Table 1.1. If you're interested in learning more about Vim, dropping "learn vim" into a search engine is a good bet. The Interactive Vim tutorial is especially recommended.

[17] Depending on your system, there may be minor differences between the Vim and `less` interfaces. For example, on my system the slash operator is case-sensitive when used with `less` and case-insensitive when used with Vim. As usual, use your technical sophistication (Box 1.3) to resolve any discrepancies.

Command	Description
`ESC:q!<return>`	The Most Important Vim Command™
`i`	Exit normal mode, enter insertion mode
`ESC`	Exit insertion mode, enter normal mode
Arrow keys	Move around
`0`	Go to beginning of line
`$`	Go to end of line
`:w<return>`	Save (write) a file
`:q<return>`	Quit a file (must be saved)
`:wq<return>`	Write and quit a file
`:q!<return>`	Force-quit a file, discarding any changes
`u`	Undo
`x`	Delete the character under the cursor
`dd`	Delete a line
`p`	Put (paste) deleted text
`it's spots`	No, you mean `its spots`
`Ctrl-F`	Go forward one screen
`Ctrl-B`	Go backward one screen
`G`	Go to last line
`1G`	Go to first line
`/<string>`	Search for `<string>`

Table 1.1: Important Vim commands from Chapter 1.

1.7.1 Exercises

1. Open `sonnets.txt`.
2. Go to the last line.
3. Go to the end of the last line.
4. Make a new line that says "That's all, folks! Bard out. <drops mic>". Make sure to move the cursor one space to the right so you don't drag the final period along.
5. Write out the file.
6. Undo your changes.
7. Write out and quit the file.
8. Reopen the file and type `2620dd`.
9. Realize that you just deleted the entire file contents, and apply the Most Important Vim Command™ to ensure that no damage is done.

Chapter 2

Modern text editors

Having learned the minimal basics of text editing with Vim, we're now in a good position to appreciate the preferred "modern" text editors mentioned at the beginning of this tutorial. These editors include cross-platform native editors such as Sublime Text, Visual Studio Code, and Atom, and editors in the cloud like Cloud9. Modern editors are distinguished by their combination of power and ease-of-use: you can do fancy things like global search-and-replace, but (unlike Vim) they let you just click in a window and start typing. In addition, many of them (including Atom and Sublime) include an option to run in Vim compatibility mode, so even if you end up loving Vim you can still use a modern editor without having to give Vim up entirely.

Throughout the rest of this tutorial, we'll explore the capabilities of modern text editors. We'll end up covering all of the topics encountered in our discussion of Vim (Chapter 1), as well as many more advanced subjects, including opening files, moving around, selecting text, cut/copy/paste, deleting and undoing, saving, and finding/replacing—all of which are important for day-to-day editing. We'll also discuss both menu items and keyboard shortcuts, which help make your text editing faster and more efficient.

For future reference, Table 2.1 shows the symbols for the various keys on a typical Macintosh keyboard. Apply your technical sophistication (Box 1.3) if your keyboard differs.

Key	Symbol
Command	⌘
Control	^
Shift	⇧
Option	⌥
Up, down, left, right	↑ ↓ ← →
Enter/Return	↩
Tab	⇥
Delete	⌫

Table 2.1: Miscellaneous keyboard symbols.

2.1 Choosing a text editor

While cloud IDEs have many advantages, every aspiring computer magician should learn at least one native editor (i.e., an editor that runs on your desktop operating system). While there are many editors to choose from, probably the most promising modern text editors in use today Sublime Text (sometimes just called "Sublime"), Visual Studio Code (VScode), and Atom.[1] Each has its own advantages and disadvantages.

2.1.1 Sublime Text

- Advantages
    1. Powerful, hackable, and easy to use
    2. Can be used for free in "evaluation mode"
    3. Fast and robust, even when editing huge files/projects
    4. Works cross-platform (Windows, macOS, Linux)
    5. Backed by a profitable company that has a good track record of support and development
- Disadvantages

[1] Other options include TextMate, NotePad++, jEdit, and BBEdit.

1. Free in neither the speech nor beer senses
2. Has a mildly annoying popup that goes away only if you buy a license
3. Costs $70 as of this writing
4. Setting up command-line tools takes some fiddling

2.1.2 Visual Studio Code (VSCode)

- Advantages
  1. Powerful, with lots of packages
  2. Free to use
  3. Fast and robust, even when editing huge files/projects
  4. Works cross-platform (Windows, macOS, Linux)
  5. Backed by Microsoft
- Disadvantages
  1. Not open-source
  2. Backed by Microsoft

2.1.3 Atom

- Advantages
  1. Powerful, hackable, and easy to use
  2. Free both as in speech and as in beer (i.e., it both costs nothing and is open-source software)
  3. Works cross-platform (Windows, macOS, Linux)
  4. Easy to set up command-line tools
  5. Backed by collaboration powerhouse GitHub

- Disadvantages

    1. Reports of being slower in some cases than Sublime or VSCode

It's hard to go wrong with any of these choices. My main day-to-day editor as of this writing is Sublime Text, and I've heard great things about VSCode, but because of its simplicity and being 100% free I think Atom is probably the best choice for new users. The good news is that the skills in the sections that follow are near-universal; if you learn Atom but decide to switch to Sublime Text or VSCode (or even a cloud IDE) for your daily editing, most of the core ideas will translate easily.

2.1.4 Exercises

Install and configure a text editor on your system as follows:

1. Download and install either Sublime Text, Visual Studio Code, or Atom. *Note*: Atom comes pre-installed on the Linux virtual machine described in Box 1.2, so if you're using the VM you can skip this step.

2. If using Sublime Text, set up the `subl` command by Googling for "sublime text command line" and following the instructions for your system. Apply your technical sophistication (Box 1.3) if you get stuck. You might also find it helpful to skip ahead to Section 3.3 to learn about how to configure your system's *path*.

3. If using VSCode, set up the `code` command by Googling for "visual studio code command line" and following the instructions for your system.

4. If using Atom, go to Atom > Install shell commands to enable the `atom` command at the command line (Figure 2.1).

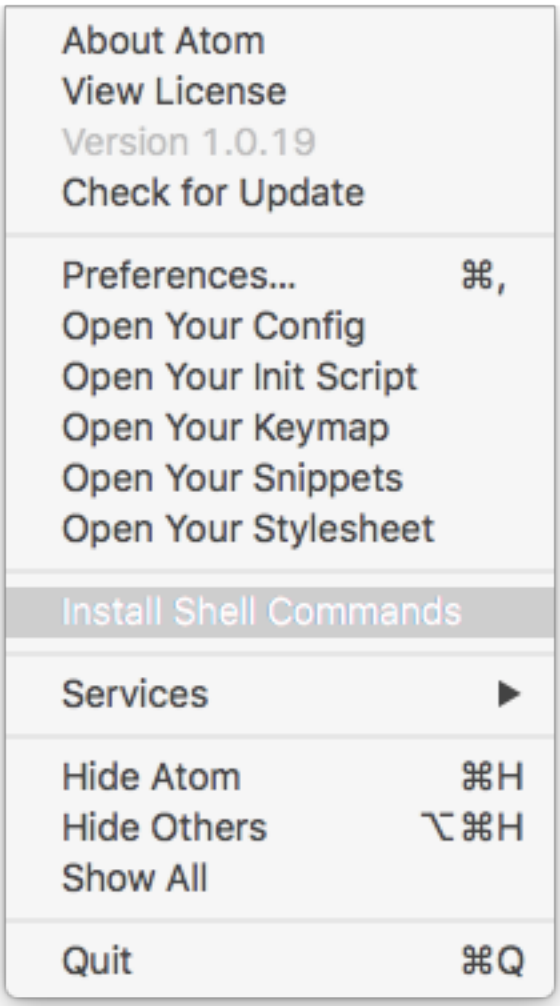

Figure 2.1: Installing Atom's shell commands.

2.2 Opening

To open files, we're going to use the command configured in Section 2.1.4 to launch the editor and open the file at the same time (a method we used with **vim** in Section 1.3). In Section 3.4, we'll cover a second method (called "fuzzy opening") that's useful when editing a project with multiple files. I'll assume you're using the **atom** command, but if you're using Sublime you should make the appropriate substitution (**subl** in place of **atom**).

Let's get started by downloading a sample file, **README.md**, from the web. As in Section 2.7.1 and Section 1.6, we'll use **curl** command to download the file at the command line:

```
$ curl -OL https://cdn.learnenough.com/README.md
```

As hinted at by the **.md** extension, the downloaded file is written in Markdown, a human-readable markup language designed to be easy to convert to HTML, the language of the World Wide Web.

After downloading **README.md**, we can open it at the command line as follows:

```
$ atom README.md
```

(If this doesn't work, be sure you've installed the Atom shell commands as shown in Figure 2.1.) The result of opening **README.md** in Atom should look something like Figure 2.2 or Figure 2.3. (If this is your first time opening Atom, it's also possible you'll see a one-time greeting screen. As usual, apply Box 1.3.) Figure 2.2 shows the usual default, which is for "word wrap" to be off; because Markdown files are typically written using a long line for each paragraph, this setting isn't ideal in this case, so I recommend turning on word wrap (called "soft wrap" in Atom) using the menu item shown in Figure 2.4.

In some editors, such as the cloud IDE at Cloud9, it's more common to open files using the filesystem navigator (although in fact the **c9** command

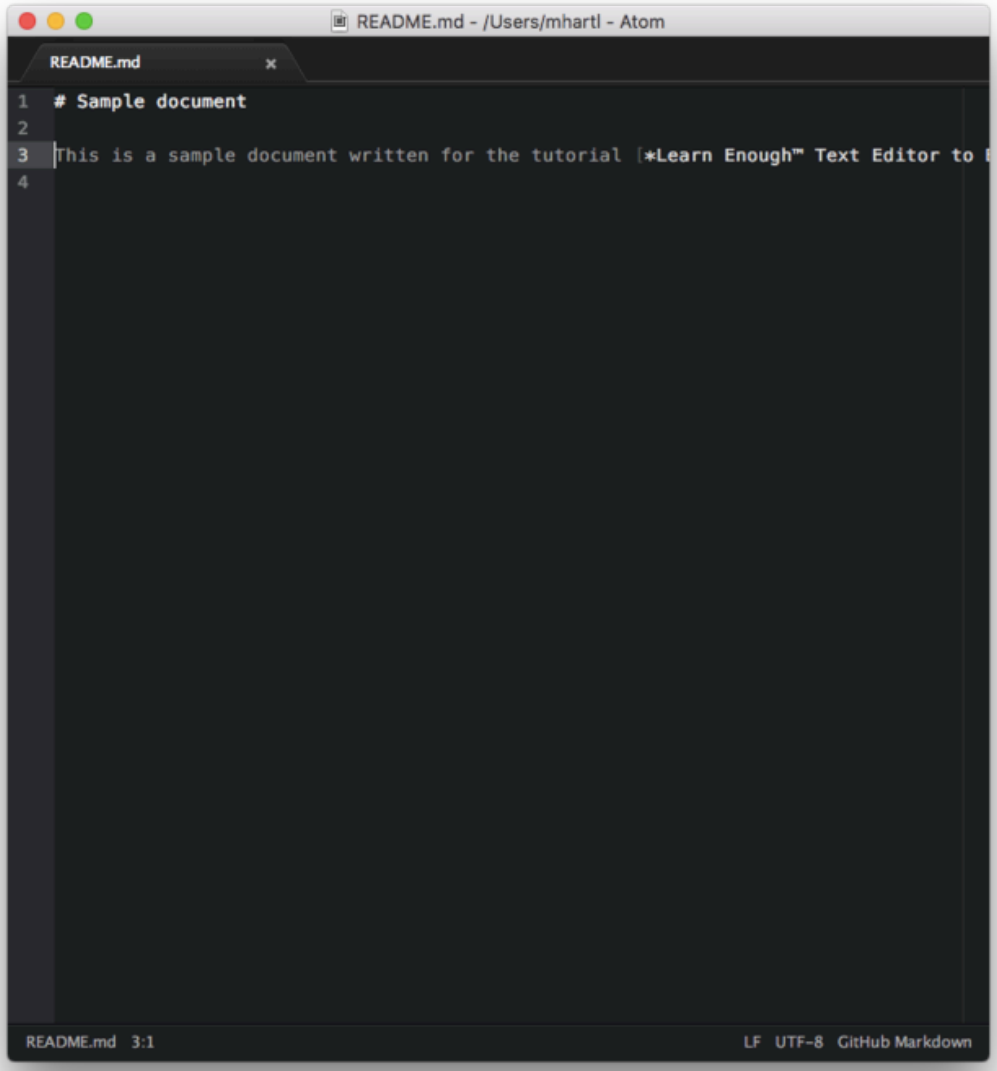

Figure 2.2: The sample file with word wrap off.

Figure 2.3: The sample file with word wrap on.

Figure 2.4: The menu item to toggle word wrap.

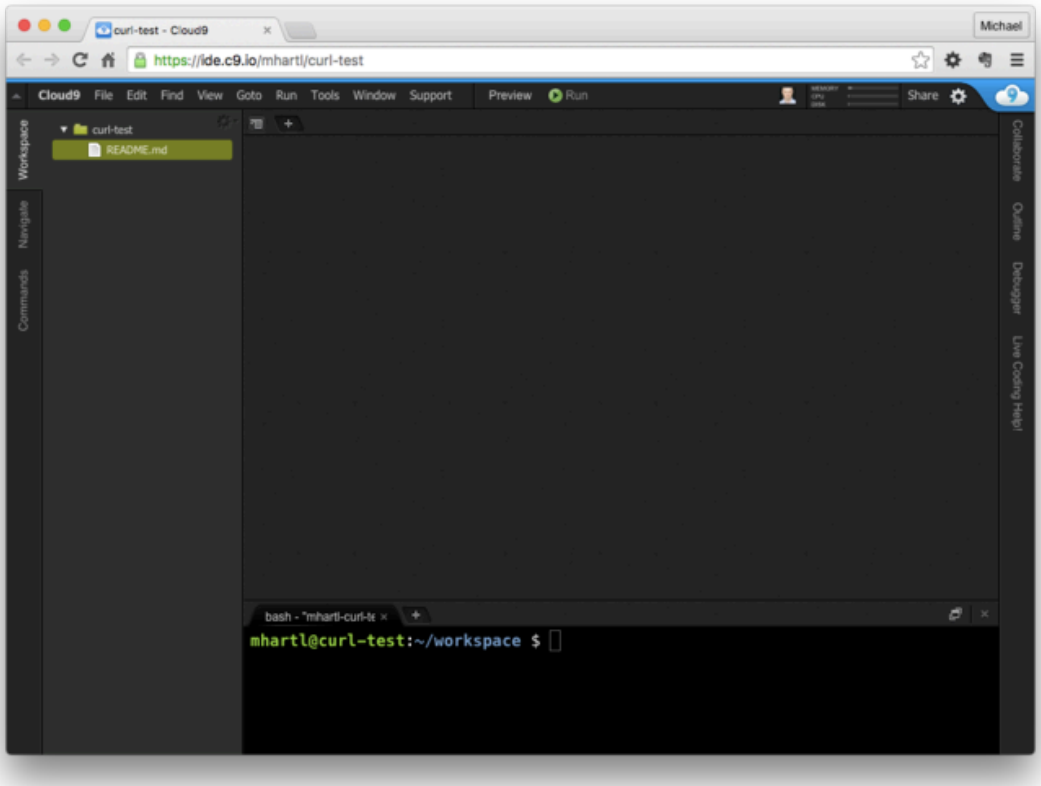

Figure 2.5: The Cloud9 filesystem navigator.

can be used to open files at the Cloud9 command line).[2] Double-clicking on **`README.md`** in the filesystem navigator (Figure 2.5) opens the file in Cloud9's editor, as shown in Figure 2.6. Figure 2.7 shows the file after clicking "Navigate" to close the filesystem navigator, and we see that, as in Figure 2.2, the line extends inconveniently off the screen. We can fix this using View > Wrap Lines as shown in Figure 2.8, with the word-wrapped result appearing as in Figure 2.9. (Figuring out that a menu item like "View > Wrap Lines" turns on word wrap is exactly the kind of thing you should be able to figure out using your technical sophistication (Box 1.3).)

[2] I used Cloud9 for over a year without discovering the **`c9`** command. Thanks to alert reader Timothy Kiefer for using his *technical sophistication* (Box 1.3) to figure it out!

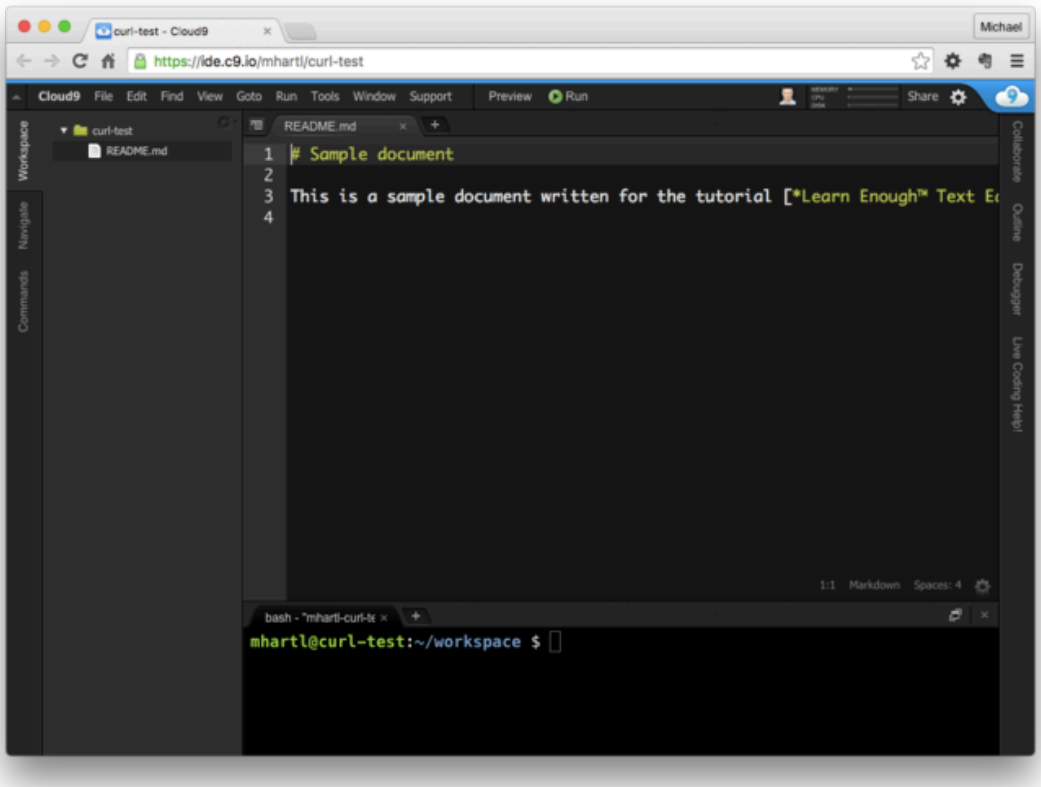

Figure 2.6: Cloud9 after double-clicking on **`README.md`**.

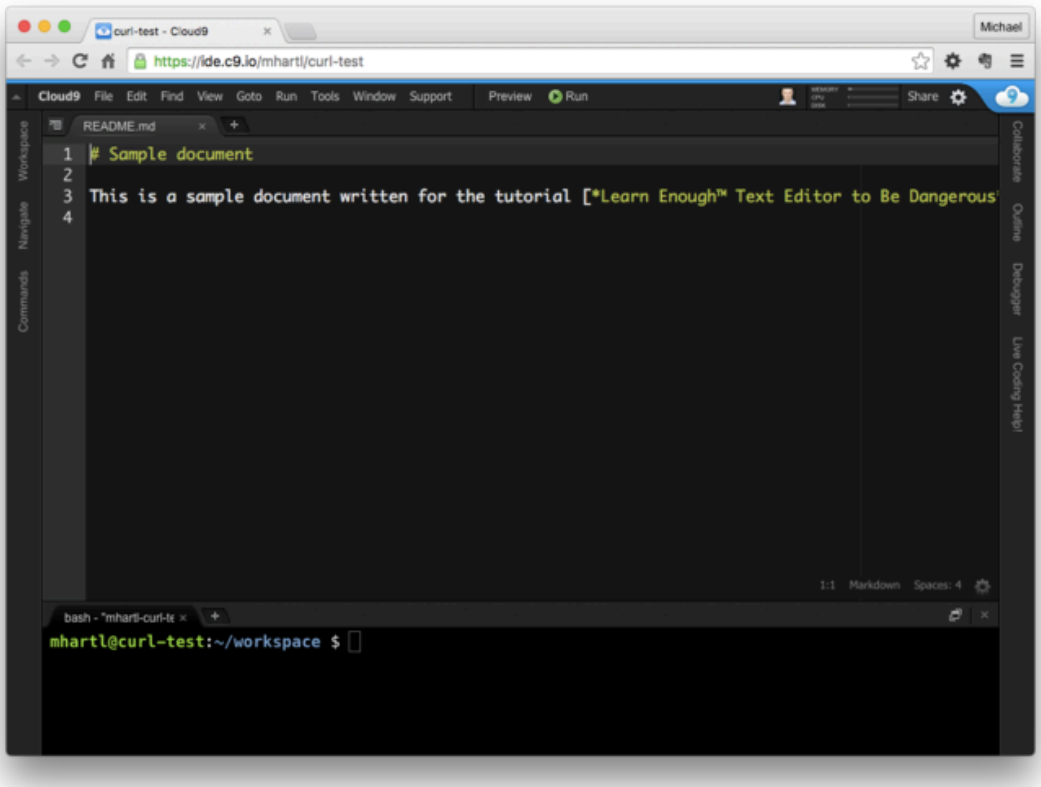

Figure 2.7: Cloud9 with word wrap off.

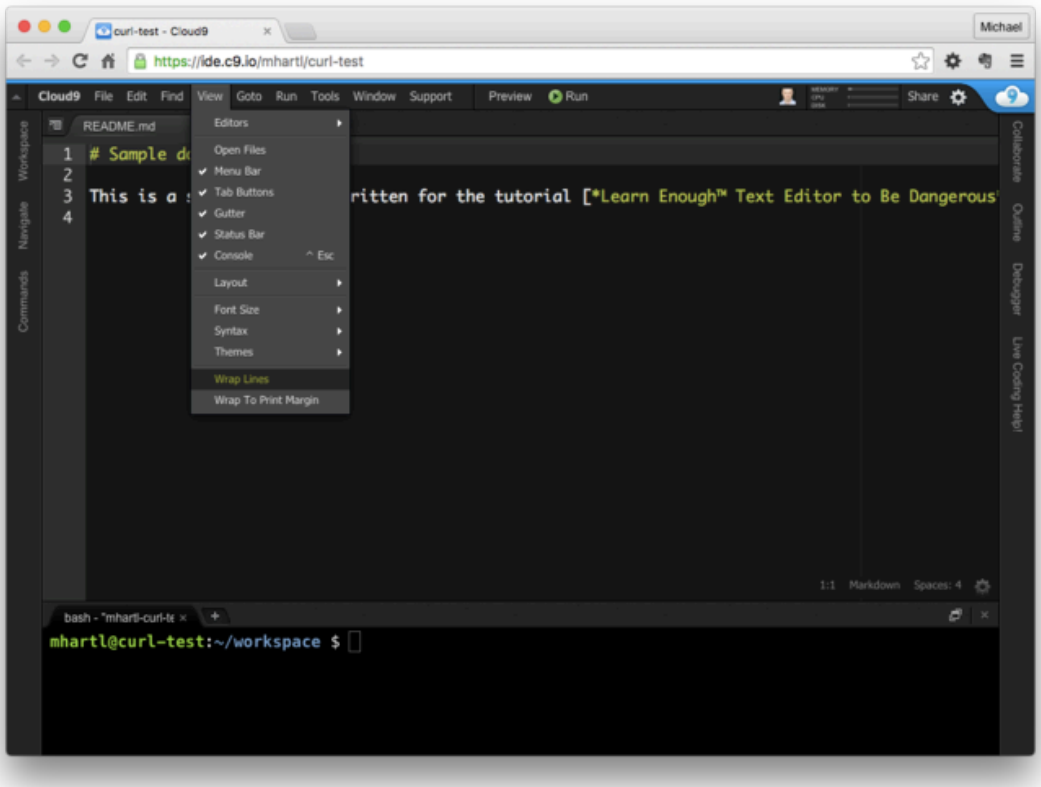

Figure 2.8: Turning word wrap on in Cloud9.

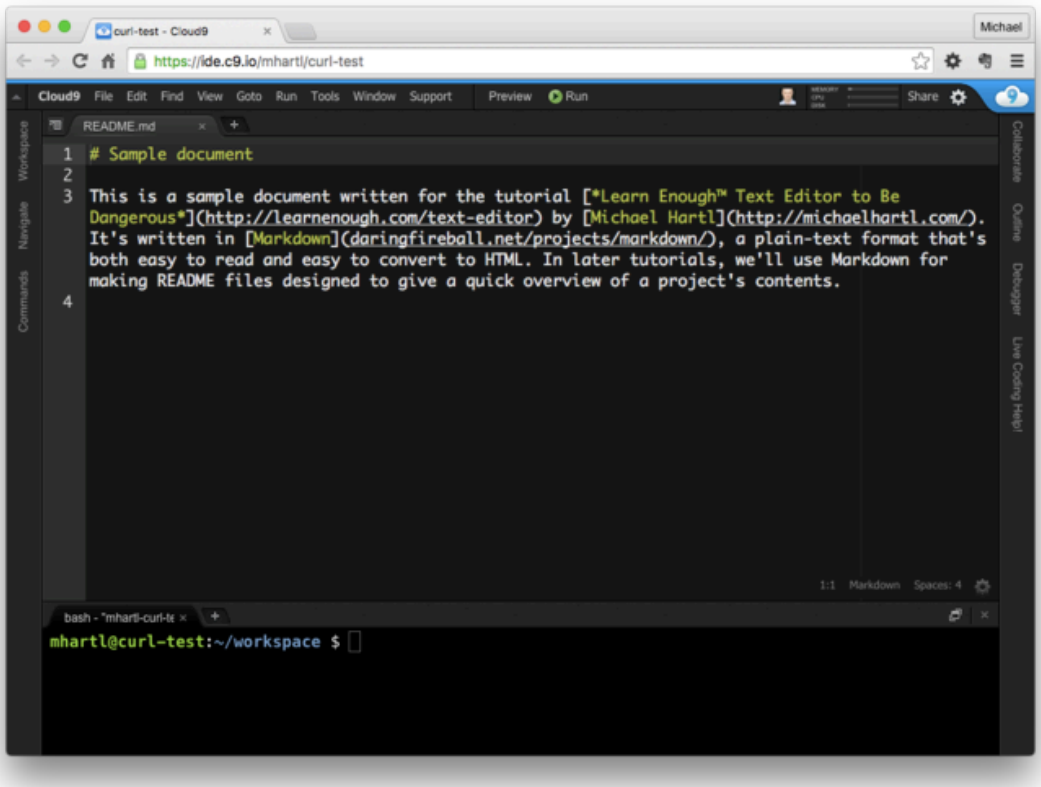

Figure 2.9: Cloud9 with word wrap on.

2.2.1 Syntax highlighting

One thing you may have noticed from inspecting Figure 2.3 and Figure 2.9 is that both Atom and Cloud9 display different aspects of the file in different colors. For example, Atom shows characters inside square brackets `[ ]` (which represent text for HTML links) in a lighter color than the rest of the text, while Cloud9 shows the same text in green. This is a practice known as *syntax highlighting*, which makes special text formatting much easier to identify visually. It's essential to understand that this practice is strictly for our benefit; as far as the computer is concerned, the document in question is still plain text.

You might wonder how Atom and Cloud9 knew which highlighting scheme to use. The answer is that they infer the document format from the file type extension (in this case, `.md` for Markdown). The highlighting in Cloud9's case is quite high-contrast, but in Atom's case it isn't particularly prominent; the most significant things are the different colors for the heading

```
# Sample document
```

and for links like

```
[Michael Hartl](http://michaelhartl.com/)
```

We'll see more dramatic examples of syntax highlighting in Section 2.7 and especially in Section 3.2.

2.2.2 Previewing Markdown

As a final trick, I'd like to note that some editors, including Atom, can preview Markdown as HTML. We can figure out how to do this using our technical sophistication (Box 1.3), in this case by clicking on the Help menu and searching for "Preview" (Figure 2.10). The result is a built-in package called Markdown Preview, which converts Markdown to HTML and shows the result, as seen in Figure 2.11. In this context, it's convenient to work with an expanded width so

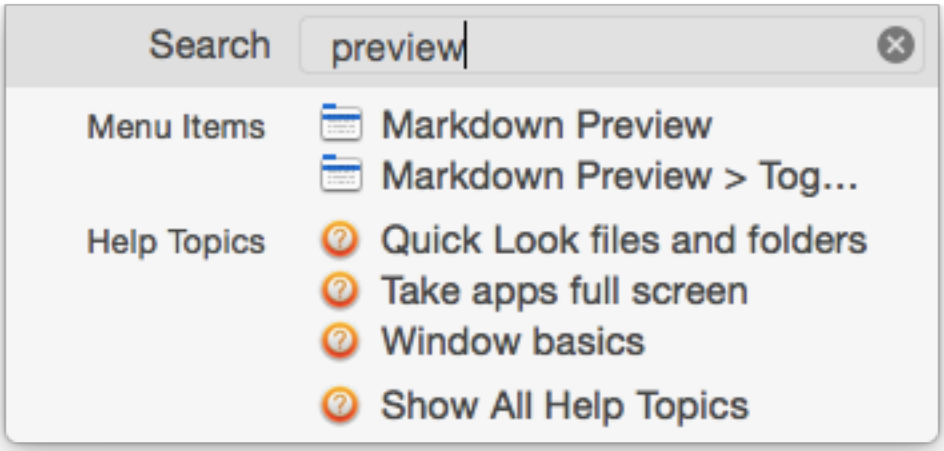

Figure 2.10: Using the Help menu to learn how to preview Markdown.

that both the source and the preview are wide enough to view easily, as seen in Figure 2.12. This is accomplished by mousing over the side of the Atom window to get a double-arrow icon and then dragging to increase the size. We'll see another example of this "double-paned" setup in a more general setting starting in Section 3.4.

2.2.3 Exercises

1. By applying the methods in Box 1.3, find an online Markdown previewer (i.e., one that runs in a web browser), and use it to look at a preview of **`README.md`**. How do the results compare to Figure 2.11?

2. Open a new document called **`lorem.txt`** and fill it with the text shown in Listing 2.1. Does the result have syntax highlighting?

3. Open a new document called **`test.rb`** and fill it with the text shown in Listing 2.2. Does the result have syntax highlighting?

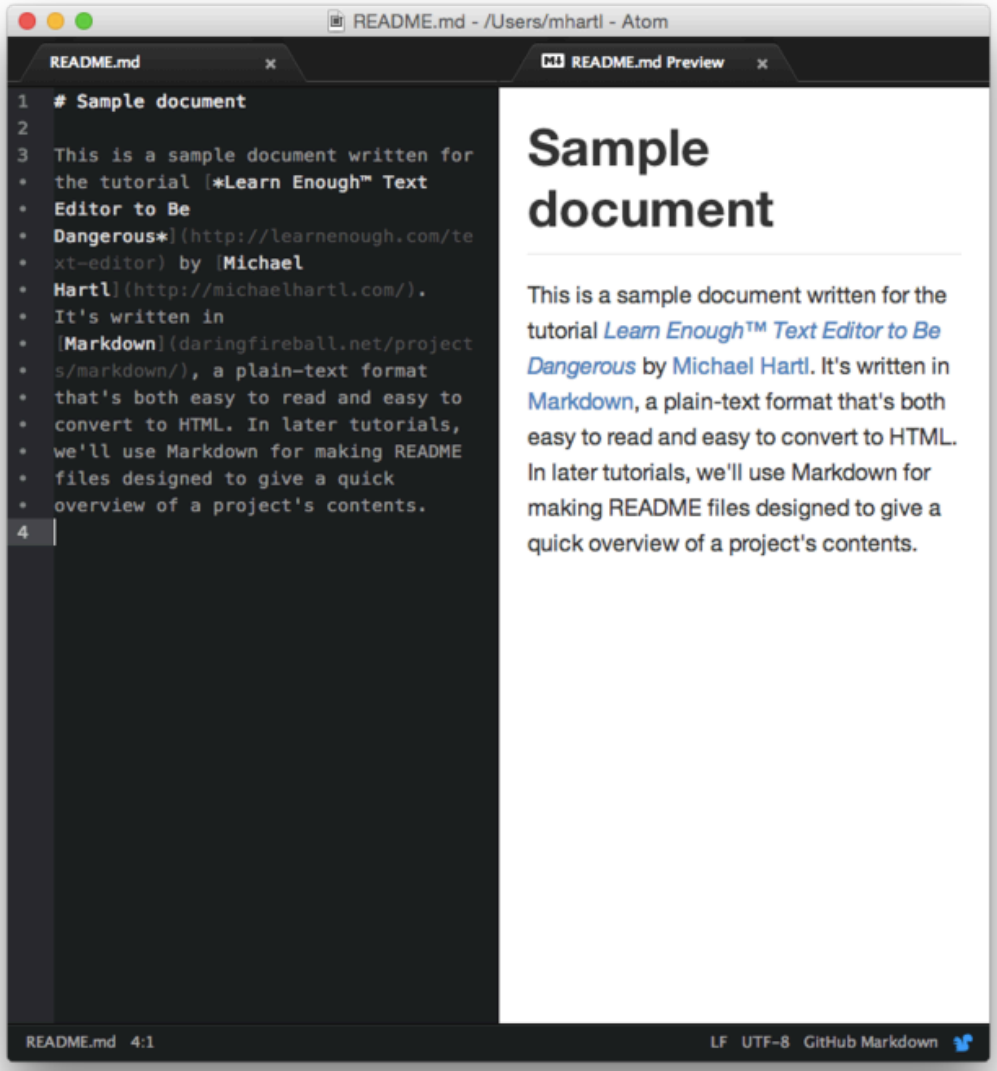

Figure 2.11: A Markdown preview in Atom.

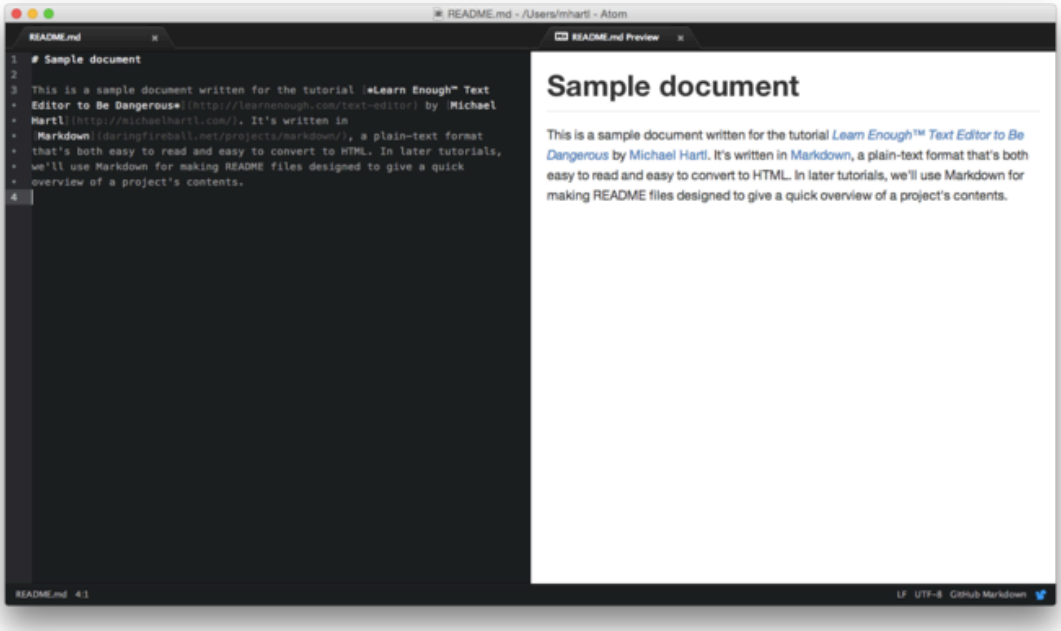

Figure 2.12: Using a wider window for the source and preview.

Listing 2.1: Some lorem ipsum text.
~/lorem.txt

```
Lorem ipsum dolor sit amet
```

Listing 2.2: A test file.
~/test.rb

```
puts "test"
```

2.3 Moving

Unlike the commands for moving around in Vim (Chapter 1, summarized in Table 1.1), the commands for moving around in modern editors generally match the techniques used in other programs such as word processors, email programs, and web browsers. As a result, it's possible you may already know some or all of these techniques; if you don't, by following the steps in this section you'll get better at navigating other programs as a side-effect.

To get started, let's open the large file from Section 1.6 consisting of the full text of Shakespeare's *Sonnets*:

```
$ atom sonnets.txt
```

(If this doesn't work, you may need to run the command in Listing 1.7, and you should also verify that you're in the right directory.) The result appears in Figure 2.13. Note that Figure 2.13 shows **`sonnets.txt`** in its own tab, with **`README.md`** from Section 2.2 occupying the other tab. Your result may vary; in any case, we'll discuss tabs further in Section 3.4.

As with most other native programs such as word processors, web browsers, etc., you can move around a modern editor using the mouse or trackpad. You can click to place the cursor, scroll using a scroll wheel or multi-touch gestures, or click and drag the scrollbar. The last of these is (as of this writing) incredibly

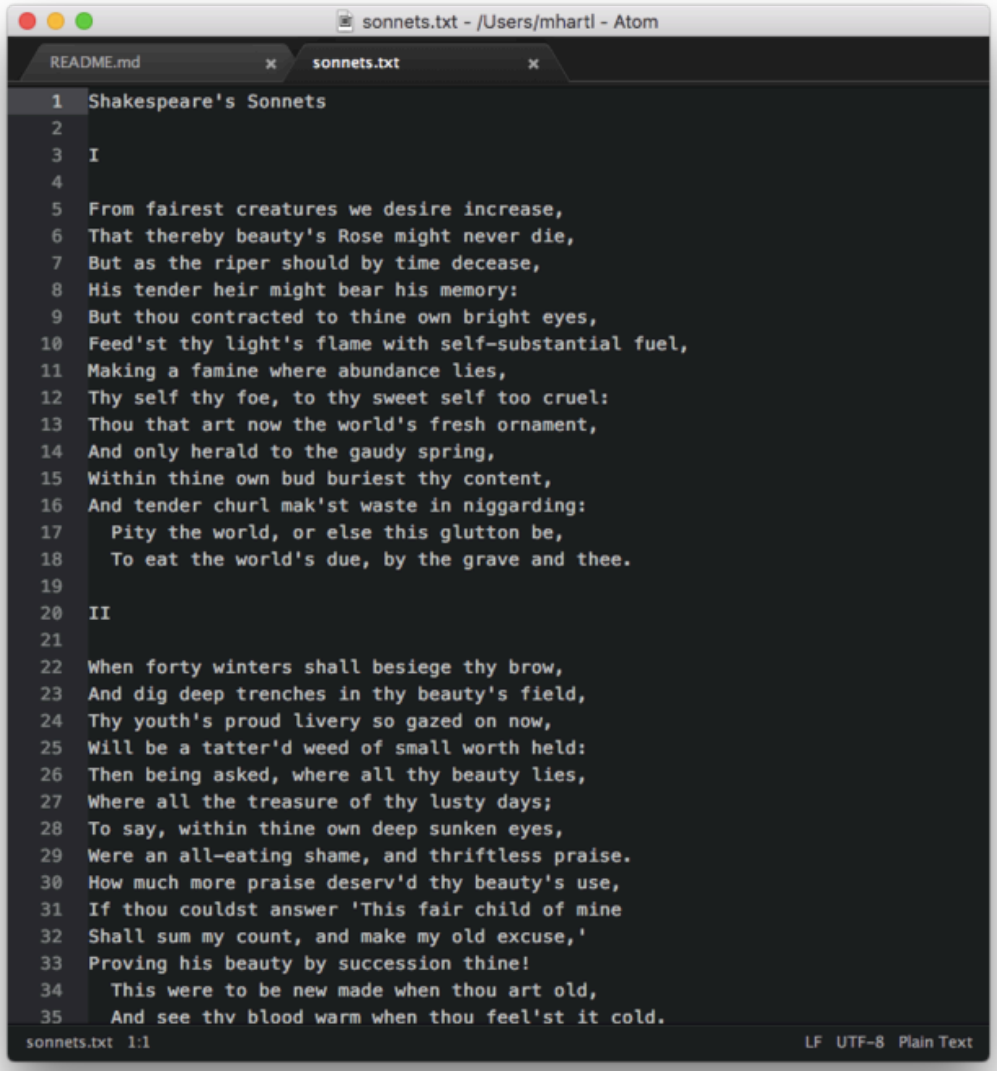

Figure 2.13: Opening Shakespeare's *Sonnets* in Atom.

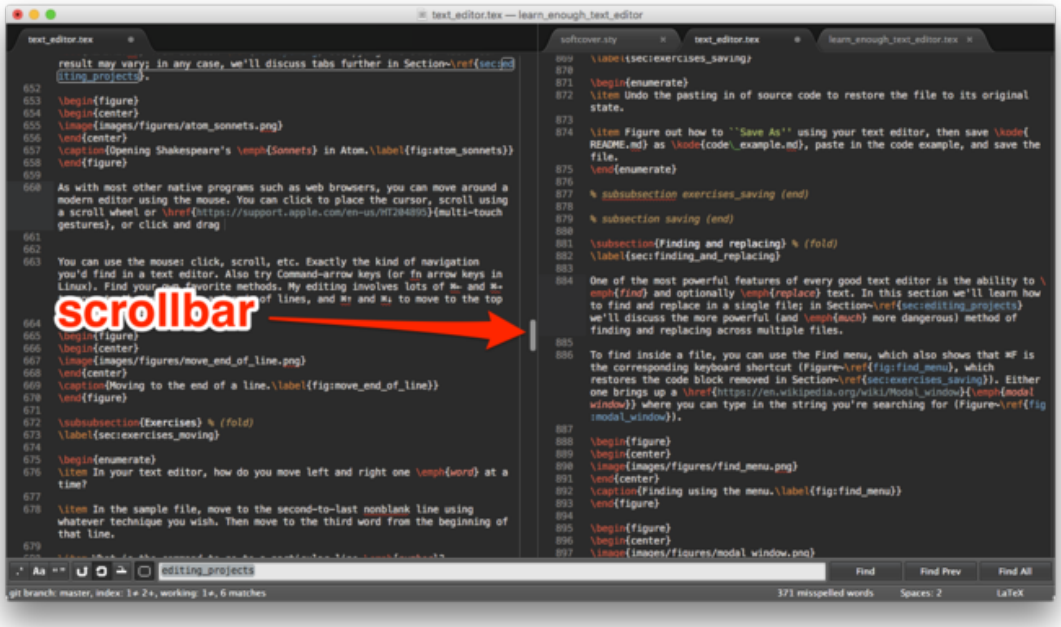

Figure 2.14: The Sublime Text scrollbar.

subtle in Atom, so Figure 2.14 shows the scrollbar for Sublime Text. Figure 2.14 also shows the sort of two-pane view mentioned briefly in Figure 2.12, which we'll discuss more in Section 3.4.

In addition to using the mouse or trackpad, I also like using the arrow keys to move around, typically in concert with the Command key ⌘ (Table 2.1). (In Linux, Command is typically replaced with the Function key fn, and in Windows it's usually Ctrl, but you'll have to apply Box 1.3 to figure out the details.) My text editing typically involves lots of ⌘← and ⌘→ to move to the beginnings and ends of lines, and ⌘↑ and ⌘↓ to move to the top and bottom of the file. An example of moving to the end of the line in **`README.md`** appears in Figure 2.15, and an example of moving to the end of the file in **`sonnets.txt`** appears in Figure 2.16.

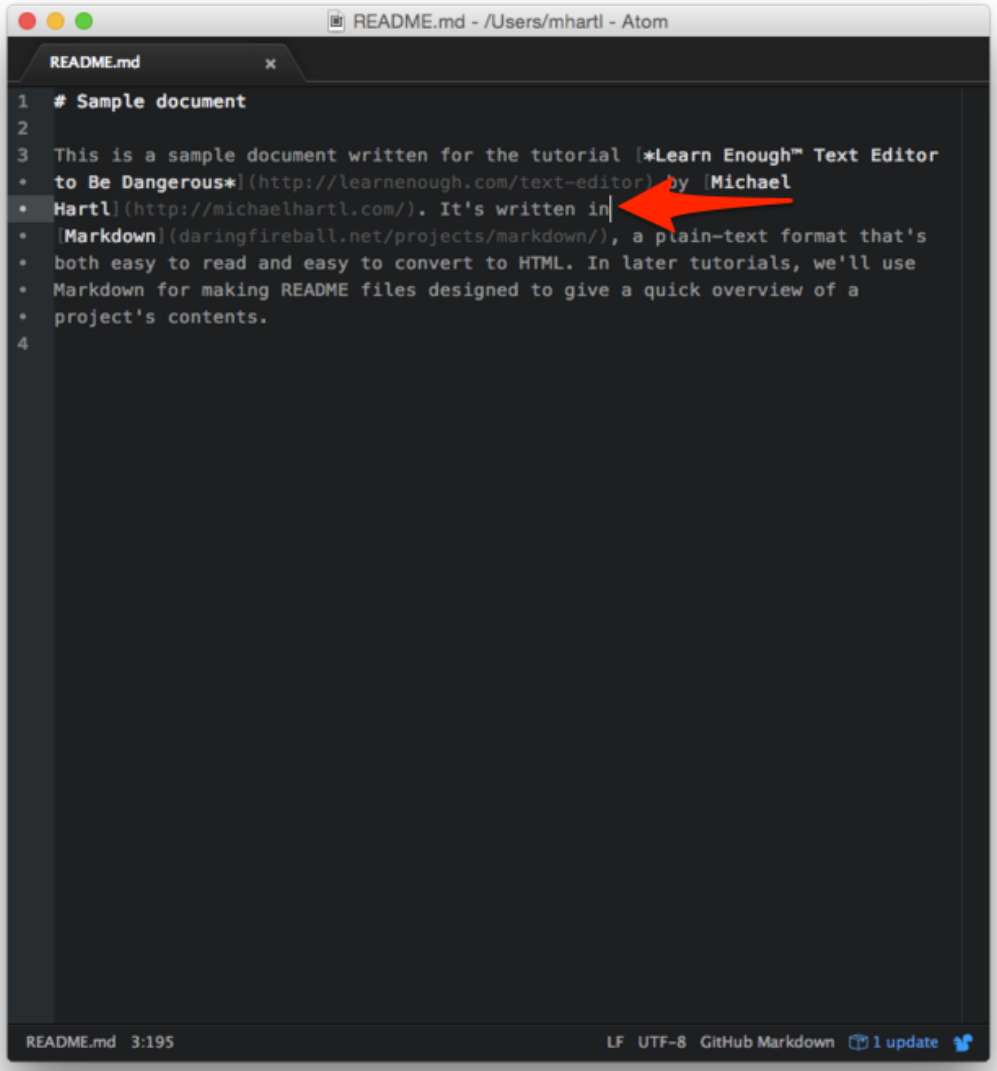

Figure 2.15: Moving to the end of a line with ⌘→.

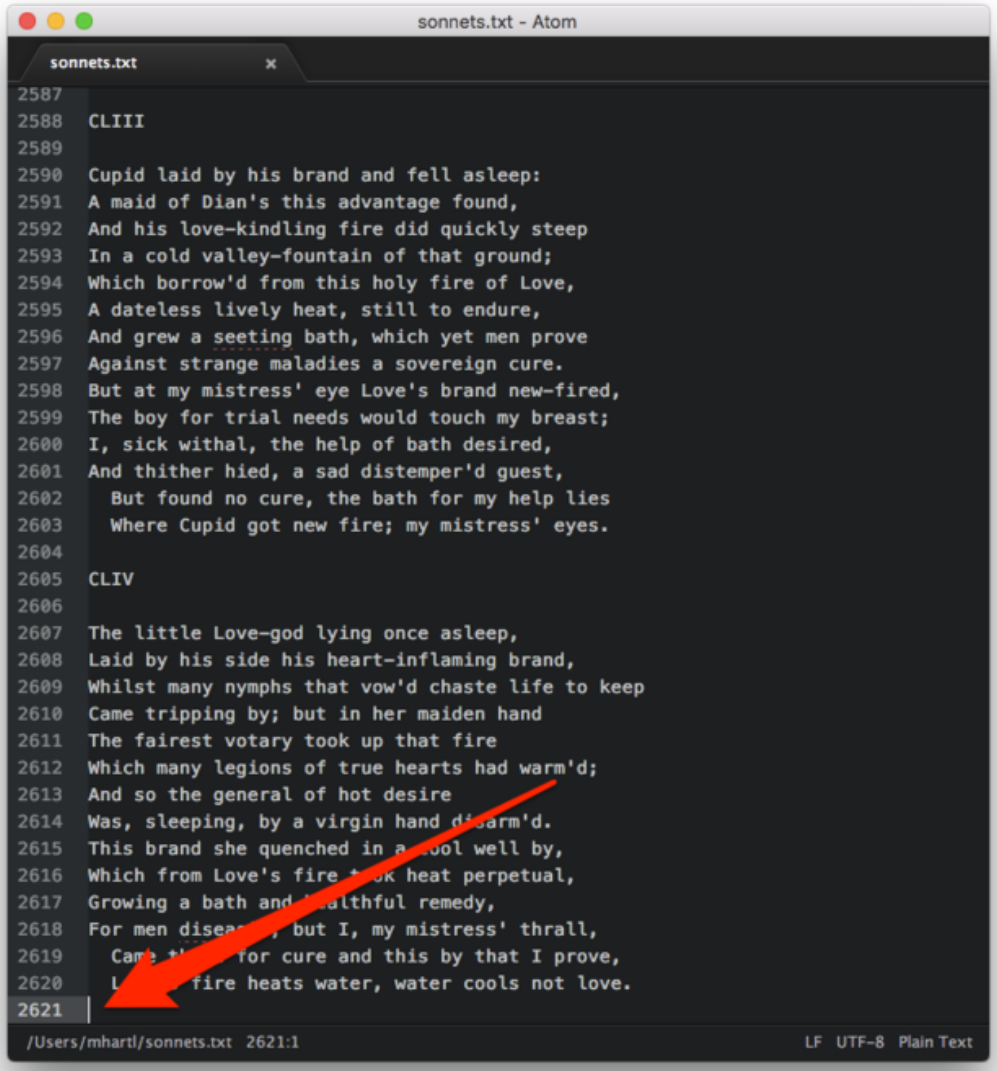

Figure 2.16: Moving to the end of the file with ⌘↓.

2.3.1 Exercises

1. In your text editor, how do you move left and right one *word* at a time? *Hint:* On some systems, the Option key ⌥ might prove helpful.

2. In `README.md`, move to the second-to-last nonblank line using whatever technique you wish. Then move to the third word from the beginning of that line.

3. What is the command to go to a particular line *number*? Use this command to go to line 293 of `sonnets.txt`. What do rough winds do?

4. By moving to the last nonblank line of `sonnets.txt` and pressing ⌘→ followed by ⌘←, show that ⌘← actually stops as soon as it reaches whitespace, with the result shown in Figure 2.17. How do you get to the true beginning of the line?

2.4 Selecting text

Selecting text is an important skill that is particularly useful for deleting or replacing content, as well as for cutting, copying, and pasting (Section 2.5). Many of the techniques in this section make direct application of the commands to move around covered in Section 2.3. As in that section, the ideas here are quite general, applying to a wide variety of applications, not just to text editors.

In much the same way that modern editors make it easy to use the mouse to move the cursor, they also make it easy to use the mouse to select text. Simply click and drag the mouse cursor, as shown in Figure 2.18. Another closely related technique is to click on one location, and then Shift-click on another location to select all the text in between.

2.4.1 Selecting a single word

When selecting text, there are some special cases that are useful enough to consider individually. We'll start with some techniques for selecting a single word:

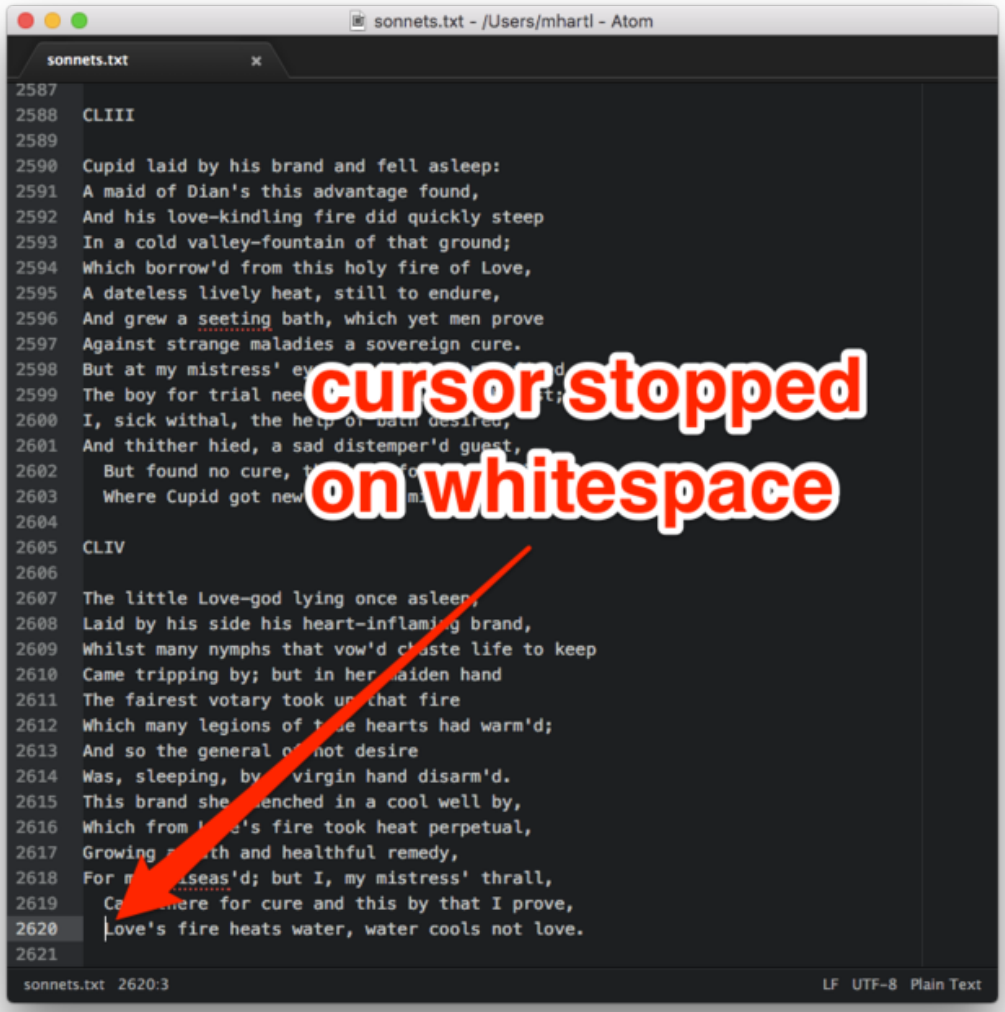

Figure 2.17: When using ⌘←, the cursor stops on whitespace.

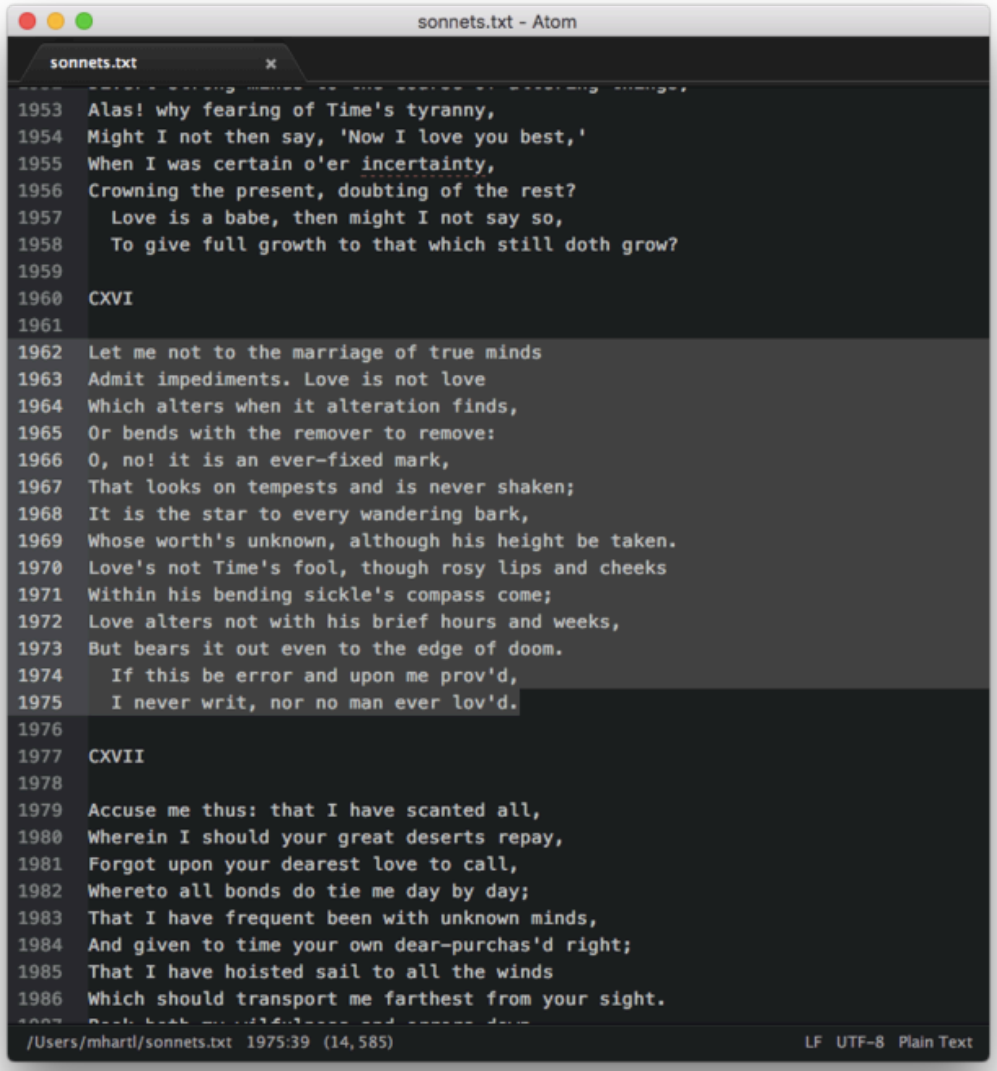

Figure 2.18: The result of clicking and dragging the mouse cursor.

- Click and drag the mouse cursor over the word
- Double-click the word with the mouse
- Press ⌘D (system-dependent; see Box 1.3)

2.4.2 Selecting a single line

Another technique, especially important when editing line-based text like computer code (or sonnets), involves selecting a full line or collection of lines. We start with ways to highlight a single line:

- Click the beginning of the line and drag the cursor to the end
- Click the end of the line and drag the cursor to the beginning
- Press ⌘← (twice) to get to the beginning of line, then press ⇧⌘→ to select to the end of line
- Press ⌘→ to get to the end of line, then press ⇧⌘← (twice) to select to the beginning of line

2.4.3 Selecting multiple lines

A comparably important technique is selecting multiple lines:

- Click and drag the mouse cursor over the words/lines
- Hold down the Shift key and move the up and down arrow keys (⇧↑ and ⇧↓)

This latter technique is one of my personal favorites, and one of my most common editing tasks involves hitting ⌘← to go to the beginning of the first line I want to select and then hitting ⇧↓ repeatedly until I've selected all the lines I want (Figure 2.19). (As noted in Section 2.3.1, in many editors ⌘← stops on whitespace, so moving to the beginning of the line actually requires two uses of ⌘← in succession. Being able to figure out details and edge cases like this is a hallmark of growing technical sophistication (Box 1.3).)

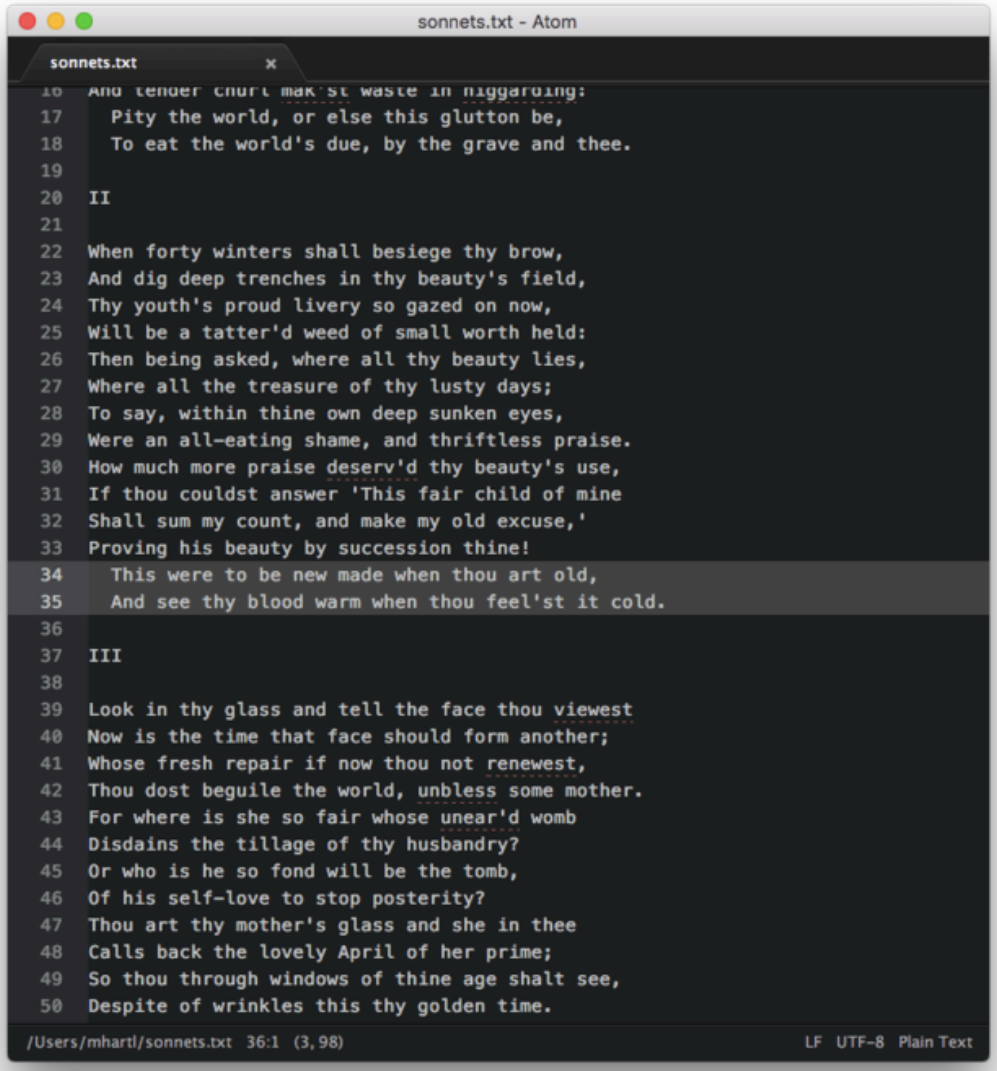

Figure 2.19: Selecting a Shakespearean couplet using ⌘← and ⇧↓.

2.4.4 Selecting the entire document

Finally, it's sometimes useful to be able to select the entire document at once. For this, there are two main techniques:

- Use a menu item called "Select All" or something similar. The specifics are editor-dependent; Figure 2.20 shows the use of the Selection menu in Sublime Text, while Figure 2.21 shows the use of the Edit menu in Atom.
- Press ⌘A

Note from Figure 2.20 that the menu actually shows the corresponding command (⌘A); bootstrapping your knowledge using the menu items is a great way to learn keyboard shortcuts, which over time will make your text editing significantly more efficient.

2.4.5 Exercises

1. Select Shakespeare's second sonnet by clicking at the beginning and then Shift-clicking at the end.
2. Select the first line in the file by moving to the beginning with ⌘↑ and pressing ⇧⌘→ (or the equivalent for your system).
3. Delete the selection in the previous exercise (using the Delete key).
4. Select the word "document" in **`README.md`** and replace it with "README".

2.5 Cut, copy, paste

The Cut/Copy/Paste triumvirate is one of the most useful sets of operations when editing text, especially when executed via the conveniently located keyboard shortcuts ⌘X/⌘C/⌘V. (Cut/Copy/Paste are available as menu items (Figure 2.22), but the operations are so common that I strongly recommend

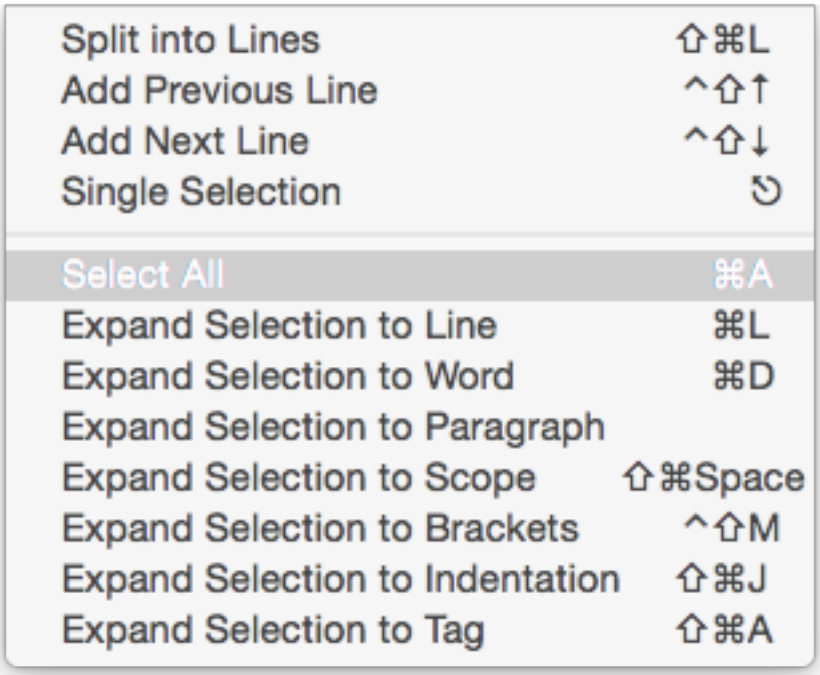

Figure 2.20: Selecting the entire document using the Selection menu (Sublime Text).

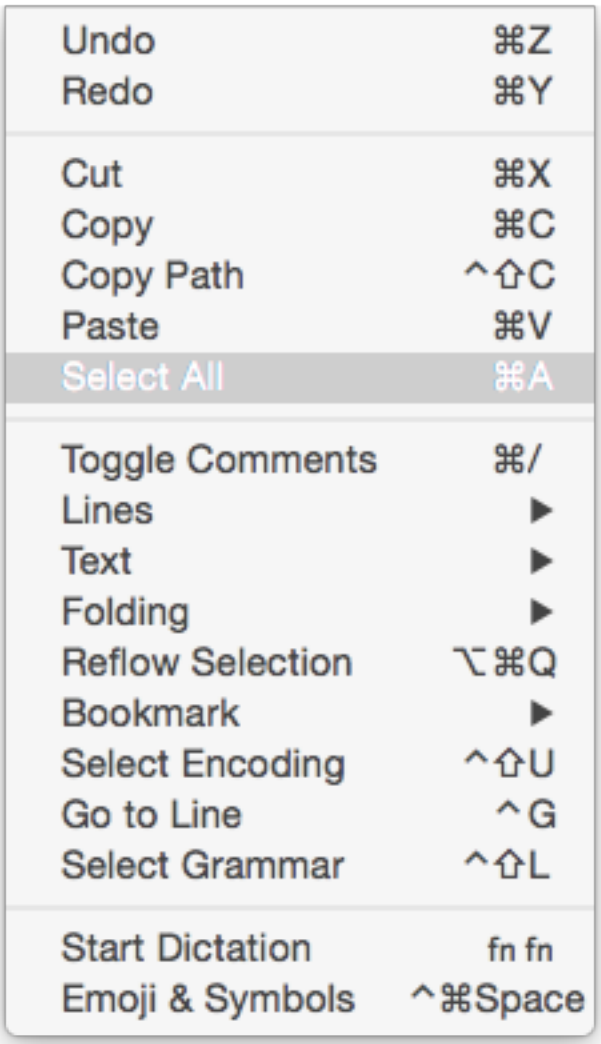

Figure 2.21: Selecting the entire document using the Edit menu (Atom).

learning and using the keyboard shortcuts right away.) Although only ⌘C is mnemonic ("C" for "Copy"), the keys are conveniently located three in a row on the bottom row of a standard QWERTY keyboard, which makes it easy to use them in combination or in quick succession (Figure 2.23).

Applying either Cut or Copy involves first selecting text (Section 2.4), and then hitting either ⌘X to Cut or ⌘C to Copy. When using ⌘C to Copy, the selected text is placed in a *buffer* (temporary memory area); moving to the desired location (Section 2.3) and hitting ⌘V lets you Paste the content into the document at the location of the cursor. ⌘X works the same way as ⌘C, except the text is removed from the document as well as being copied into the buffer.

As a concrete example, let's select a Markdown link from the sample `README` file, **`README.md`**, as shown in Figure 2.24. After copying with ⌘C, we can then paste the link several times (with returns in between) by repeatedly hitting ⌘V and the Enter key, as shown in Figure 2.25. Finally, Figure 2.26 shows the result of cutting **`README`** from the main text and pasting it in at the end of the file.

2.5.1 Jumpcut

Although Cut/Copy/Paste is all that's strictly necessary for everyday editing, there is one big downside, which is that there is only room in the buffer for a single string. Among other things, this means that if you Cut something and then accidentally hit "copy" instead of "paste" (which is easy since the letters are adjacent on the keyboard), you overwrite the buffer, and the text you Cut is gone forever (unless you undo as described in Section 2.6). If you happen to be developing on a Mac, there's a solution to this problem: a free program called Jumpcut. This remarkable little utility app expands the buffer by maintaining more than one entry in the history. You can navigate this expanded buffer either using the Jumpcut menu (Figure 2.27) or the keyboard shortcuts ^⌥V (cycle forward) and ⇧^⌥V (cycle backward). I use Jumpcut dozens or even hundreds of times a day, and I strongly suggest giving it a try.

Figure 2.22: The Cut/Copy/Paste menu items (which you should never use).

Figure 2.23: The XCV keys on a standard QWERTY keyboard.

2.5.2 Exercises

1. Select the entire document, Copy it, and Paste several times. The result should look something like Figure 2.28.

2. Select the entire document and Cut it. Why might this be preferable to deleting it?

3. Select and copy the couplet at the end of Sonnet 1 and paste it into a new file called `sonnet_1.txt`. How do you create a new file directly in your editor?

2.6 Deleting and undoing

We mentioned deleting before in Section 2.4.5 (the exercises for Section 2.4), which of course simply involves pressing the Delete key, sometimes written as ⌫ (Table 2.1). As with Cut/Copy/Paste (Section 2.5), deletion is especially useful when combined with the selection techniques from Section 2.4.

Figure 2.24: Selecting a Markdown link

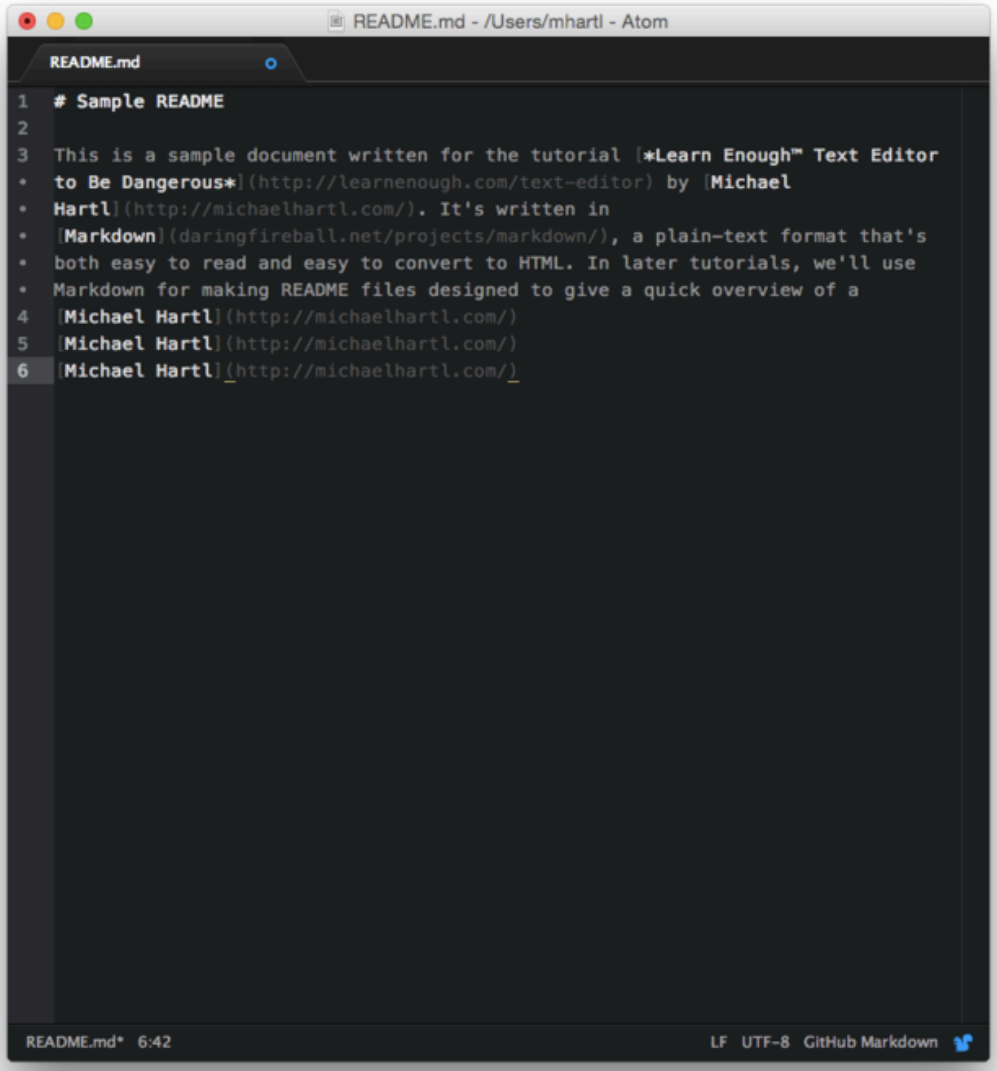

Figure 2.25: Pasting link text several times (with returns in between).

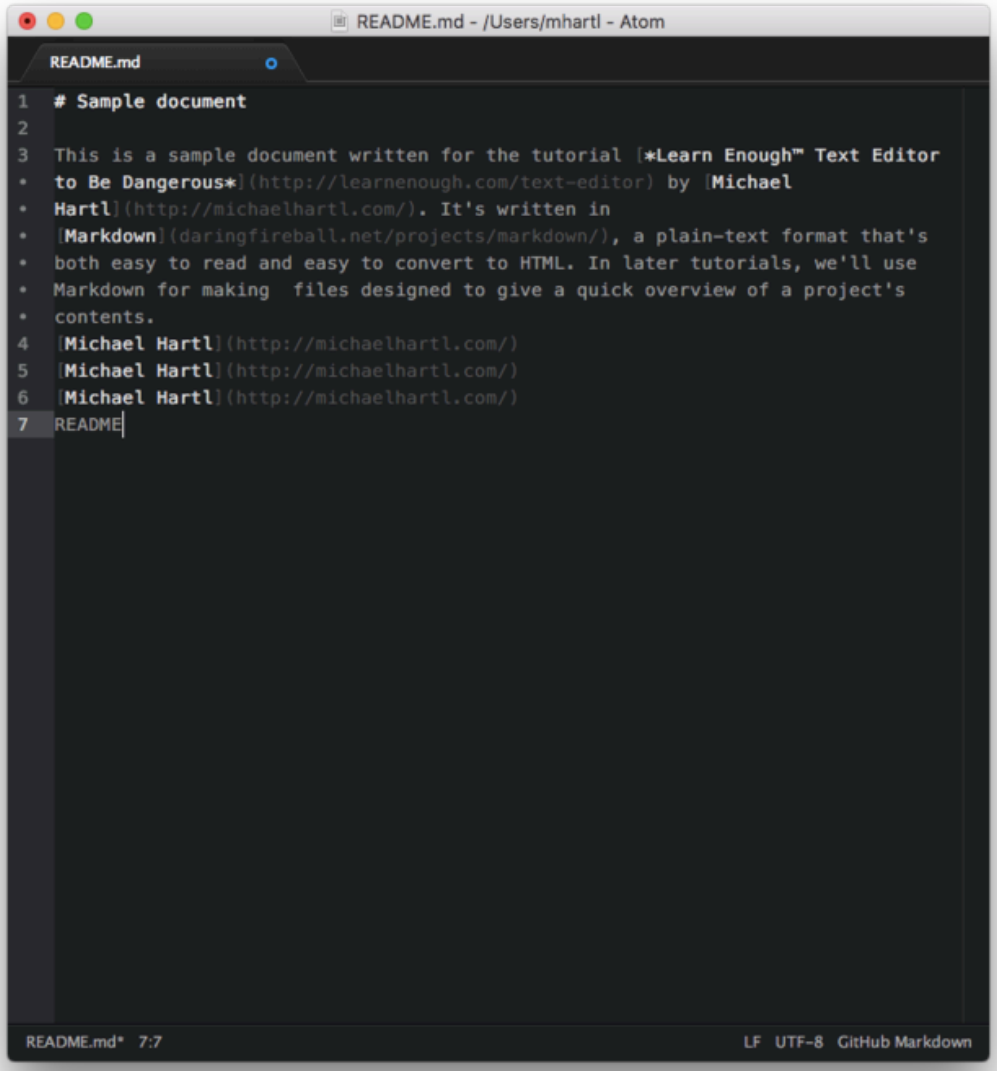

Figure 2.26: The result of cutting "README" and pasting at the end of the file.

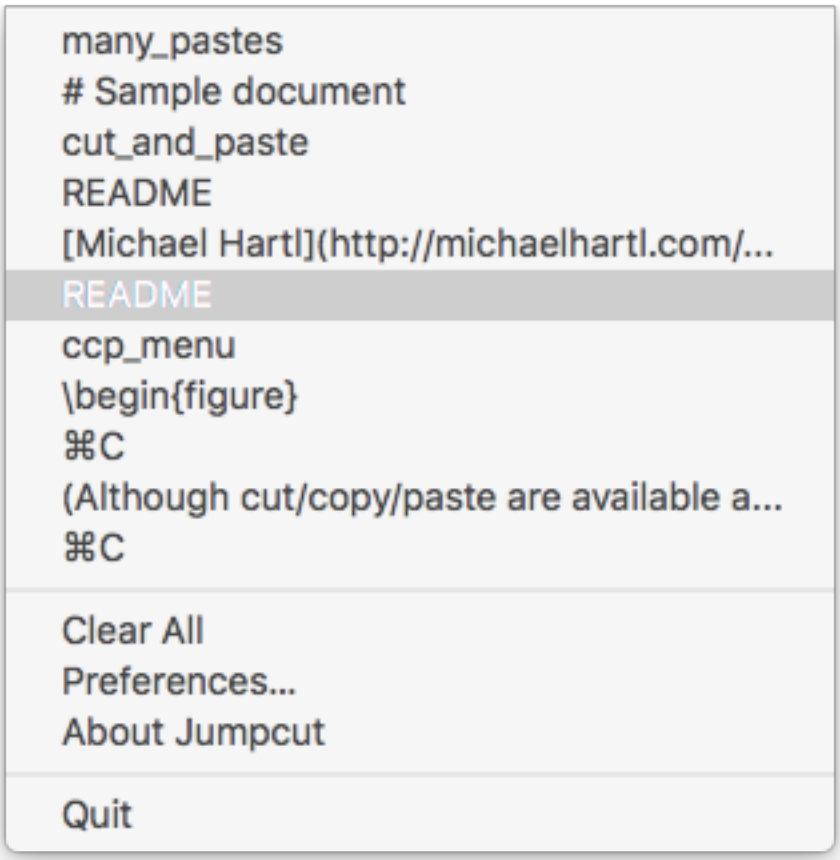

Figure 2.27: Jumpcut expands the copy-and-paste buffer to include a longer history.

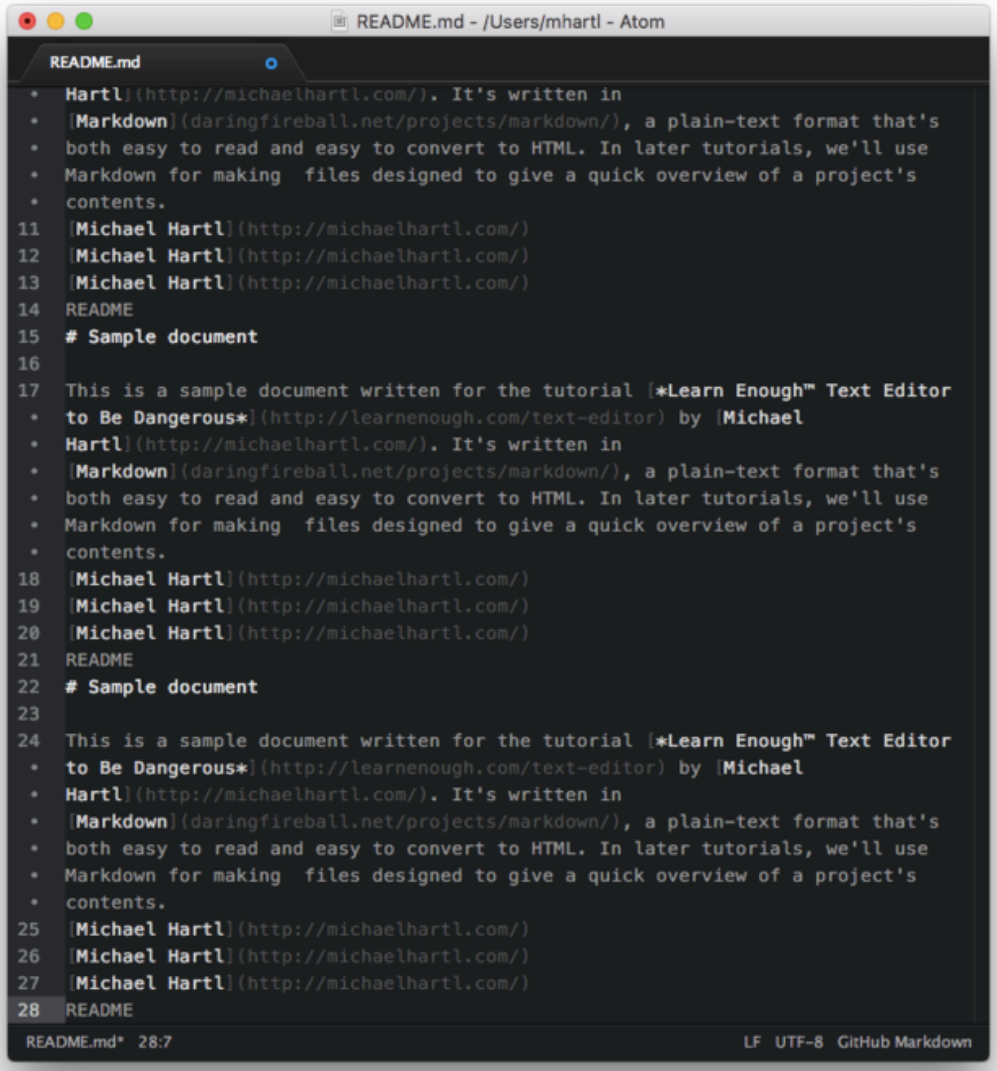

Figure 2.28: The result of pasting the whole document several times.

In addition to the obvious technique of selecting and deleting text, on a Mac I especially like using ⌥⌫ to delete one word at a time. I'll frequently use this combination if I need to delete a medium number of words (say 2–5) to restart a phrase when writing. For shorter deletion tasks, such as one word, it's usually faster to hit ⌫ repeatedly, as context-switching to use ⌥⌫ incurs some overhead that makes it faster to just delete directly. Don't worry too much about these micro-optimizations, though; with experience, as a matter of course you'll come up with your own set of favorite techniques.

Paired with deletion is one of the most important commands in the history of the Universe, Undo. In modern editors, Undo uses the native keybinding, typically ⌘Z or ^Z. Its inverse, Redo, is usually something like ⇧⌘Z or ⌘Y. You can also use the menu (typically Edit), but, as with Cut/Copy/Paste (Section 2.5), Undo is so useful that I recommend memorizing the shortcut as soon as possible. Without Undo, operations like deletion would be irreversible and hence potentially harmful, but with Undo it's easy to reverse any mistakes you make while editing.

One practice I recommend is using Cut instead of Delete whenever you're not 100% sure you'll never want the content again. Although you can usually Undo your way to safety if you accidentally delete something important, putting the content into the buffer with Cut gives you an additional layer of redundancy. (Using Jumpcut (Section 2.5.1) gives you another layer still.)

Finally, Undo provides us with a useful trick for finding the cursor, a common task when editing larger files. The issue is that you'll be writing some text and then need to move (Section 2.3) or find (Section 2.8) elsewhere in the document. On these occasions, it can be hard to relocate the cursor. There are several ways around this problem—you can move the arrow keys, or just start typing—but my favorite technique is to Undo and then immediately Redo (⌘Z/⇧⌘Z or ⌘Z/⌘Y), which is guaranteed to find the cursor without making any undesired changes.

2.6.1 Exercises

1. Use Undo repeatedly until all the changes you've made to **`README.md`** have been undone.

Figure 2.29: Undo and Redo in the editor menu.

2. Using any technique you want from Section 2.4, select the word "written" in **`README.md`** and delete it, then undo the change.

3. Redo the change from the previous exercise, then undo it again.

4. Make an edit somewhere in **`sonnets.txt`**, then scroll around so you get lost. Use the Undo/Redo trick to find the cursor again. Then keep using Undo to undo all your changes.

2.7 Saving

Once we've made some edits to a file, we can save it using the menu or with ⌘S. I strongly recommend using the keyboard shortcut, which among other things makes it easier to save the file whenever you reach a temporary pause in your writing or coding—a valuable habit to cultivate. Basically, if you're not doing something else, you should be hitting Save. This habit goes a long way toward preventing lost work (and, as discussed in *Learn Enough Git to Be Dangerous*, is especially powerful when combined with version control).

As an example, we can add some source code to our README file and save the result. We start by pasting in the code from Listing 2.3, as shown in Figure 2.30 (which includes some nice high-contrast syntax highlighting). As you can see from the circled indicator in Figure 2.30, Atom (as with most modern editors) includes a subtle indicator that the file is unsaved, in this case a small open circle. After running Save (via ⌘S, for example), the circle disappears, to be replaced with an X (Figure 2.31).

Listing 2.3: A code snippet.

````
```ruby
def hello
  puts "hello, world!"
end
```
````

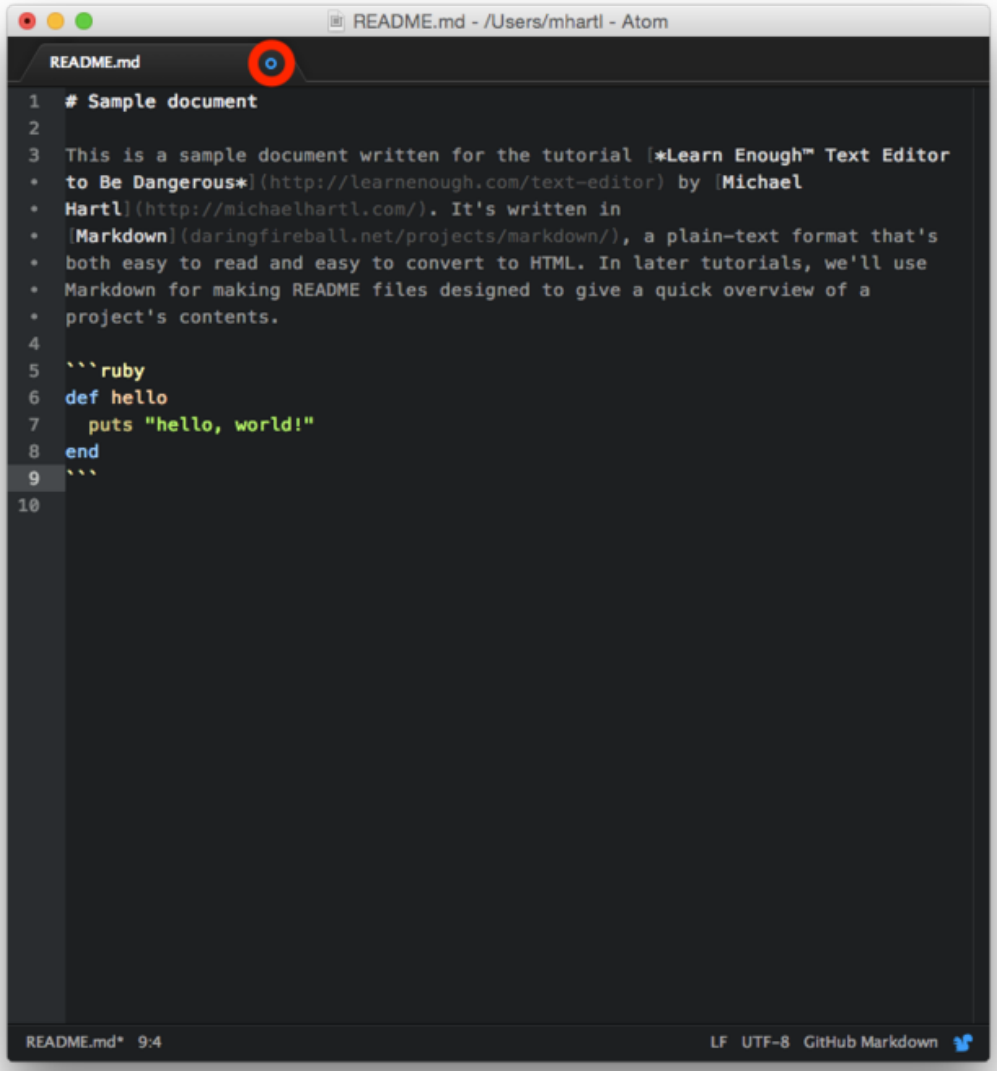

Figure 2.30: An unsaved file.

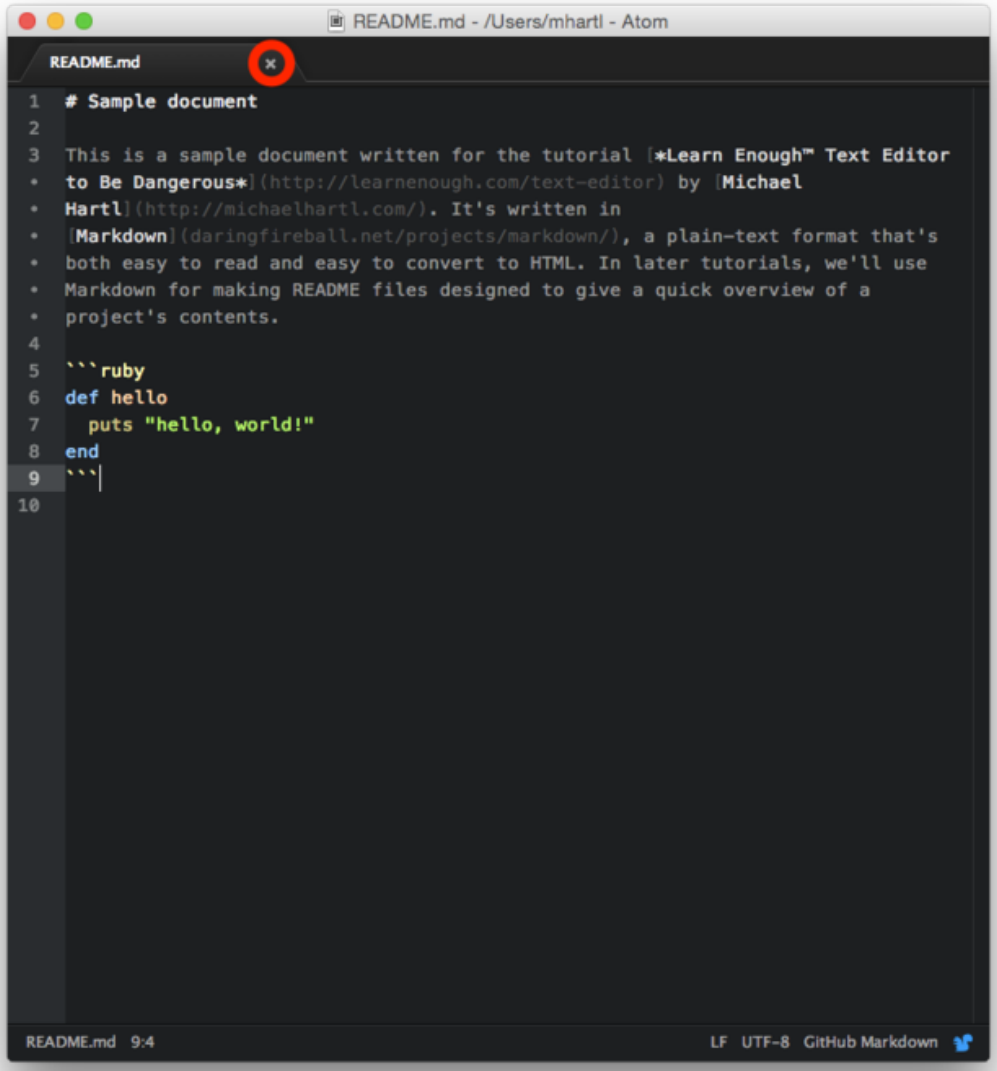

Figure 2.31: The file from Figure 2.30 after saving.

2.7.1 Exercises

1. Undo the pasting in of source code to restore the file to its original state.

2. Figure out how to "Save As", then save `README.md` as `code_example-.md`, paste in the code example, and save the file.

3. The default Bash prompt for my command-line terminal appears as in Listing 2.4, but I prefer the more compact prompt shown in Listing 2.5. In *Learn Enough Command Line to Be Dangerous*, I promised to show how to customize the prompt in *Learn Enough Text Editor to Be Dangerous*. Fulfill this promise by editing the `.bash_profile` file to include the lines shown in Listing 2.6. Source the Bash profile as in Listing 1.5 and confirm that the prompt on your system matches the one shown in Listing 2.5. (To learn how to customize the prompt using Z shell, the current default shell on macOS, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".)

Listing 2.4: The default terminal prompt on my system.

```
MacBook-Air:~ mhartl$
```

Listing 2.5: My preferred, more compact prompt.

```
[~]$
```

Listing 2.6: The Bash lines needed to customize the prompt as shown in Listing 2.5.
~/.bash_profile

```
alias lr='ls -hartl'
# Customize prompt to show only working directory.
PS1='[\W]\$ '
```

Figure 2.32: Finding using the menu.

2.8 Finding and replacing

One of the most powerful features of every good text editor is the ability to *find* and optionally *replace* text. In this section we'll learn how to find and replace in a single file; in Section 3.4 we'll discuss the more powerful (and *much* more dangerous) method of finding and replacing across multiple files.

To find inside a file, you can use the Find menu, shown in Figure 2.32, which also shows that ⌘F is the corresponding keyboard shortcut. Either one brings up a *modal window* where you can type in the string you're searching for (Figure 2.33).

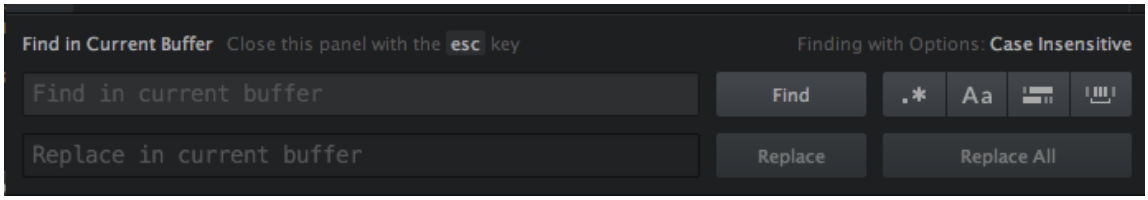

Figure 2.33: A modal window for finding and replacing.

For example, suppose we search for the string "sample". As seen in Figure 2.34, both "Sample" and "sample" are highlighted. The reason our search finds both is because we've opted to search case-insensitively (which is usually the default).

Figure 2.35 shows how to use the modal window to find "sample" and replace with "example". In order to avoid replacing "Sample", we first click on Find to select the next match, and then click on Replace to replace the second match (Figure 2.36). (Changing to case-sensitive search would also work in this case; learning how to do this is left as an exercise (Section 2.8.1).)

As seen in Figure 2.32, you can also type ⌘G to find the next match using a keyboard shortcut. This ⌘F/⌘G combination also works in many other applications, such as word processors and web browsers.

2.8.1 Exercises

1. In Section 2.3.1, we found Sonnet 18 by going directly to line 293, but of course I didn't search the file line by line to write the exercise. Instead, I searched for "shall I compare thee". Use your text editor to search for this string in **`sonnets.txt`**. On what line does "rosy lips and cheeks" appear?

2. The example in this section shows one of the pitfalls of mechanically finding and replacing text: we've ended up with the ungrammatical result "a example" instead of "an example". Rather than fix this by hand, use find and replace to replace "a example" with "an example" in your document. (Although in the present case there's only one occurrence, this

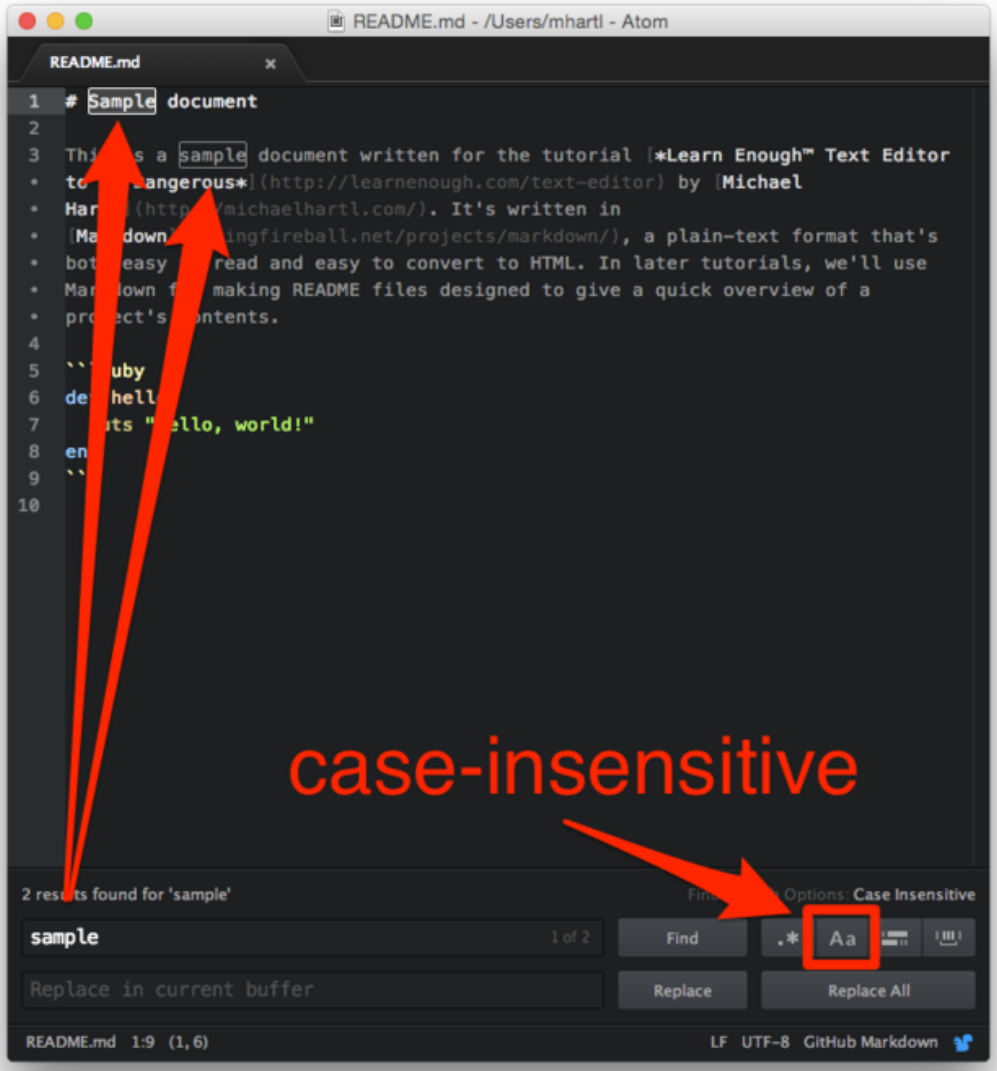

Figure 2.34: Finding the string “sample”.

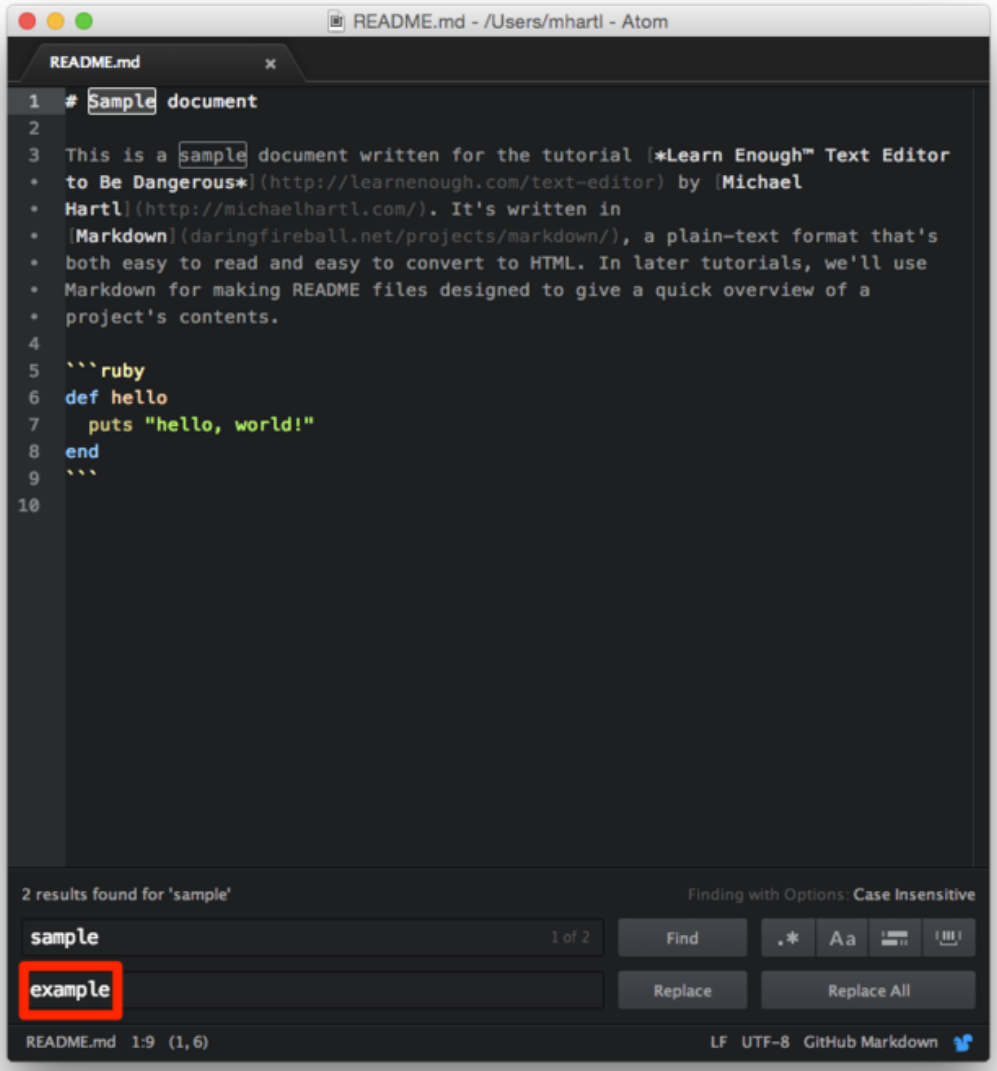

Figure 2.35: Finding and replacing.

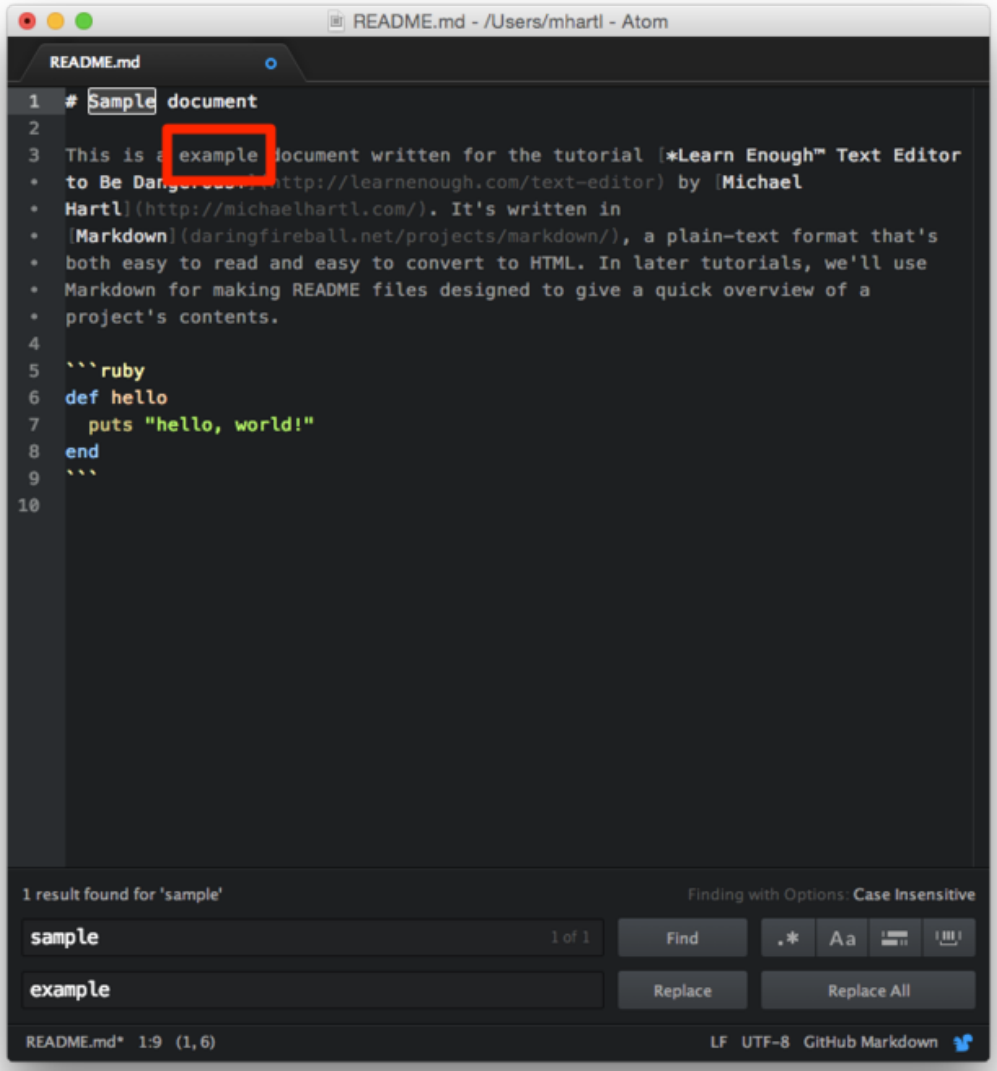

Figure 2.36: The result of replacing “sample” with “example”.

more general technique scales up to documents much longer than our toy example.)

3. What is the keyboard shortcut in your editor for finding the previous match?

4. What is the keyboard shortcut to replace in the current buffer (file)? How does this differ from the keyboard shortcut for simply finding?

2.9 Summary

- Atom and Sublime Text are both excellent choices for a primary modern text editor.
- One common way to open files is to use a command at the command line.
- For files containing things like prose with long lines, it's a good idea to turn on word wrap.
- Moving around text files can be accomplished many different ways, including using the mouse and arrow keys (especially in combination with the Command/Control key).
- One convenient way to select text is to hold down Shift and move the cursor.
- The Cut/Copy/Paste triumvirate is incredibly useful.
- Undo can save your bacon (Figure 2.37).[3]

Important commands from this section are summarized in Table 2.2.

[3] Image retrieved from https://www.flickr.com/photos/cookbookman/6175755733 on 2015-11-12 and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 2.37: Undo can save your bacon.

Command	Description
⌘←	Move to beginning of line (stops on whitespace)
⌘→	Move to end of line
⌘↑	Move to beginning of file
⌘↓	Move to end of file
⇧-move	Select text
⌘D	Select current word
⌘A	Select All (entire document)
⌘X/⌘C/⌘V	Cut/Copy/Paste
⌘Z	Undo
⇧⌘Z or ⌘Y	Redo
⌘S	Save
⌘F	Find
⌘G	Find next

Table 2.2: Important commands from Chapter 2.

Chapter 3

Advanced text editing

Having covered the basic functions of modern text editors in Chapter 2, in this section we'll learn about a few of the most common advanced topics. Even more than in Chapter 2, details will vary based on the exact editor you choose, so use your growing technical sophistication (Box 1.3) to figure out any necessary details. The most important lesson is that the advanced functions in this section are all things that *any* professional-grade editor can do, so you should be able to figure out how to do them no matter which editor you're using.

3.1 Autocomplete and tab triggers

Two of the most useful features of text editors are *autocomplete* and *tab triggers*, which you can think of as roughly command-line style tab completion for text files. (See *Learn Enough Command Line to Be Dangerous* for details on tab completion.) Both features allow us to type potentially large amounts of text with only a few keystrokes.

3.1.1 Autocomplete

The most common variant of autocomplete lets us type the first few letters of a word and then gives us the ability to complete it from a menu of options, typically by using the arrow keys and hitting Tab to accept the completion. An

example of autocompleting the word "Markdown" in **README.md** appears in Figure 3.1.

The autocomplete menu itself is populated using the current document, so autocomplete is particularly useful in longer documents that contain a large number of possible completions. For instance, the source for *Learn Enough Text Editor to Be Dangerous* (which is written using the powerful markup language LaTeX) makes use of a large number of labels for making cross-references, and these labels are often long enough that it's much easier to autocomplete them than to type them out by hand. An example is the oft-cited Box 1.3, whose source looks like Listing 3.1.

Listing 3.1: A cross-reference with a label I usually autocomplete.

```
Box~\ref{aside:technical_sophistication}
```

When writing a string like **technical_sophistication** in Listing 3.1, I nearly always use autocomplete instead of typing it out in full.[1] (As mentioned below, the rest of the cross-reference is generated using a custom tab trigger.) Similar considerations frequently occur when writing source code, where (as we'll learn in *Learn Enough Ruby to Be Dangerous*) we might encounter something like this:

```
ReallyLongClassName < ReallyLongBaseClassName
```

In such cases, rather than typing out the long names by hand, it's usually easier to type **Rea** and then select the relevant autocompletion.

3.1.2 Tab triggers

Tab triggers are similar to autocompletion in that they let us type a few letters and then hit *tab* to work some magic, but in this case many of them come predefined with the editor, with the exact triggers typically based on the particular

[1] I actually have my Sublime Text editor configured to use the **ESC** key for autocompletion instead of using a menu, mainly because I got used to that design when using my previous editor (TextMate). I arranged for this setup using my technical sophistication (Box 1.3).

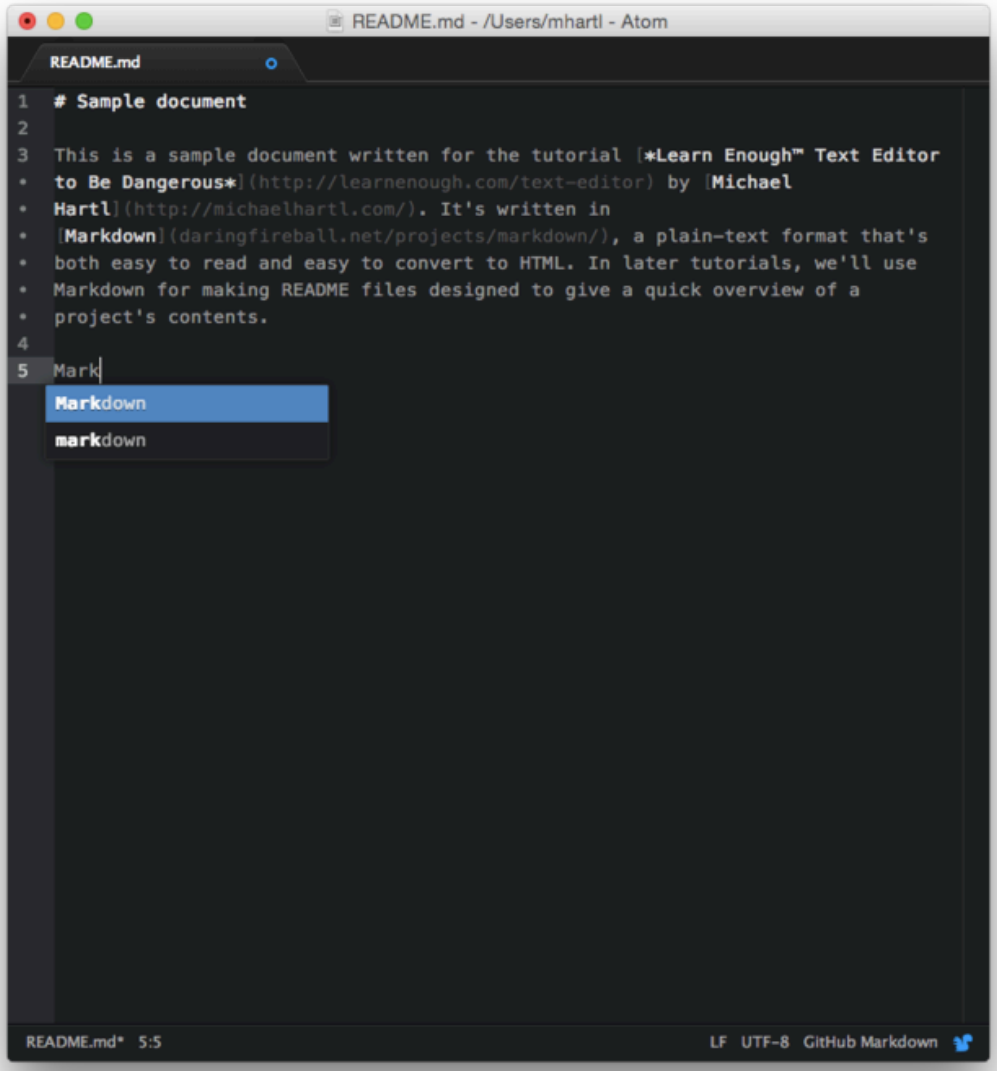

Figure 3.1: Autocomplete for “Markdown”.

type of document we're editing. For example, in Markdown and other markup files (HTML, LaTeX, etc.), typing `lorem`⇥ or `lo`⇥ yields so-called *lorem ipusum* text, a slightly corrupted Latin fragment from a book by Cicero that is often used as dummy text in programming and design. We saw *lorem ipsum* briefly before in Listing 2.1; a second example appears in Figure 3.2, which shows the result of typing `lo` in Atom. A closeup appears in Figure 3.3. After hitting ⇥ to invoke the tab trigger, the full *lorem ipsum* text appears as in Figure 3.4.

Tab triggers are especially useful when editing more syntax-heavy files like HTML and source code. For instance, when writing HTML, many editors support the creation of an HTML skeleton using the trigger `html`⇥, together with HTML *tags* (covered in *Learn Enough HTML to Be Dangerous*) using the tag name with a tab, such as `h1`⇥ for an **`h1`** or top-level heading tag. In Atom, we can do something like this:

```
$ atom index.html
```

The result of applying the various tab triggers then might look something like Figure 3.5.
Because HTML, or HyperText Markup Language, is the language of the World Wide Web, navigating to the file in a browser then shows a simple but real web page (Figure 3.6).

Similarly, when writing Ruby code, typing `def`⇥ in Atom creates a Ruby *define* statement to make a *function*, which looks like this:

```
def method_name

end
```

After typing the name of the function (which replaces the placeholder text **`method_name`**), we can hit ⇥ again to place the cursor in the right location to start writing the main part of the function. These sorts of auto-expansions of content can speed up code production considerably, while also lowering the

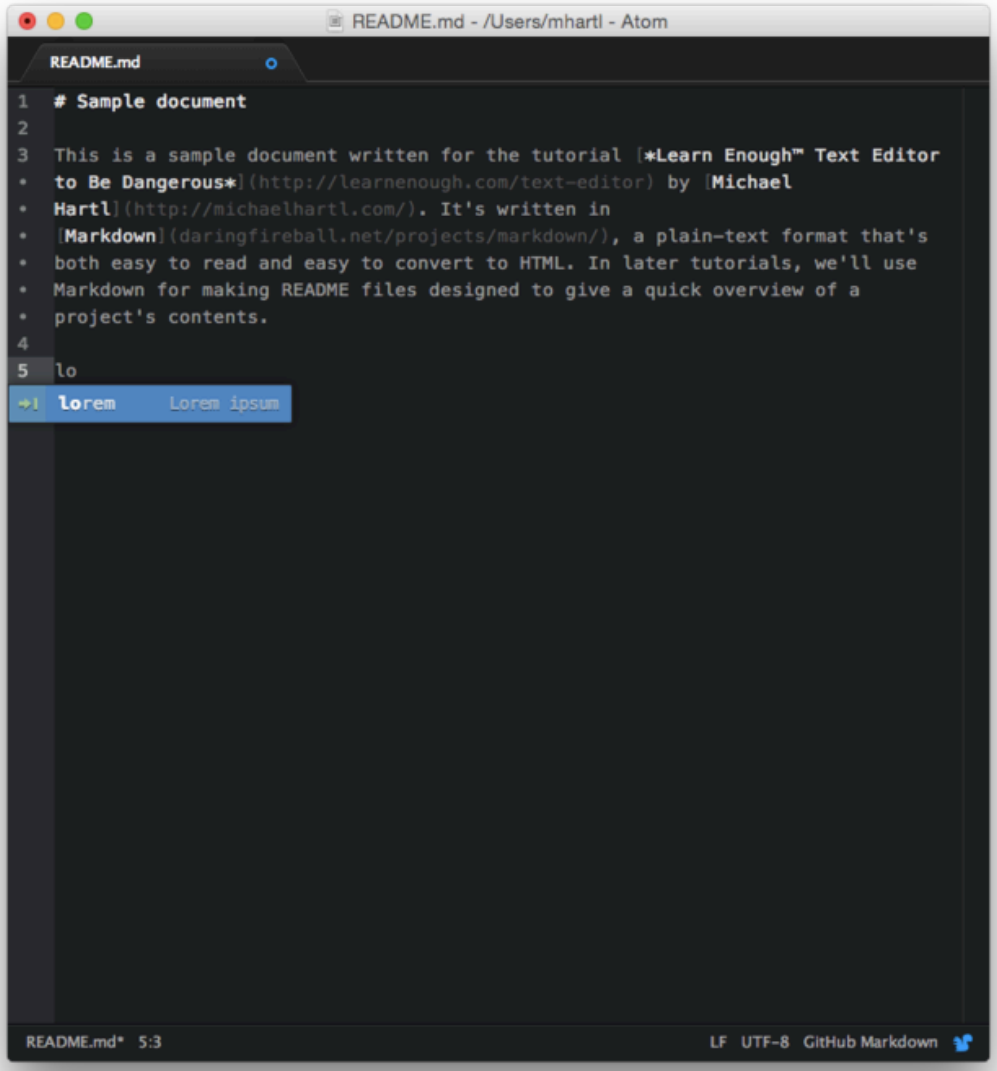

Figure 3.2: Typing “lo” in Atom prepares to activate a tab trigger.

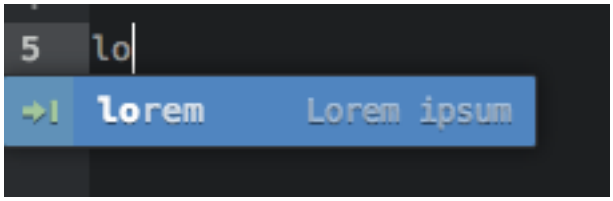

Figure 3.3: A more detailed view of the trigger in Figure 3.2.

cognitive load of programming. We'll see a concrete example of this technique in Section 3.2.

Finally, it's possible to define tab triggers of your own. My own editing makes extensive use of tab triggers; for example, to make the text in Listing 3.1, instead of typing

```
Box~\ref{aside:technical_sophistication}
```

by hand I used the custom tab trigger **`bref`** (for "box reference") to generate

```
Box~\ref{aside:}
```

and then filled in the label **`technical_sophistication`** using autocomplete (Section 3.1.1). Defining custom tab triggers is highly editor-dependent and is beyond the scope of this tutorial, but some hints about how to figure it out for yourself appear in Section 3.5.

3.1.3 Exercises

1. Add some more *lorem ipsum* text to **`README.md`** using a tab trigger.
2. Add another occurrence of the word "consectetur" using autocomplete.
3. Write the sentence "As Cicero once said, 'quis nostrud exercitation ullamco laboris'." with the help of as many uses of autocomplete as you want.

Figure 3.4: The result of the tab trigger in Figure 3.2.

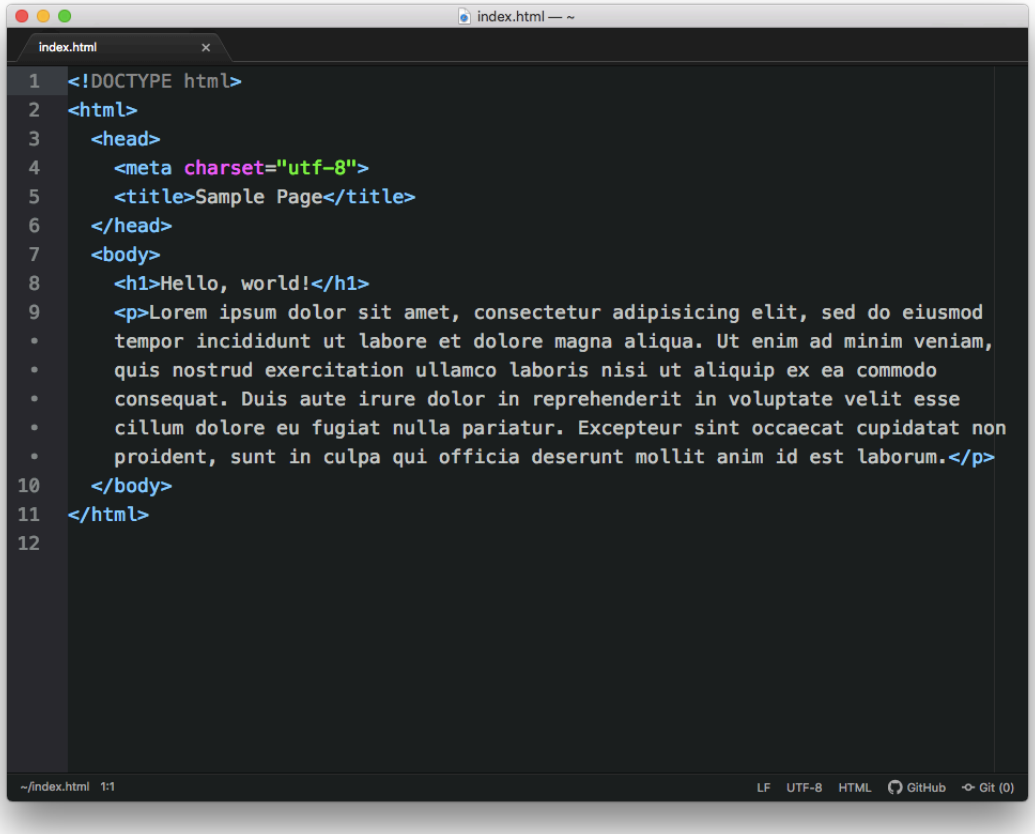

Figure 3.5: The result of applying HTML tab triggers.

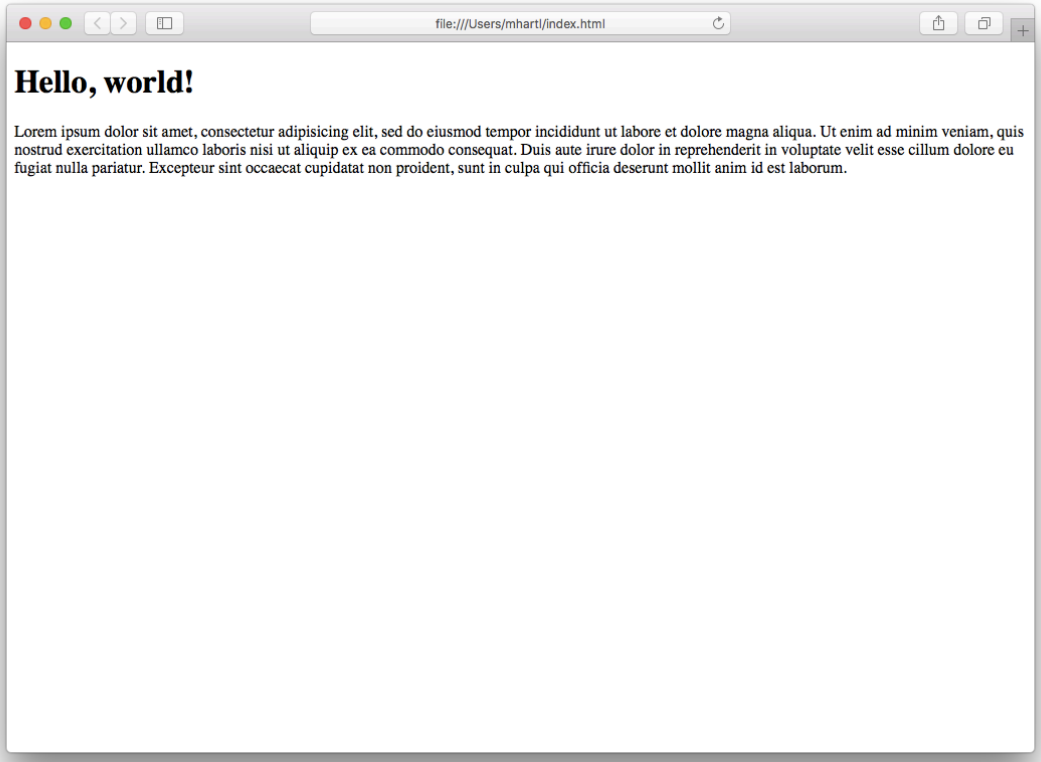

Figure 3.6: The result of applying tab triggers to an HTML page.

3.2 Writing source code

As hinted at in Section 3.1.2, in addition to being good at editing markup like HTML and Markdown, text editors excel at writing computer programs. Any good programmer's text editor supports many specialized functions for writing code; this section covers a few of the most useful. Even if you don't know how to program (yet!), it's still useful to know about some of the ways text editors support writing code.

An example of computer code appears in Listing 3.2, which shows a variant of a "hello, world" program written in the Ruby programming language. (You are not expected to understand this program.)

Listing 3.2: A variant of "hello, world" in Ruby.

```
# Prints a greeting.
def hello(location)
  puts "hello, #{location}!"
end

hello("world")
```

To see the contents from Listing 3.2 in a text editor, we can fire up Atom as follows:

```
$ atom hello.rb
```

Upon pasting in the content of Listing 3.2, we get the result shown in Figure 3.7. (For extra credit, type in Listing 3.2 by hand using the **`def`** tab trigger discussed in Section 3.1.)

3.2.1 Syntax highlighting

As we saw in Section 2.2.1 with **`README.md`**, Atom uses the filename extension to determine the proper syntax highlighting. In that case the (rather subtle) highlighting was for Markdown; in this case, Atom infers from **`.rb`** that the

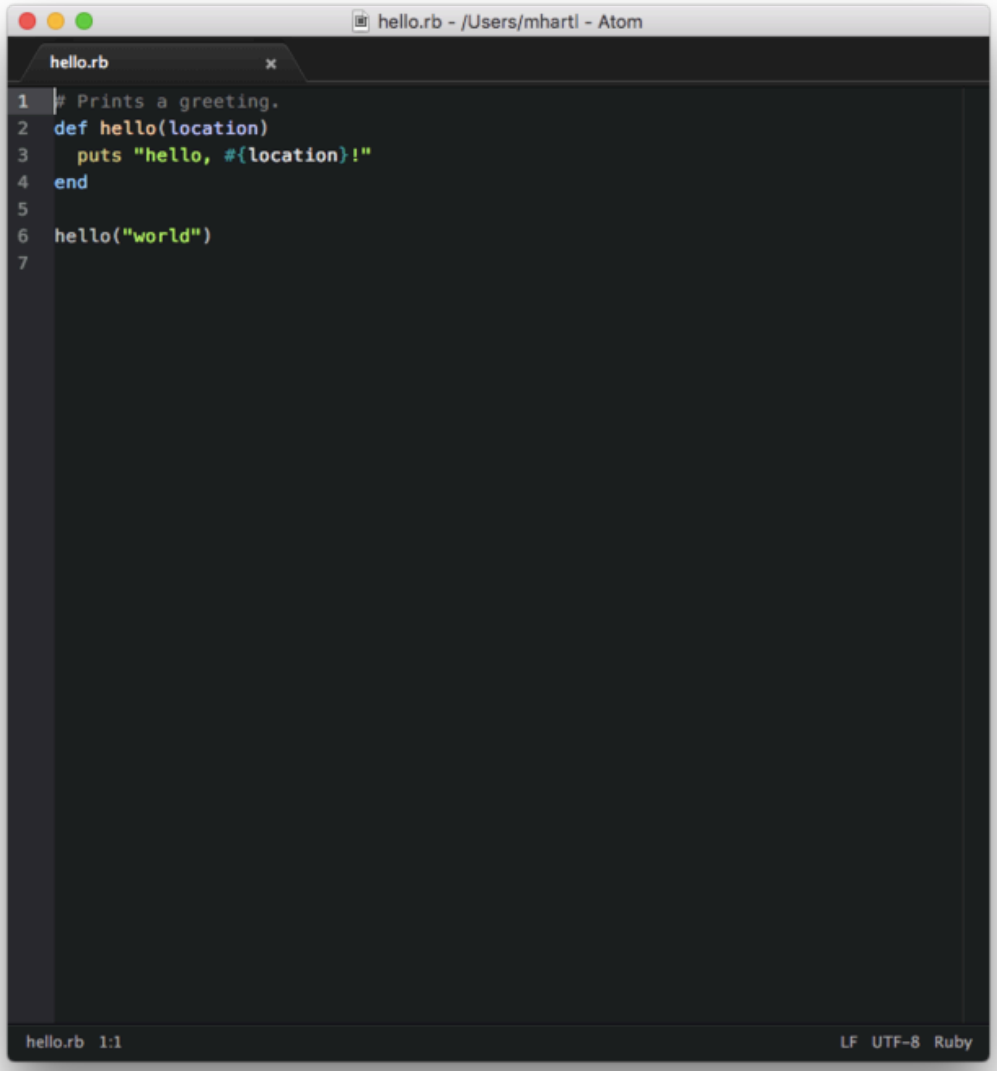

Figure 3.7: A Ruby program in Atom.

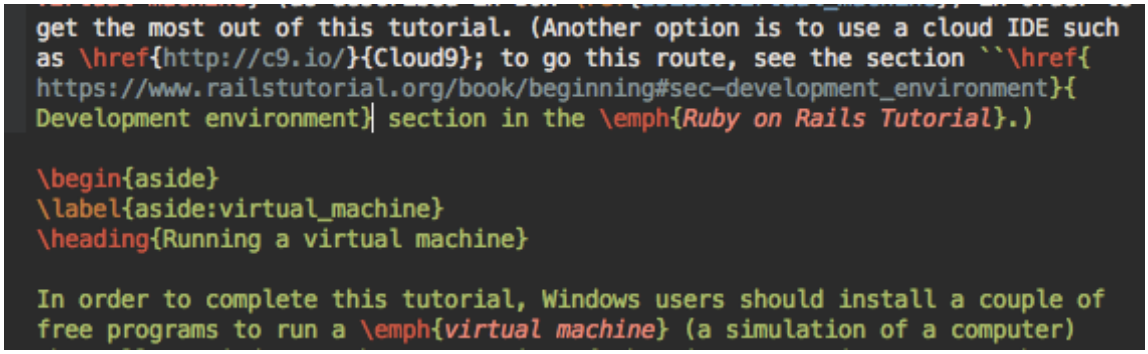

Figure 3.8: An error in LaTeX source caught by syntax highlighting.

file contains Ruby code, and highlights it accordingly. As before, it's essential to understand that the highlighting isn't inherent to the text, which is still plain. Syntax highlighting is purely for our benefit as readers of the code.

In addition to making it easier to parse the source code visually (e.g., distinguishing keywords, strings, constants, etc.), syntax highlighting can also be useful for catching bugs. For example, at one point when editing *Learn Enough Command Line to Be Dangerous* I accidentally deleted a LaTeX closing quote (which consists of the two single quotes `''`), with the result shown in Figure 3.8. This changed the color of the main text from the default white to the color used for quoted strings (green), which made it apparent at a glance that something was wrong. Upon fixing the error, the highlighting changed back to the expected white text, as shown in Figure 3.9.

3.2.2 Commenting out

One of the most useful functions of a text editor is the ability to "comment out" blocks of code, a technique often used to temporarily prevent execution of certain lines without having to delete them entirely (which is often particularly helpful when debugging). Most programming and markup languages support comment lines that exist for the benefit of humans reading the code but are

```
get the most out of this tutorial. (Another option is to use a cloud IDE such
as \href{http://c9.io/}{Cloud9}; to go this route, see the section ``\href{
https://www.railstutorial.org/book/beginning#sec-development_environment}{
Development environment}'' section in the \emph{Ruby on Rails Tutorial}.)

\begin{aside}
\label{aside:virtual_machine}
\heading{Running a virtual machine}

In order to complete this tutorial, Windows users should install a couple of
free programs to run a \emph{virtual machine} (a simulation of a computer)
```

Figure 3.9: Error fixed, syntax highlighting as expected.

ignored by the programming language itself.[2] An example of a Ruby comment appears in the first line of Listing 3.2:

```
# Prints a greeting.
```

Here the leading hash symbol `#` is Ruby's way of indicating a comment line.

Suppose we wanted to comment out the next three lines (lines 2–4), to change

```
# Prints a greeting.
def hello(location)
  puts "hello, #{location}!"
end

hello("world")
```

to

```
# Prints a greeting.
# def hello(location)
#   puts "hello, #{location}!"
# end

hello("world")
```

[2] Technically, comments are ignored by the compiler or interpreter. Some languages have automated documentation systems that do process the comments.

It's possible to do this by hand, of course, simply by inserting a # at the beginning of each line. This is inconvenient, though, and becomes increasingly so as the length of the commented-out text grows. Instead, we can select the desired text (Section 2.4) and use a menu item or keyboard shortcut to comment out the selection. In Atom, we can comment out lines 2–3 by selecting those lines (Figure 3.10) and hitting ⌘/, as shown in Figure 3.11. (Note from Figure 3.11 that the subtle save indicator shown in Figure 2.31 has been filled in; this is because I habitually press ⌘S after making changes, as recommended in Section 2.7.)

The commenting-out feature typically toggles back and forth, so by hitting ⌘/ a second time we can restore the file to its previous state (Figure 3.10). This is useful when restoring some commented-out text after, for example, doing some debugging.

3.2.3 Indenting and dedenting

Another element of code formatting made easier by text editors is *indentation*, which consists of the leading spaces at the beginning of certain lines. It used to be common to use *tab* characters for indentation, but unfortunately the number of spaces *displayed* for a tab is system-dependent, leading to unpredictable results: some people might see four "spaces" per tab, some might see eight, and some might see only two.

In recent years, many programmers have switched to *emulated tabs*, where pressing the tab key inserts a standard number of ordinary spaces (typically two or four). True tabs still have some partisans, though, and tabs vs. spaces remains holy war territory (Box 1.6). (Luckily, there is one thing everyone agrees on, which is that *mixing* tabs and spaces is a bad idea.)

To see how this works, we can take a look at some Ruby code, which typically uses two spaces for indentation:

```
def hello(location)
  puts "hello, #{location}!"
end
```

This would typically be achieved by hitting return after "(location)" and then pressing the tab key, although pressing the spacebar twice would also work.

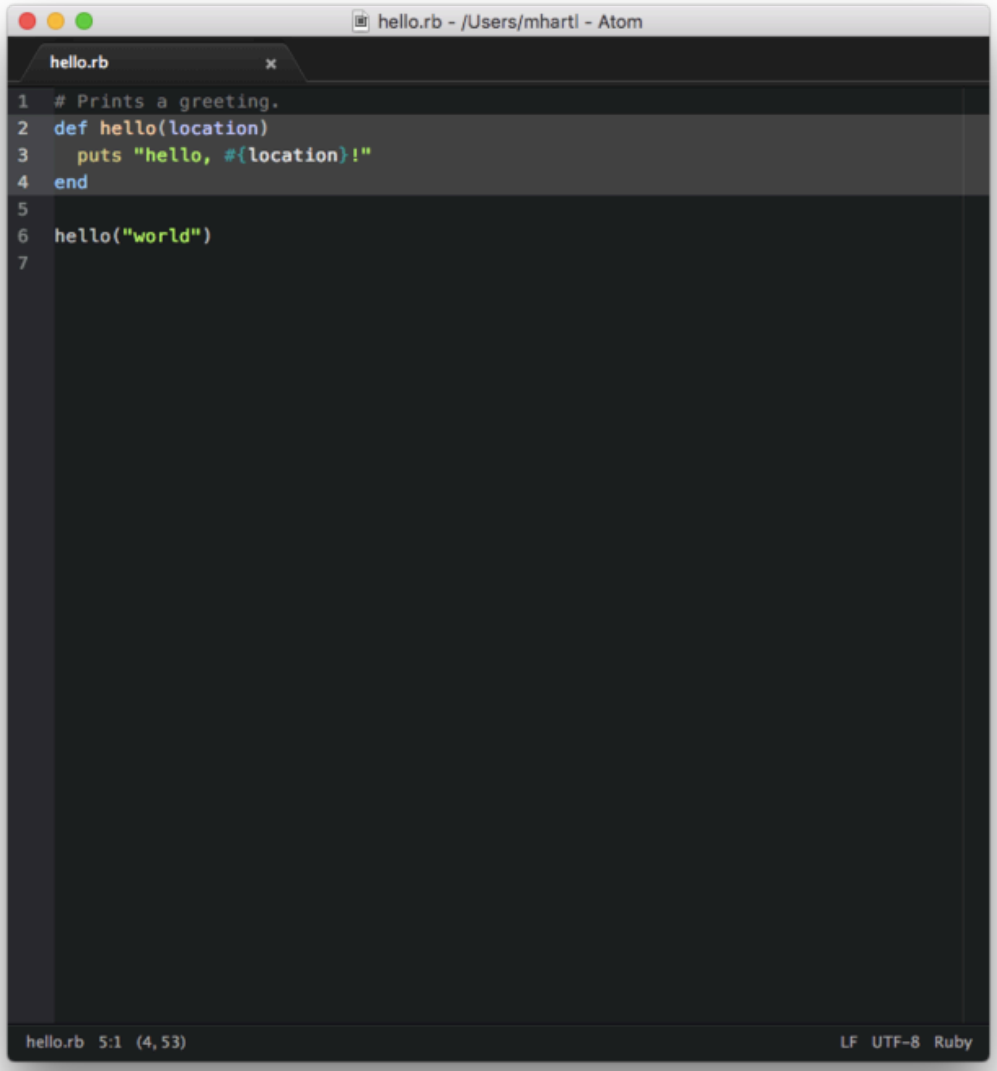

Figure 3.10: Preparing to comment out some lines.

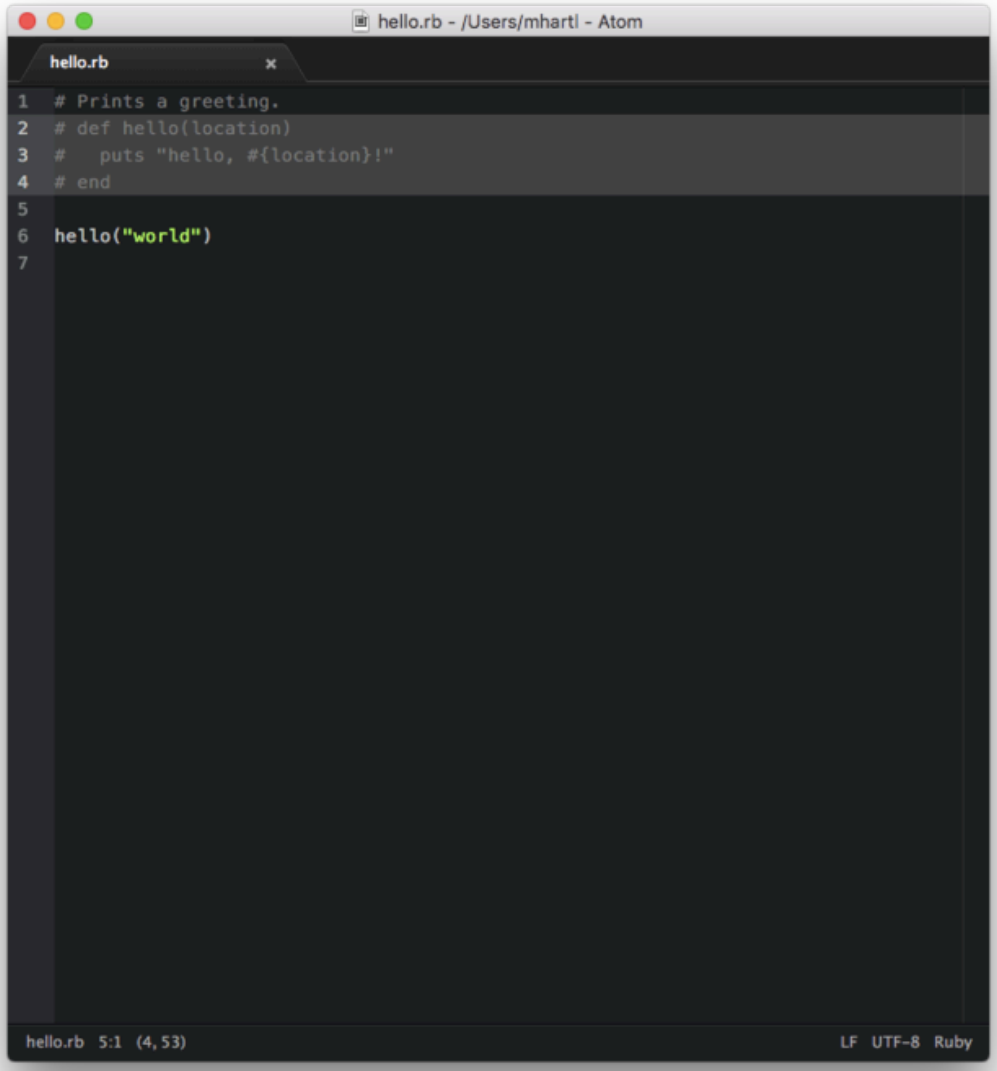

Figure 3.11: Commented-out lines.

Assuming that the editor has been configured to use two spaces to emulate tabs, we'd get the result shown above. In most languages, this would be equivalent to the following:[3]

```
def hello(location)
puts "hello, #{location}!"
end
```

This second example is harder to read, though, and it's important to indent properly for the sake of humans reading the code, even if the programming language doesn't care.[4]

Text editors help maintain proper indentation in two main ways. First, new lines are typically inserted at the same level of indentation as the previous line, which you can verify by going to the end of line 3 in Listing 3.2 and typing in the following two lines:

```
puts "Uh, oh."
puts "Goodbye, #{location}!"
```

The result appears in Figure 3.12.

The second main way text editors help maintain good code formatting is by supporting block indentation, which works in much the same way as commenting out code blocks. Suppose, for example, that (contrary to conventional Ruby practices) we decided to indent lines 3–5 in Figure 3.12 six extra spaces, making eight spaces total. As with commenting out, the first step is to select the text we want to indent (Figure 3.13). We can then type the tab key ⇥ to indent one "soft tab" (which is usually two spaces for Ruby) at a time. (If for any reason the default indentation in your editor doesn't match the convention for the language you're using, apply your technical sophistication (Box 1.3) to figure out how to change it.) The result of applying three tabs in succession is shown in Figure 3.14.

[3] Python is a notable exception.

[4] Some languages, notably Python, actually enforce some measure of proper indentation, but most language compilers and interpreters ignore it.

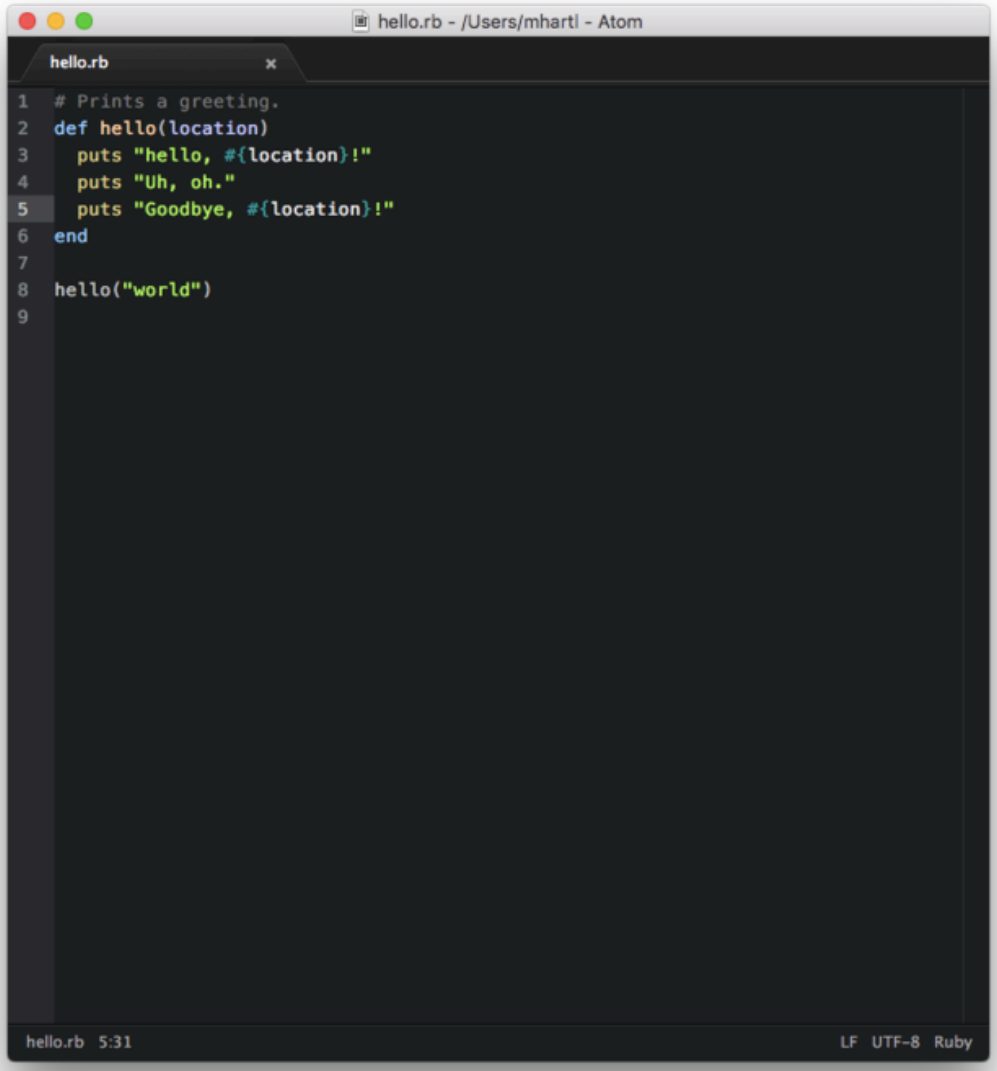

Figure 3.12: Adding two indented lines.

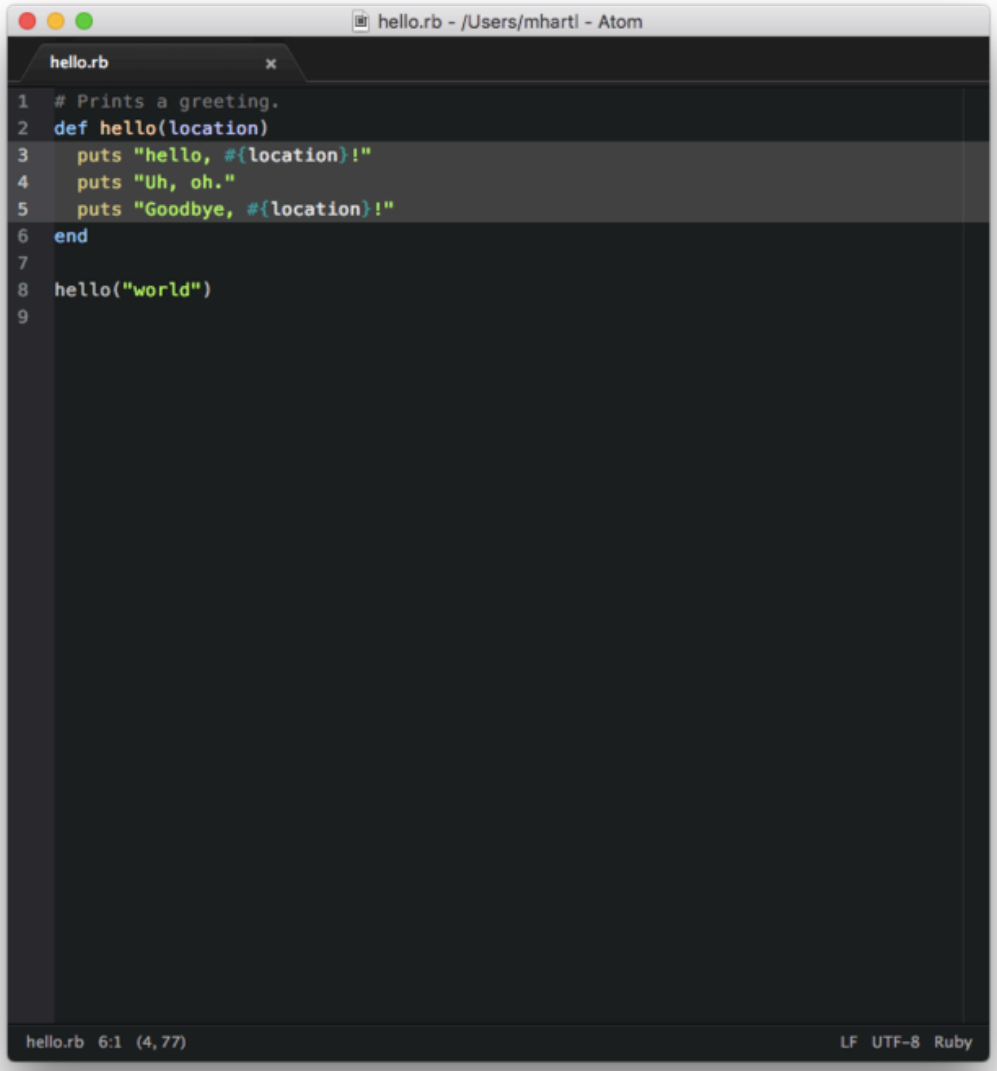

Figure 3.13: Preparing to indent some lines.

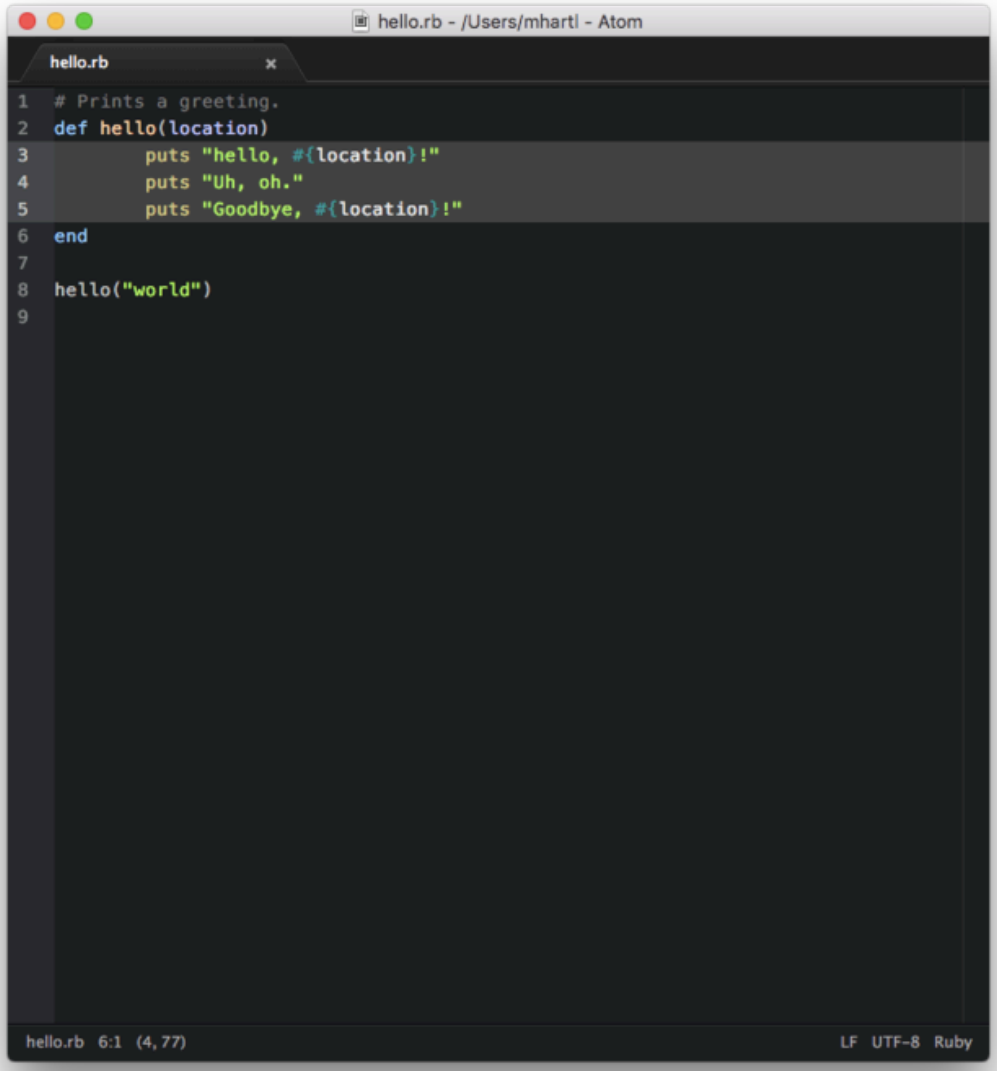

Figure 3.14: A block of Ruby code indented more than usual.

Because each extra tab just indents the block more, we can't use the same command to undo indentation the way we did when commenting out code. Instead, we need to use a separate "dedent" command, which in Atom is ⇧⇥. Applying this command three times in succession returns us to our original state, as shown in Figure 3.15. (By the way, many editors (including Atom) support the alternate keyboard shortcuts ⌘] and ⌘[for indenting and dedenting, respectively.)

3.2.4 Goto line number

It's often important to be able to go to a particular line number, such as when debugging a program that has an error on (say) line 187. We saw this feature in Section 1.6, where we learned that the Vim command **`<n>G`** takes us to line **`<n>`**. In many other editors, the relevant shortcut is ^G. This opens a modal box where you can type in the line number, as shown in Figure 3.16. (Incidentally, the syntax **`:<number>`** shown in Figure 3.16, which is for Sublime Text, also works in Vim.)

3.2.5 80 columns

Finally, many text editors help programmers enforce a line limit of 80 characters across, usually called an "80-column limit". Not all programmers observe this limit, but keeping our code to 80 columns makes it easier to read and display, for example in fixed-width terminals, blog posts, or tutorials such as this one.[5] An 80-column limit also enforces good coding discipline, as exceeding 80 columns is often a sign that we would do well to introduce a new variable or function name.[6] Because it's difficult to tell at a glance if a particular line exceeds 80 characters, many editors (including Atom and Sublime Text) include the option to display a subtle vertical line showing where the limit is, as shown in Figure 3.17.[7] If the 80-column limit indicator isn't shown by default in your

[5] The original source of this constraint is actually IBM punch cards.

[6] The main exception to the 80-column rule is markup like HTML, Markdown, or LaTeX, which is why in these cases we often activate word wrap as in Section 2.2.

[7] In some editors the line is at something like 78 columns instead of 80 to allow a small margin for error.

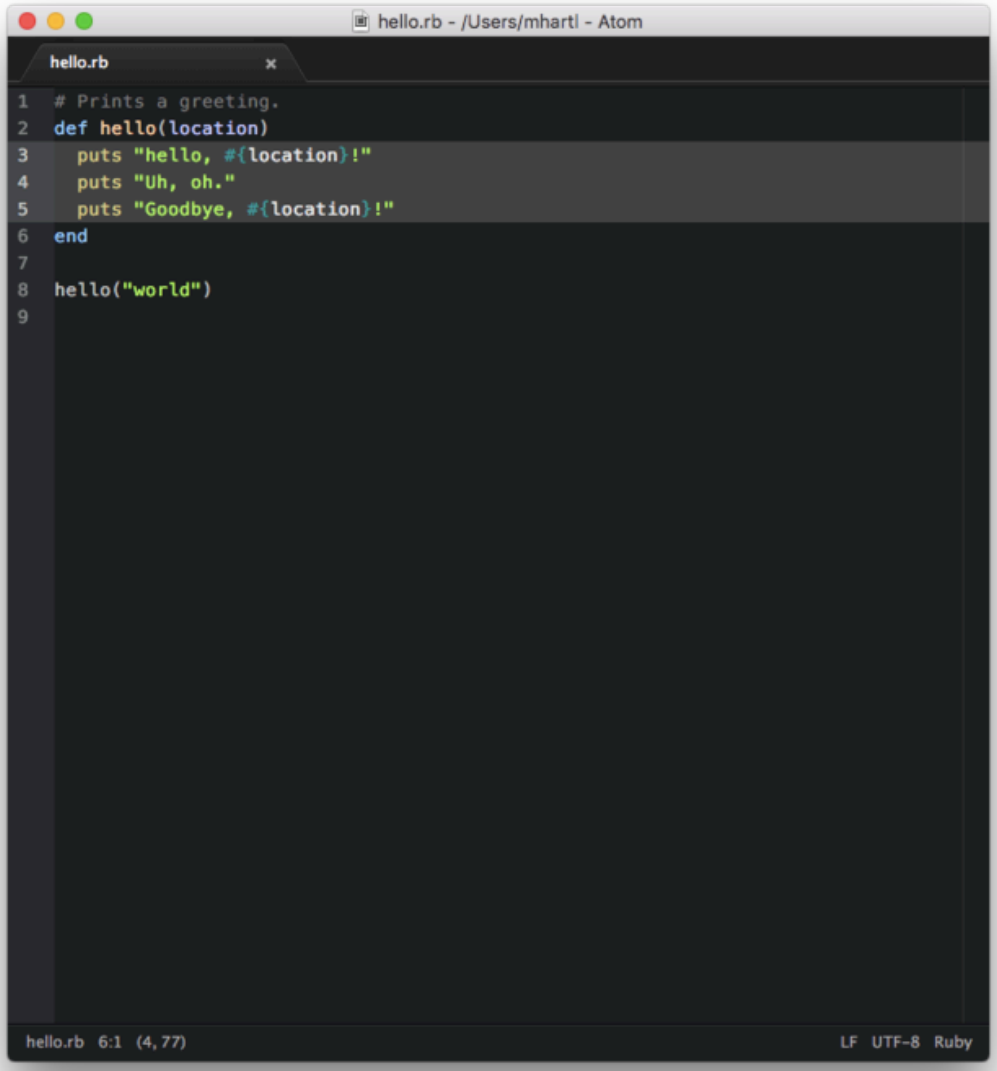

Figure 3.15: Dedenting the code block in Figure 3.14.

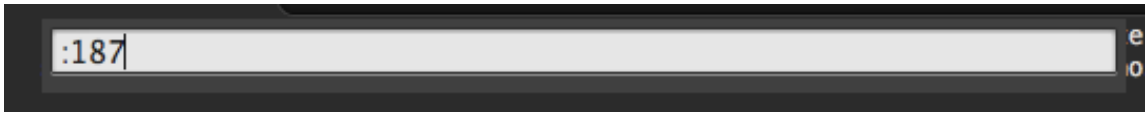

Figure 3.16: The modal box for going to a particular line number.

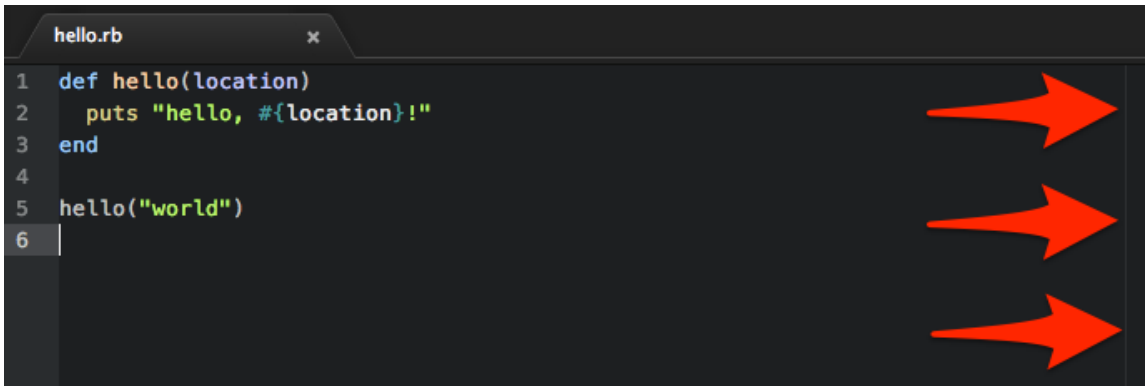

Figure 3.17: Unsubtle arrows pointing at the subtle 80-column indicator in Atom.

editor, flex your technical sophistication to figure out how to enable it. (It's often associated with a setting called "word wrap column".)

3.2.6 Exercises

1. Create the file **`foo.rb`**, then define the class **`FooBar`** (Listing 3.3) using a tab trigger. *Hint*: Chances are the trigger is something like `cla`⇥.

2. Referring to Listing 3.4, add the definition of **`bazquux`** using the `def`⇥ trigger, then add the final line shown by using autocomplete to type **`FooBar`** and **`bazquux`**. (Type the interstitial **`.new.`** by hand.)

3. Using tab triggers and autocomplete, make a file called **`greeter.rb`** with the contents shown in Listing 3.5.

4. By cutting and pasting the text for the **`hello`** definition and indenting the

block, transform Listing 3.5 into Listing 3.6.

Listing 3.3: Creating a class using a tab trigger.
~/foo.rb

```
class FooBar

end
```

Listing 3.4: Using autocomplete to make a class name.
~/foo.rb

```
class FooBar
  def bazquux
    puts "Baz quux!"
  end
end

FooBar.new.bazquux
```

Listing 3.5: A proto-Greeter class in Ruby.
~/greeter.rb

```
class Greeter
end

def hello(location)
  puts "hello, #{location}!"
end

Greeter.new.hello("world")
```

Listing 3.6: A completed Greeter class in Ruby.

```
class Greeter
  def hello(location)
    puts "hello, #{location}!"
  end
end

Greeter.new.hello("world")
```

3.3 Writing an executable script

As a practical application of the material in Section 3.2, in this section we're going to write something that's actually useful: a *shell script* designed to kill a program as safely as possible. (A *script* is a program that is typically used to automate common tasks, but the detailed definition isn't important at this stage.) En route, we'll cover the steps needed to add this script to our command-line shell.

As discussed in *Learn Enough Command Line to Be Dangerous*, Unix user and system tasks take place within a well-defined container called a *process*. Sometimes, one of these processes will get stuck or otherwise misbehave, in which case we might need to terminate it with the **`kill`** command, which sends a *terminate code* to kill the process with a given id:

```
$ kill -15 12241
```

(See the discussion in *Learn Enough Command Line to Be Dangerous* for more on how to find this id on your system.) Here we've used the terminate code **`15`**, which attempts to kill the process as gently as possible (meaning it gives the process a chance to clean up any temporary files, complete any necessary operations, etc.). Sometimes terminate code **`15`** isn't enough, though, and we need to escalate the level of urgency until the process is well and truly dead. It turns out that a good sequence of codes is **`15`**, **`2`**, **`1`**, and **`9`**. Our task is to write a command to implement this sequence, which we'll call **`ekill`** (for "escalating kill"), so that we can kill a process as shown in Listing 3.7.

Listing 3.7: An example of using **`ekill`** (to be defined).

```
$ ekill 12241
```

As with the Ruby example in Section 3.2, don't worry about the details of the code; focus instead on the mechanics of the text editing.

As preparation for adding **`ekill`** to our system, we'll first make a new directory in our home directory called **`bin`** (for "binary"):

```
$ mkdir ~/bin
```

(It's possible that this directory already exists on your system, in which case you'll get a harmless warning message.) We'll then change to the **bin** directory and open a new file called **ekill**:

```
$ cd ~/bin
$ atom ekill
```

The **ekill** script itself starts with a "shebang" line (pronounced "shuh-BANG", from "shell" and "bang", with the latter being the common pronunciation of the exclamation mark **!**):

```
#!/bin/bash
```

This line tells our system to use the shell program located in **/bin/bash** to execute the script. The **bash** program corresponds to the Bourne-again shell (Bash) mentioned in Section 1.3, and in this context a shell script is often called a *Bash script*.[8] Despite appearances, here the hash symbol **#** is *not* a comment character, which is potentially confusing because (as in Ruby) **#** is the character ordinarily used for a Bash comment line. Indeed, the initial version of our script includes several comment lines, as shown in Listing 3.8.

Listing 3.8: A custom escalating kill script.
~/bin/ekill

```
#!/bin/bash

# Kill a process as safely as possible.
# Tries to kill a process using a series of signals with escalating urgency.
# usage: ekill <pid>

# Assign the process id to the first argument.
pid=$1
kill -15 $pid || kill -2 $pid || kill -1 $pid || kill -9 $pid
```

[8] To learn how to write this same script using Zsh, see "Using Z Shell on Macs with the Learn Enough Tutorials".

Apart from the shebang in line 1, all other uses of `#` introduce comments. Then, Line 8 assigns the process id `pid` to `$1`, which in a shell script is the first argument to the command, e.g., `12241` in Listing 3.7. Line 9 then uses the "or" operator `||` to execute the `kill` command using the code `15` or `2` or `1` or `9`, stopping on the first successful `kill`. (Again, don't worry if you find this confusing; I include the explanation for completeness, but at this stage there's no need to understand the details.)

After typing the contents of Listing 3.8 into the script file, one thing you might notice is that the result has no syntax highlighting, as seen in Figure 3.18. This is because, unlike `README.md` (Section 2.2) and `hello.rb` (Section 3.2), the name `ekill` has no filename extension. Although some people would use a name like `ekill.sh` for shell scripts like this one—which would in fact allow our editor to highlight the syntax automatically—using an explicit extension on a shell script is a bad practice because the script's name is the user interface to the program. As users of the system, we don't care if `ekill` is written in Bash or Ruby or C, so calling it `ekill.sh` unnecessarily exposes the implementation language to the end-user. Indeed, if we wrote the first implementation in Bash but then decided to rewrite it in Ruby and then in C, every program (and programmer) using the script would have to change the name from `ekill.sh` to `ekill.rb` to `ekill.c`—an annoying and avoidable complication.

Even though we've elected not to use a filename extension for the `ekill` script, we'd still like to get syntax highlighting to work. One way is to click on "Plain Text" in the lower right-hand corner of the editor (Figure 3.18) and change the highlighting language to the one we're using. This requires us to *know* the language, though, and it would be nicer if we could get the editor to figure it out automatically. Happily, we can arrange exactly that, simply by closing the file and opening it again. To do this, click on the X to close the `ekill` tab (or press ⌘W) and then re-open it from the command line:

```
$ atom ekill
```

Because of the shebang line in Listing 3.8, Atom infers that the file is a Bash script. As a result, the detected file type changes from "Plain Text" to "Shell Script", and syntax highlighting is activated (Figure 3.19).

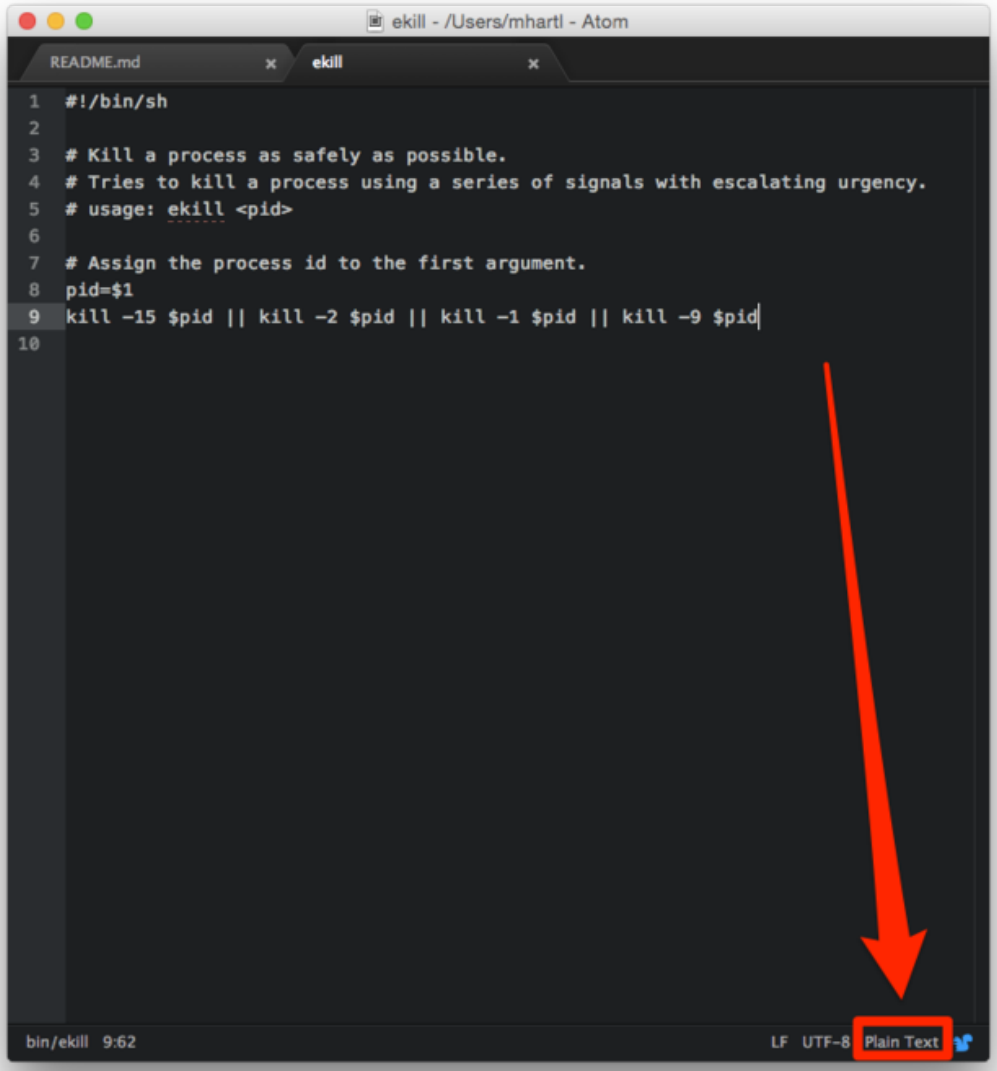

Figure 3.18: The `ekill` script with no syntax highlighting.

Figure 3.19: The `ekill` script with syntax highlighting and a new detected file type.

At this point, we have a complete shell script, but typing `ekill <pid>` at the command line still won't work. To add `ekill` to our system, we need to do two things:

1. Make sure the `~/bin` directory is on the system *path*, which is the set of directories where the shell program searches for *executable* scripts.
2. Make the script itself executable.

The list of directories on the path can be accessed via the special `$PATH` variable at the command line:

```
$ echo $PATH
```

If `~/bin` is on the list, you can skip this step, but it does no harm to follow it.

Note: The literal directory `~/bin` won't appear in the `$PATH` list; instead the tilde will be expanded to your particular home directory. For me, `~/bin` is the same as `/Users/mhartl/bin`, so that's what appears in my `PATH`, but it will be different for you.

To make sure `~/bin` is on the path, we'll edit the Bash profile file we saw in Section 1.3. Open `~/.bash_profile` as follows:

```
$ atom ~/.bash_profile
```

Then add the `export` line shown in Listing 3.9.

Listing 3.9: Adding `~/bin` to the path.
~/.bash_profile

```
alias lr='ls -hartl'
export PATH="~/bin:$PATH"
```

This uses the Bash `export` command to add `~/bin` to the current path. (It's worth noting that some systems use the *environment variable* `$HOME` in place

of `~`, but the two are synonyms. If for any reason `~` doesn't work for you, it's worth trying `$HOME` instead, as in `$HOME/bin:$PATH`.)

To use it, we need to use `source` as in Section 1.4:

```
$ source ~/.bash_profile
```

To make the resulting script executable, we need to use the "change mode" command `chmod` to add the "execute bit" `x` as follows:

```
$ chmod +x ~/bin/ekill
```

At this point, we can verify that the `ekill` script is ready to go using the `which` command:

```
$ which ekill
```

(This command is covered in *Learn Enough Command Line to Be Dangerous*.) The result should be the full path to `ekill`, which on my system looks like this:

```
$ which ekill
/Users/mhartl/bin/ekill
```

On some systems, running `source` on `.bash_profile` might not be sufficient to put `ekill` on the path, so if `which ekill` returns no result then you should try exiting and restarting the shell program to reload the settings.

As you can see by typing `ekill` by itself at the command line, the current behavior is confusing if we neglect to include a process id:

```
$ ekill
<confusing error message>
```

To make **ekill** friendlier in this case, we'll arrange to print a usage message to the screen if the user neglects to include a process id. We can do this with the code in Listing 3.10, which I recommend typing in rather than copying and pasting. When writing the **if** statement, I especially recommend trying `if`→| to see if your editor comes with a tab trigger for making Bash **if** statements.

Listing 3.10: An enhanced version of the escalating kill script.
~/bin/ekill

```
#!/bin/bash

# Kill a process as safely as possible.
# Tries to kill a process using a series of signals with escalating urgency.
# usage: ekill <pid>

# If the number of argument is less than 1, exit with a usage statement.
if [[ $# -lt 1 ]]; then
  echo "usage: ekill <pid>"
  exit 1
fi

# Assign the process id to the first argument.
pid=$1
kill -15 $pid || kill -2 $pid || kill -1 $pid || kill -9 $pid
```

After adding the code in Listing 3.10, running **ekill** without an argument should produce a helpful message:

```
$ ekill
usage: ekill <pid>
```

All we have left to do is to verify that **ekill** can actually be used to kill a process. This is left as an exercise (Section 3.3.1.)

3.3.1 Exercises

1. Let's test the functionality of **ekill** by making a process that hangs and applying the lessons from grepping processes in *Learn Enough Command Line to Be Dangerous*. We'll start by opening two terminal tabs. In one

tab, type `tail` to get a process that just hangs. In the other tab, use `ps aux | grep tail` to find the process id, then run `ekill <pid>` (substituting the actual id for `<pid>`). In the tab running `tail`, you should get something like "Terminated: 15" (Figure 3.20).

2. Write an executable script called `hello` that takes in an argument and prints out "Hello" followed by the argument. Be sure to `chmod` the script so it can run properly. *Hint*: Use the `echo` command. *Bigger hint*: Bash scripts *interpolate* dollar-sign variables into strings, so the `$1` variable from Listing 3.8 can be used in a string like this: `"Hello, $1"`

3.4 Editing projects

So far we've used our text editor to edit single files, but it can also be used to edit entire projects all at once. As an example of such a project, we'll download the sample application from the 3rd edition of the *Ruby on Rails Tutorial*. We won't be running this application, but it will give us a large project to work with. As in Section 1.6 and Section 2.2, we'll use the `curl` command to download the file to our local disk:

```
$ cd
$ curl -OL https://source.railstutorial.org/sample_app.zip
```

As indicated by the `.zip` filename extension, this is a ZIP file, so we'll unzip it (using the `unzip` command) and then `cd` into the sample app directory:

```
$ unzip sample_app.zip
   creating: sample_app_3rd_edition-master/
   .
   .
   .
$ cd sample_app_3rd_edition-master/
```

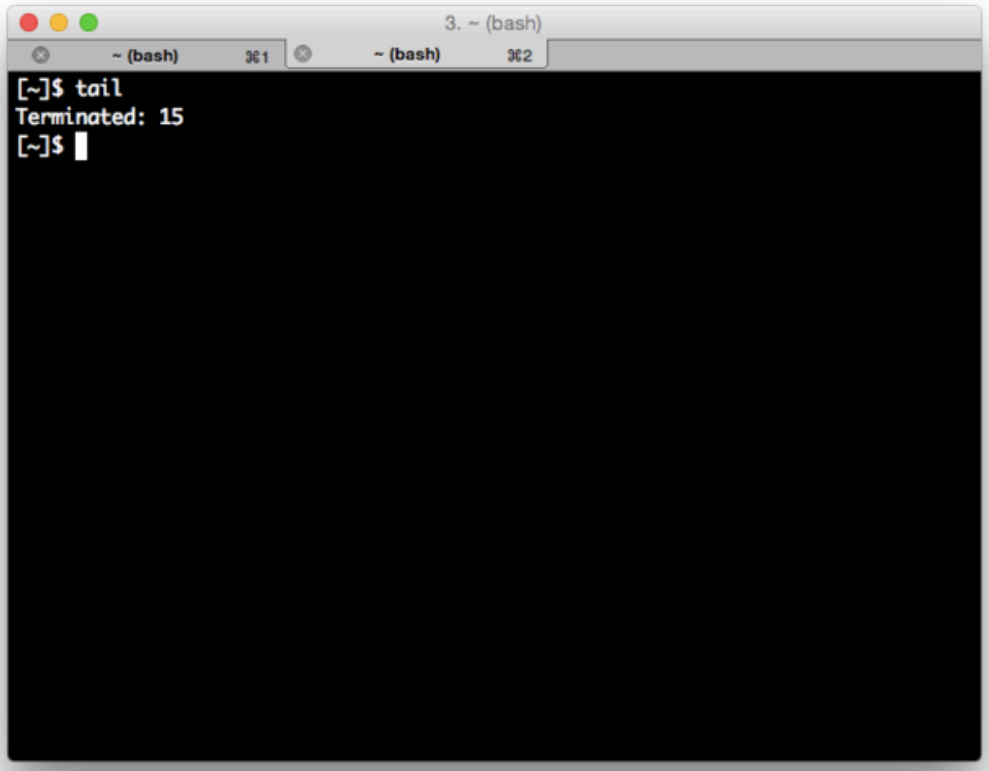

Figure 3.20: The result of using **ekill** to kill a **tail** process.

The way to open a project is to use a text editor to open the entire directory. Recall (from, e.g., navigating directories in *Learn Enough Command Line to Be Dangerous*) that `.` ("dot") is the current directory, which means that we can open it using "atom dot":

```
$ atom .
```

The resulting text editor window includes the directory structure for our project, called a "tree view", as seen in Figure 3.21. We can toggle its display using the View menu or a keyboard shortcut (Figure 3.22).

3.4.1 Fuzzy opening

It's possible to open a file by double-clicking on it in the tree view, but in a project with a lot of files this is often cumbersome, especially when the file is buried several subdirectories deep. A convenient alternative is *fuzzy opening*, which lets us open files by hitting (in Atom) ⌘P and then typing some of the letters in the filename we want. For example, we can open a file called `users_controller_test.rb` by typing, say, "userscon" and then selecting from the drop-down menu, as shown in Figure 3.23. The letters don't have to be contiguous in the filename, though, so typing "uctt" (for **u**sers **c**on**t**roller **t**est) will also work, as seen in Figure 3.24.

As a result of opening multiple files in a project, you will generally have multiple tabs open in your editor (Figure 3.25). I recommend learning the keyboard shortcuts to switch between them, which are typically things like ⌘1, ⌘2, etc. (By the way, this trick also works in many browsers, such as Chrome and Firefox.)

3.4.2 Multiple panes

The default editor view we've seen in most of the previous examples consists of a single *pane* (as in "window pane"), but it's often convenient to split the editor into multiple panes so that we can see more than one file at a time (Figure 3.26).

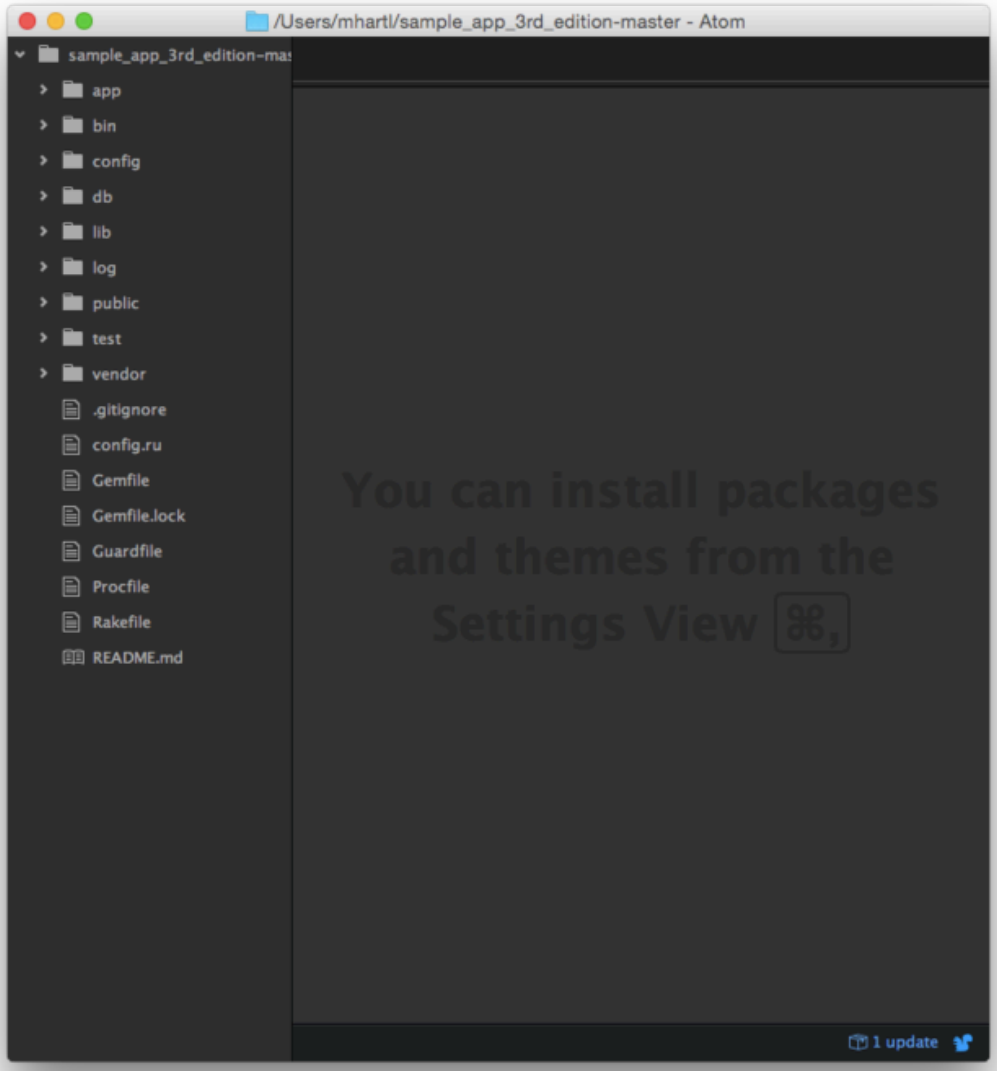

Figure 3.21: The Rails Tutorial sample app in Atom.

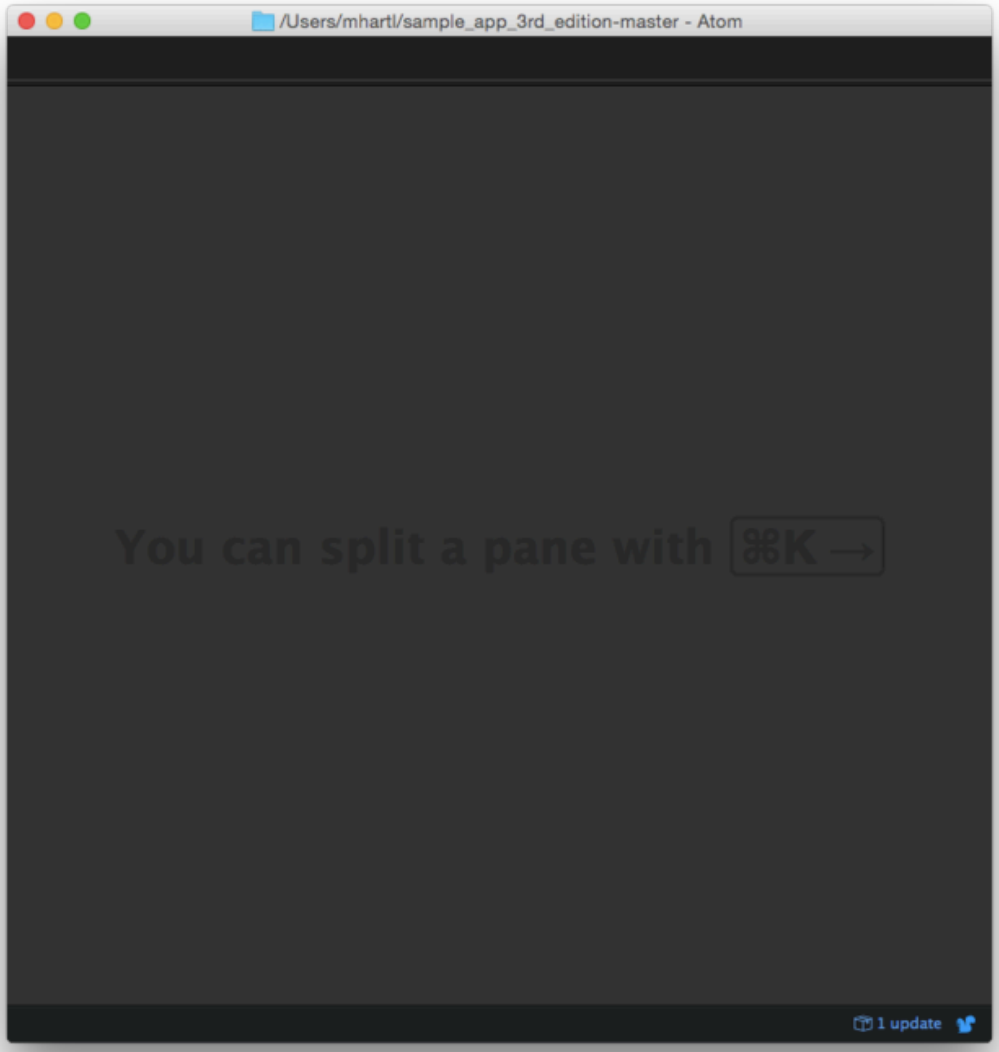

Figure 3.22: Toggling the tree view.

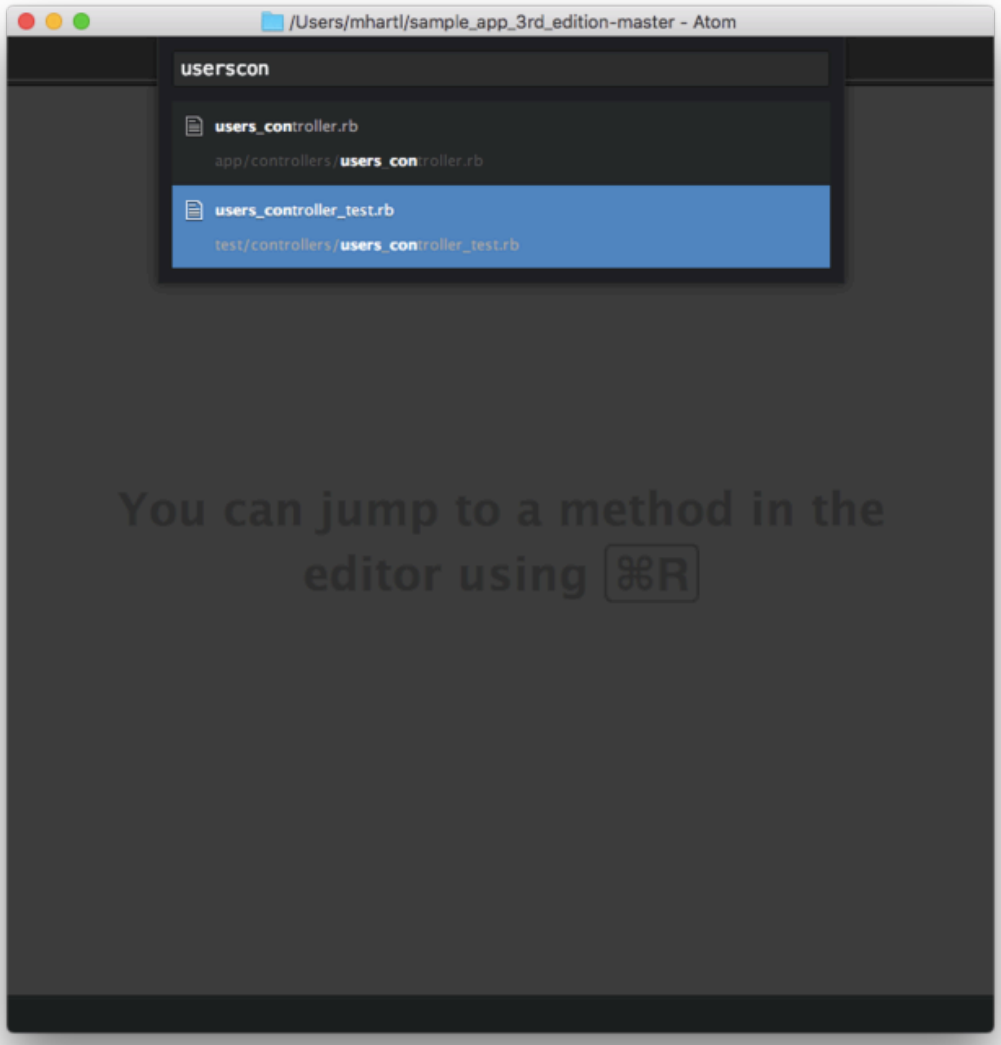

Figure 3.23: One way to open a file with fuzzy opening.

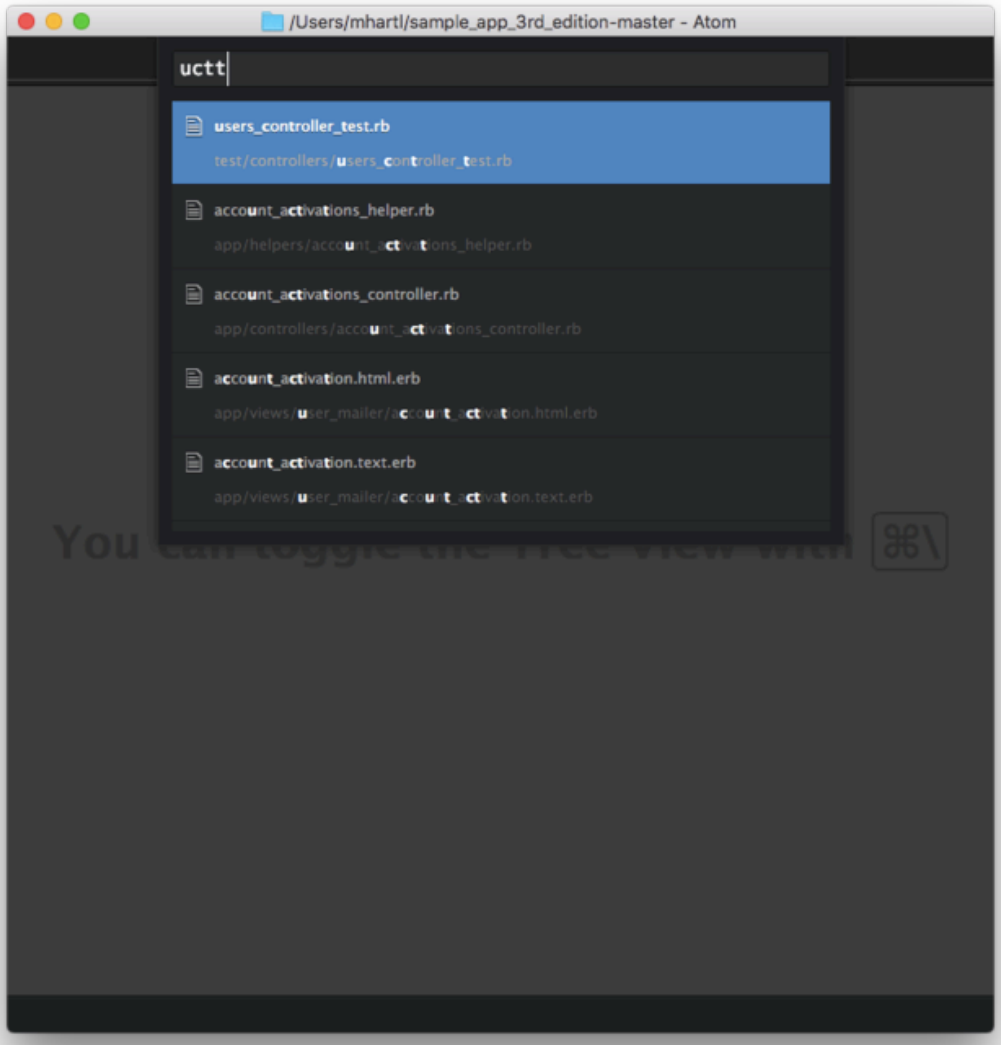

Figure 3.24: A second way to open a file with fuzzy opening.

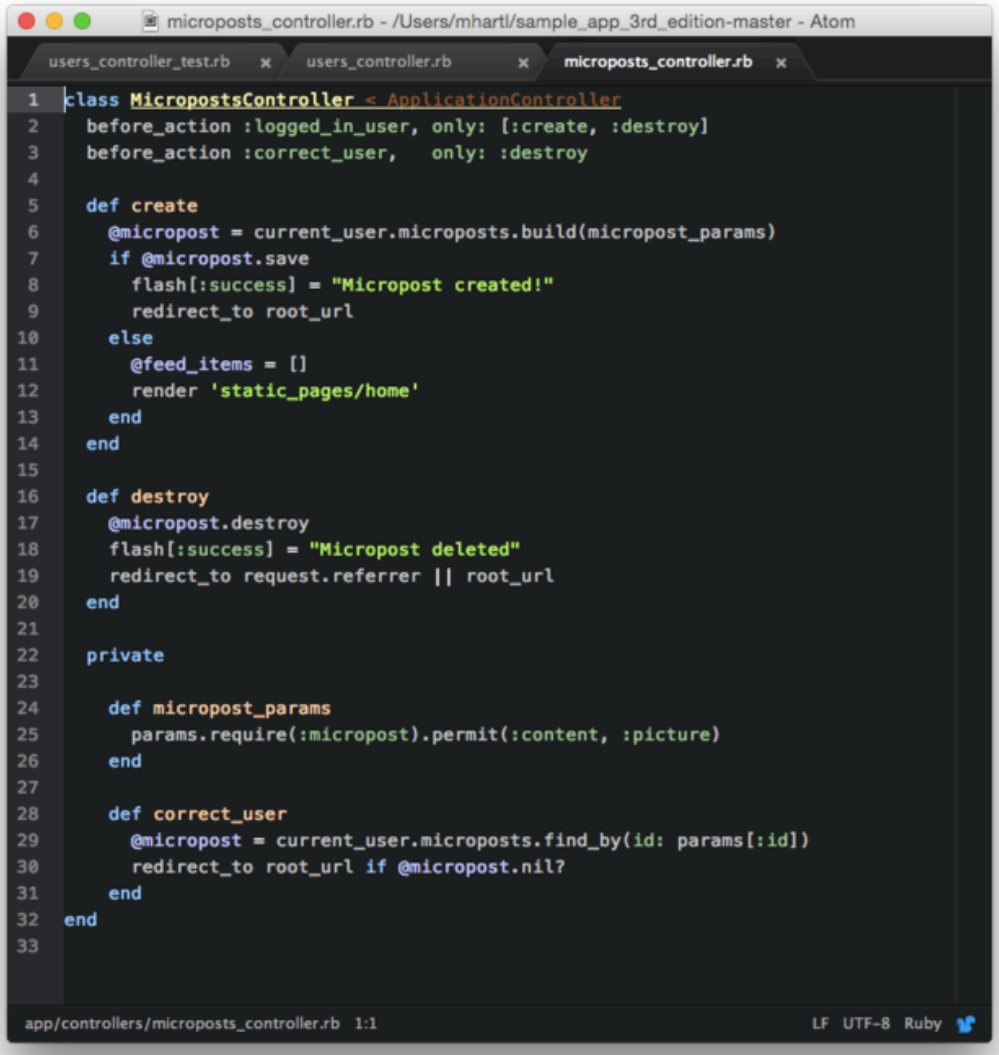

Figure 3.25: Opening multiple tabs.

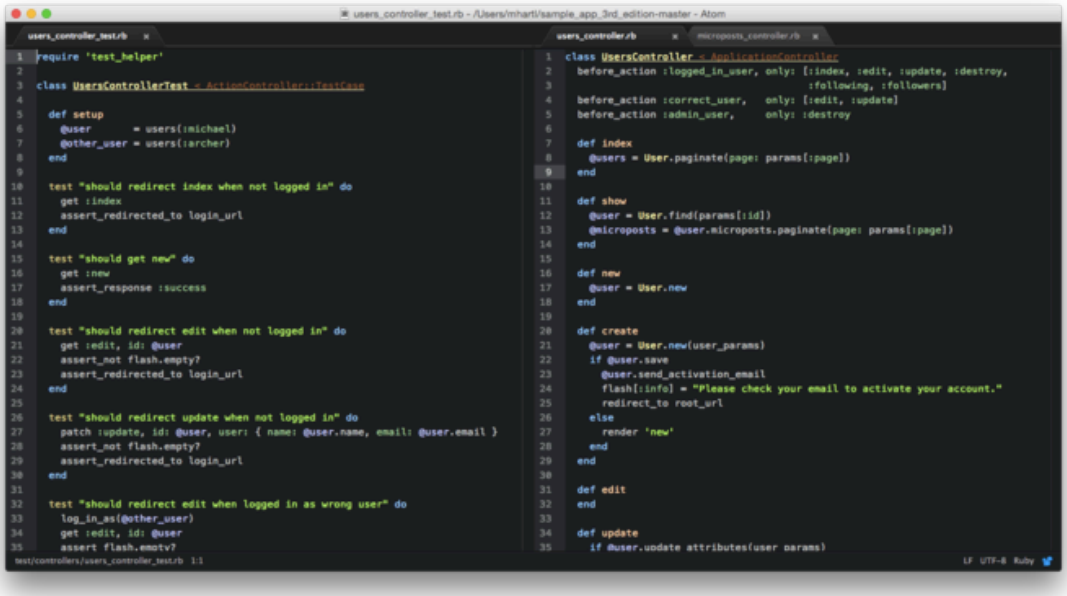

Figure 3.26: Using multiple panes.

I especially like to use different panes for different types of files, such as using the left pane for test code and them right pane for application code. It's also often useful to open the same file in two different panes (Figure 3.27); as I wrote in the meta-tutorial *Learn Enough Tutorial Writing to Be Dangerous*:

> When searching through the document for whatever reason (to fix an error, look up a label for a cross-reference, find a particular string, etc.), it's usually inconvenient to move the cursor and hence lose our place. In this context, it's useful to have the same file open in two different text editor windows… This way, we can use one pane as the main writing area and the other pane as a sort of "random access" window for moving around in the document.

Note: "Panes" are sometimes called "Groups" (e.g., in Sublime Text).

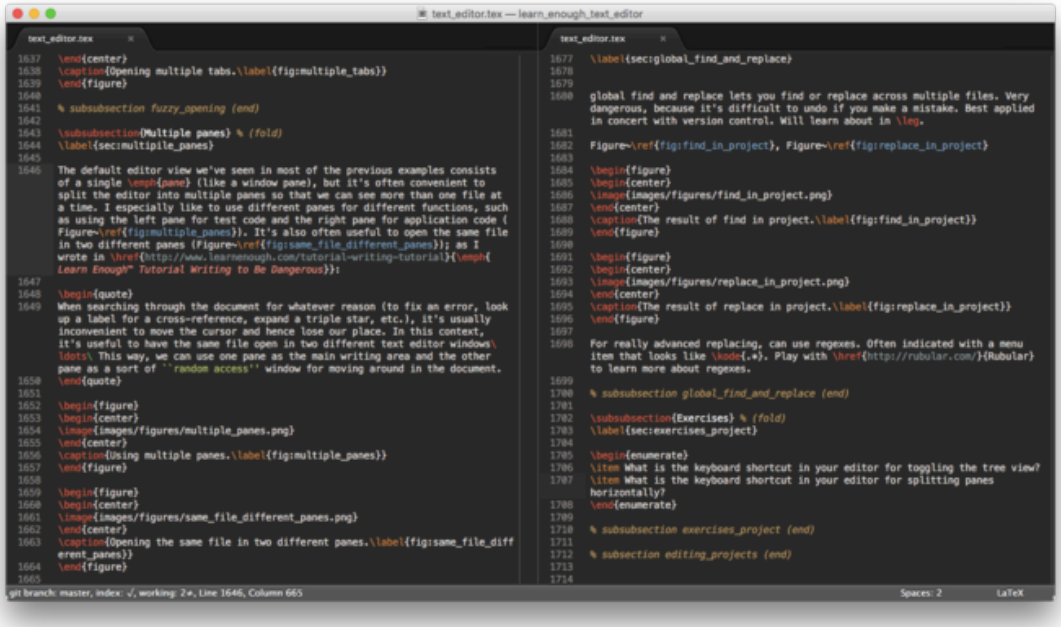

Figure 3.27: Opening the same file in two different panes.

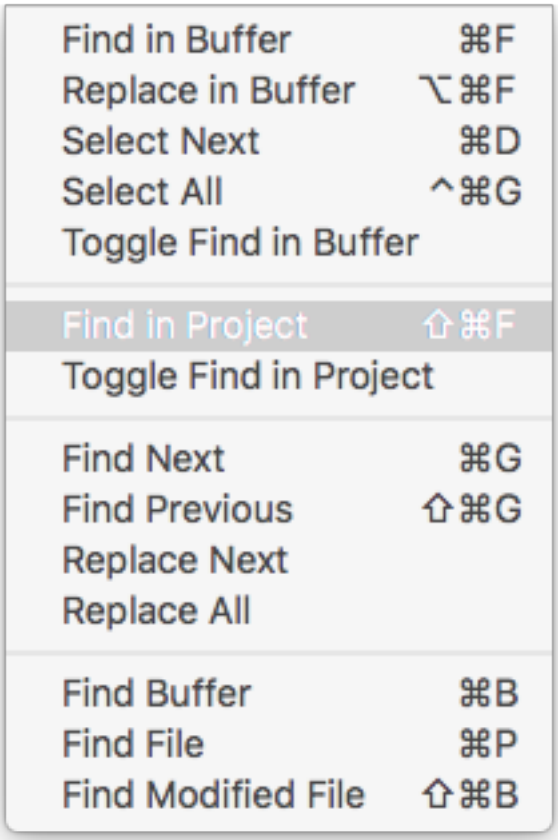

Figure 3.28: A menu item for global find and replace.

3.4.3 Global find and replace

We saw in Section 2.8 how to find and optionally replace content in a single file. When editing projects, it's often useful to be able to do a *global* find and replace across multiple files. As usual, most editors have both a menu item (Figure 3.28) and a keyboard shortcut (often ⇧⌘F).

An example of global find appears in Figure 3.29, which searches for the string "@user" in all project files. The command to globally replace this with "@person" appears in Figure 3.30.

For really advanced replacing, we can use a mini-language for text pat-

Figure 3.29: The result of finding in project.

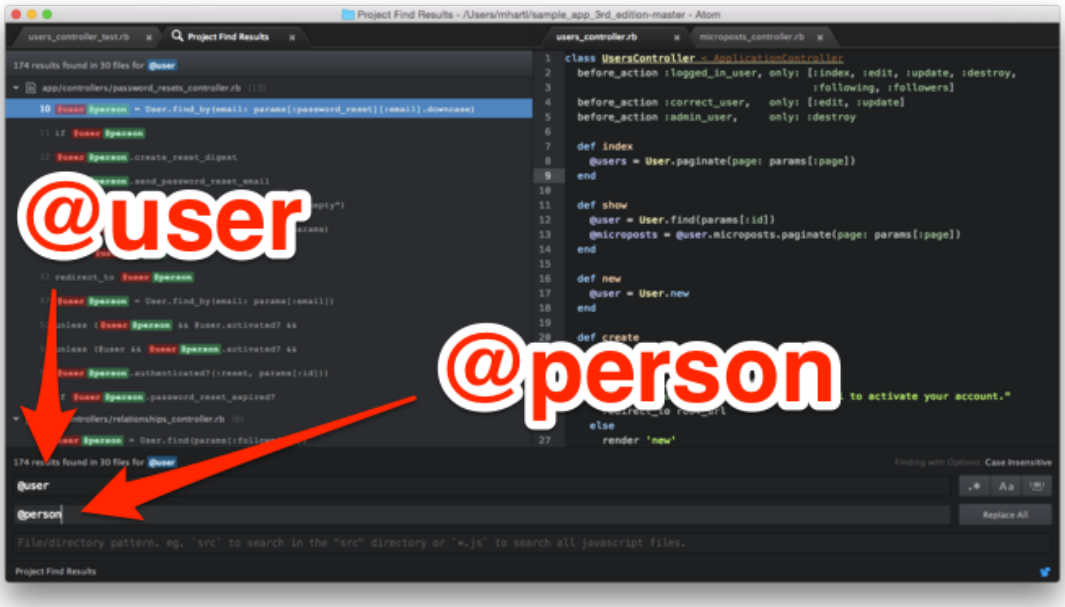

Figure 3.30: The result of replacing in project.

tern matching called *regular expressions* (or *regexes* for short). (Regexes were mentioned briefly in *Learn Enough Command Line to Be Dangerous*.) Let's see how to use regexes to add an annotation to all function definitions in the project, changing

```
def foo
```

to

```
def foo    # function definition
```

and

```
def bar
```

to

```
def bar    # function definition
```

My favorite way to build up regular expressions is using a web application like regex101, which lets us create regexes interactively (Figure 3.31). Moreover, such resources typically include a quick reference to assist us in finding the code for matching particular patterns (Figure 3.32).

We can use the reference in Figure 3.32 to discover a regex for `def` followed by any sequence of characters, which looks like this:

```
def .*
```

Here `.` represents "any character", while `*` matches zero or more of them. Doing a global search using the regex `def .*` matches all the function definitions in the project, as seen in Figure 3.33. Note that in most editors you'll have to enable regex matching by clicking the regex match icon (.* in Figure 3.33).

We can do the replacement mentioned above using parentheses to create two *match groups*:

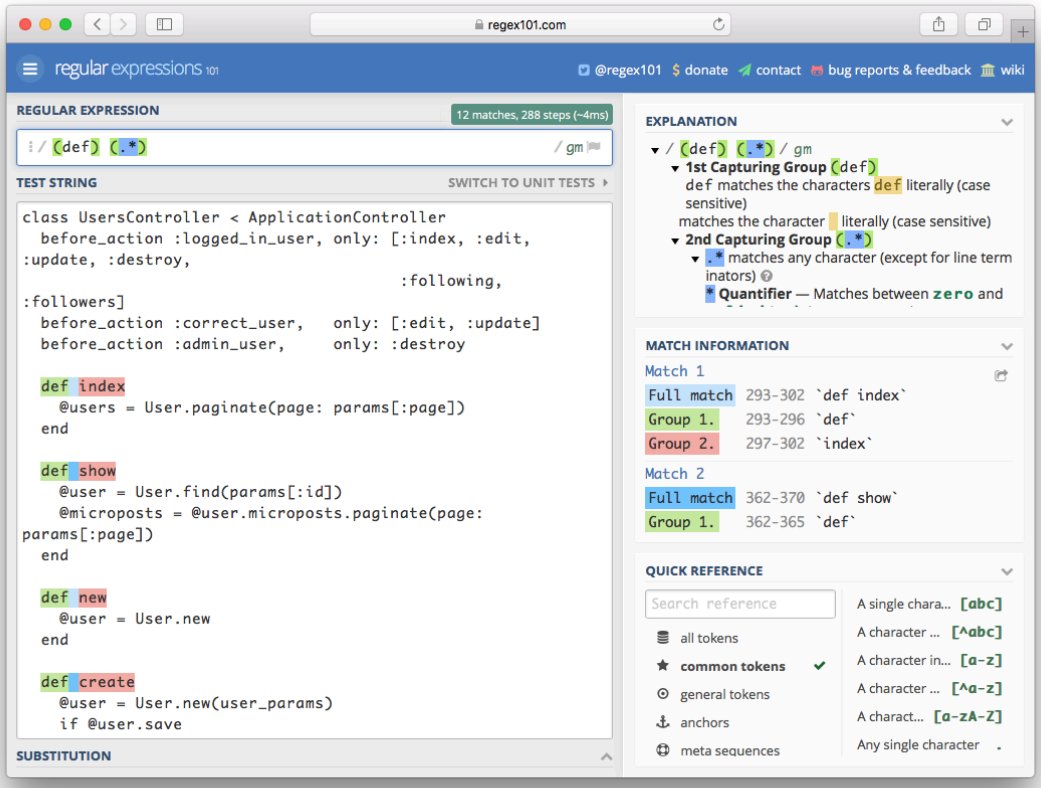

Figure 3.31: An online regex tester.

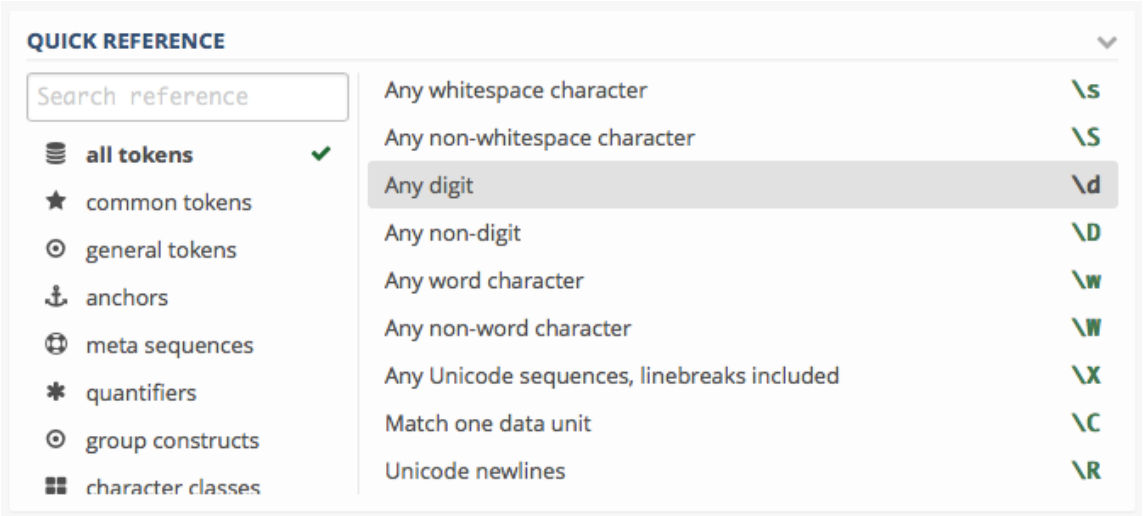

Figure 3.32: A close-up of the regex reference.

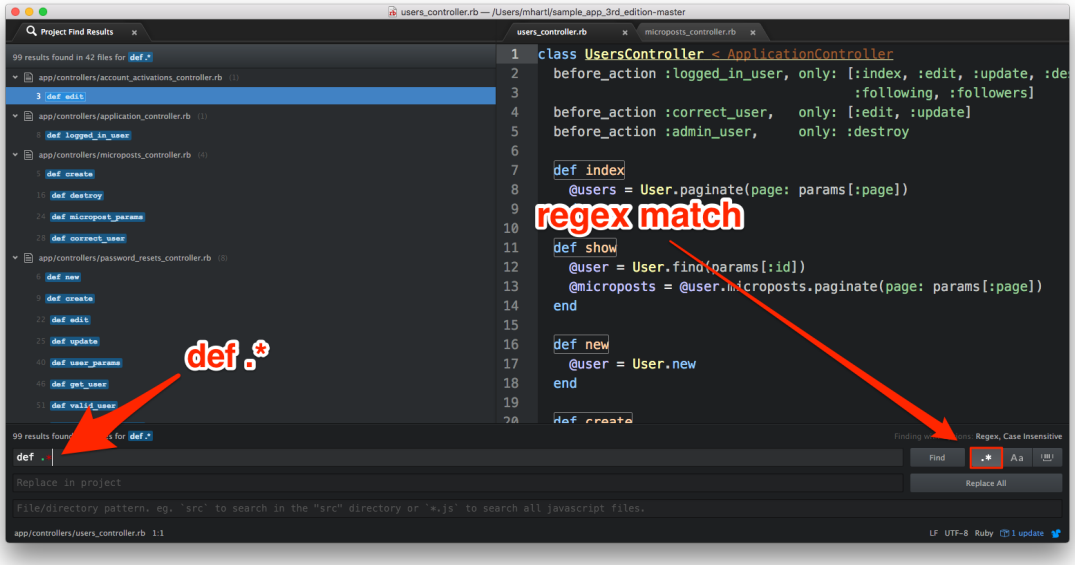

Figure 3.33: Matching a regular expression.

```
(def) (.*)
```

The first match group here is just the constant string **def**, while the second is whatever the function definition happens to be. (These match groups also appear in Figure 3.31.) Inside the "Replace" field in the editor, we can reference these groups using special dollar-sign match numbers, so that we can replace

```
(def) (.*)
```

with

```
$1 $2    # function definition
```

For example, when matching **def foo**, **$1** is **def** and **$2** is **foo**; when matching **def bar**, **$1** is still **def**, but **$2** is **bar**. This means we can annotate all the function definitions at the same time using the commands shown in Figure 3.34. Actually completing this replacement is left as an exercise (Section 3.4.4).

One thing to bear in mind when using global find and replace is that it can be hard to undo. In the case of a single file, it's easy enough to undo a bad replacement with ⌘Z (Section 2.6), but when replacing across multiple files we have to run ⌘Z in *every* affected file, which could be dozens. As a result, I recommend using global find and replace with great caution, and preferably in combination with a version control system such as Git. My general practice is to make a *commit* before any global search and replace so that I can easily undo it if there turns out to be a mistake. (See *Learn Enough Git to Be Dangerous* for more information.)

3.4.4 Exercises

1. What is the keyboard shortcut in your editor for toggling the tree view?
2. What is the keyboard shortcut in your editor for splitting panes horizontally?

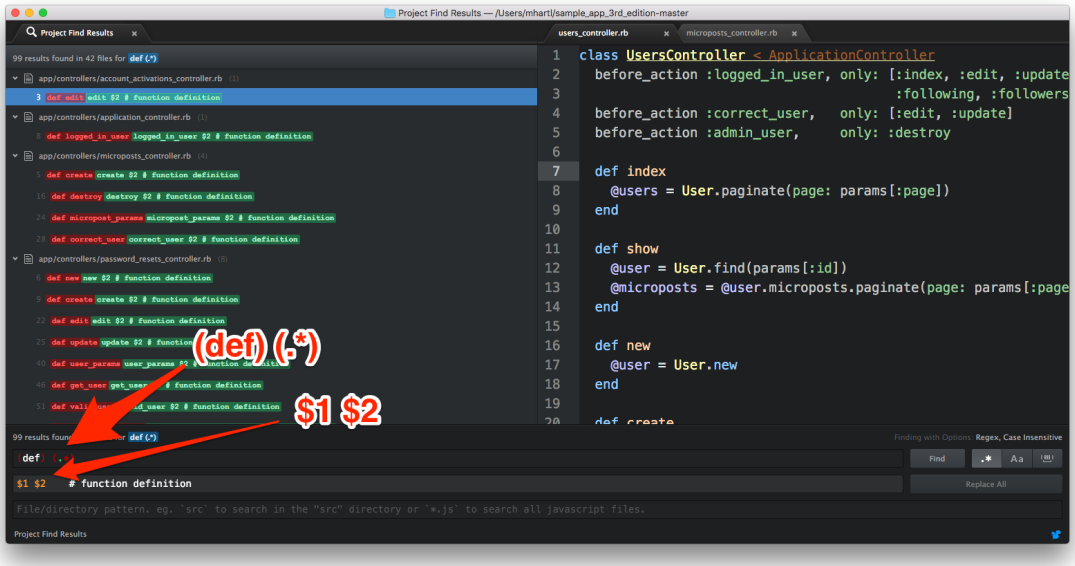

Figure 3.34: Using a match grouping.

3. In the Rails Tutorial sample app project, open the file **`static_pages_controller.rb`** using fuzzy opening.

4. Use global find to find all occurrences of the string **`@user`**.

5. Use global replace to change all occurrences of **`@user`** to **`@person`**.

6. Use a regex match to annotate all function definitions with **`# function definition`** as described in the text.

3.5 Customization

All good text editors are highly customizable, but the options are highly editor-dependent. The most important things are (a) to know what kind of customization is possible and (b) to apply your technical sophistication (Box 1.3) to figure out how to make the desired changes.

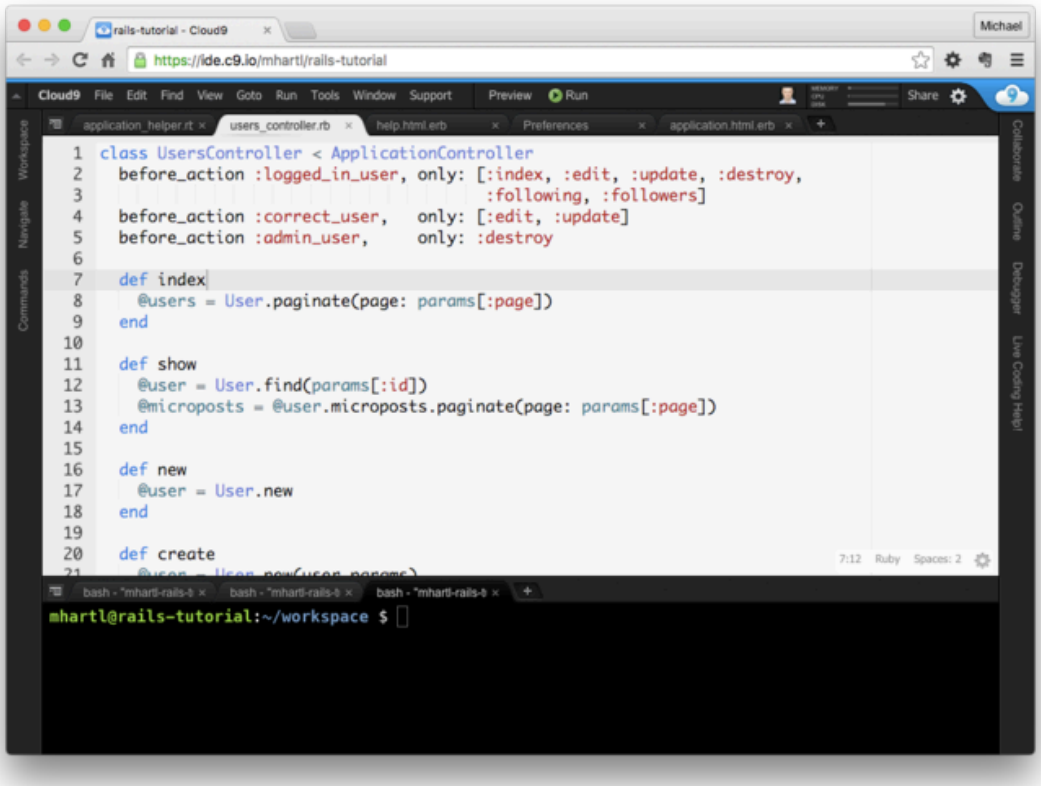

Figure 3.35: The Cloud9 editor with a light background.

For example, one student of the Ruby on Rails Tutorial wrote in asking about the dark background in the Cloud9 editor (e.g., Figure 2.5), wondering if it was possible to use a light background instead. I responded that it was almost certainly possible to change to a light background, even though I didn't know how to do it offhand. I knew that every good programmer's editor has multiple highlighting color schemes, font sizes, tab sizes, etc., so I was confident I could figure out how to change the background color on the Cloud9 editor. And indeed, by clicking around and looking for promising menu items (a textbook application of Box 1.3), I was able to discover the answer (Preferences > Themes > Syntax Theme > Cloud9 Day), as shown in Figure 3.35.

Another feature common to good text editors is some sort of package sys-

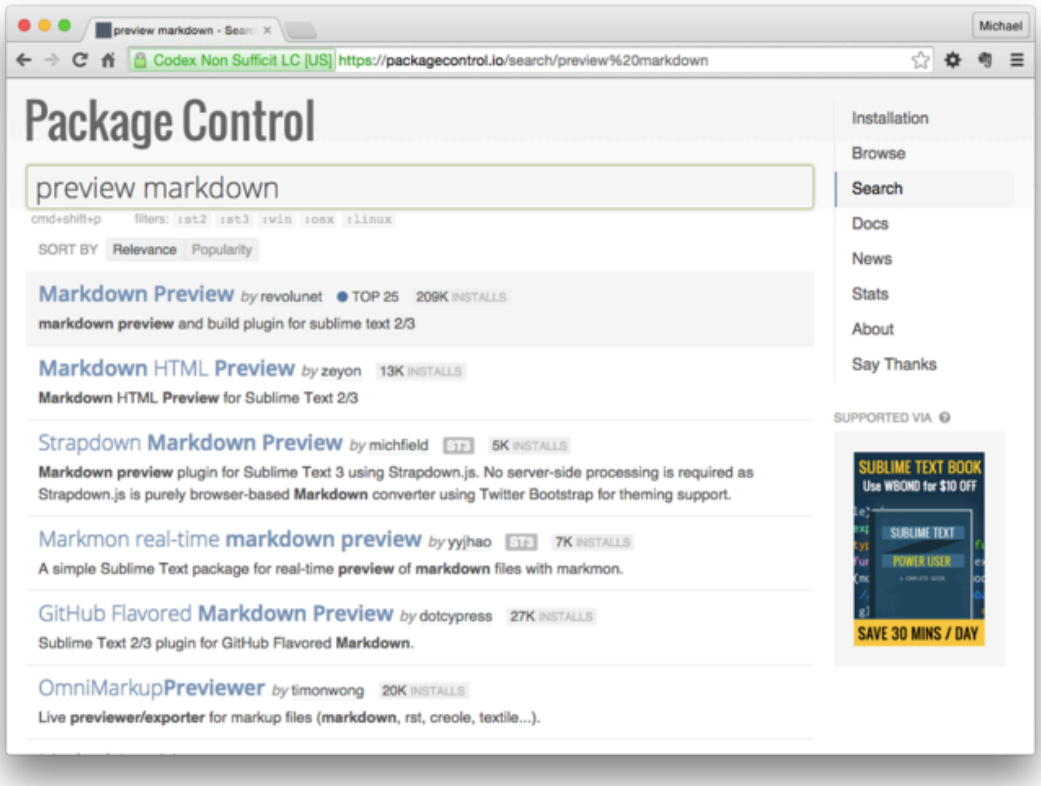

Figure 3.36: Searching for a Sublime Text package.

tem. For example, we saw in Section 2.2.2 that Atom comes with a built-in package to preview Markdown, but in Sublime Text we need to install a separate package called Package Control to do it. One way to find new packages is to Google around for more information, leading to a site like that shown in Figure 3.36. The result is a new option, Sublime Text > Preferences > Package Control, as shown in Figure 3.37 and Figure 3.38.

Most editors allow you to create your own packages of commands, as well as often supporting *snippets* that let you define your own tab triggers (Section 3.1). These are advanced topics, so I recommend deferring them for now. Once you start becoming annoyed by having to repeatedly type the same boilerplate (as in, e.g., Listing 3.11), Google around to figure out how to add custom

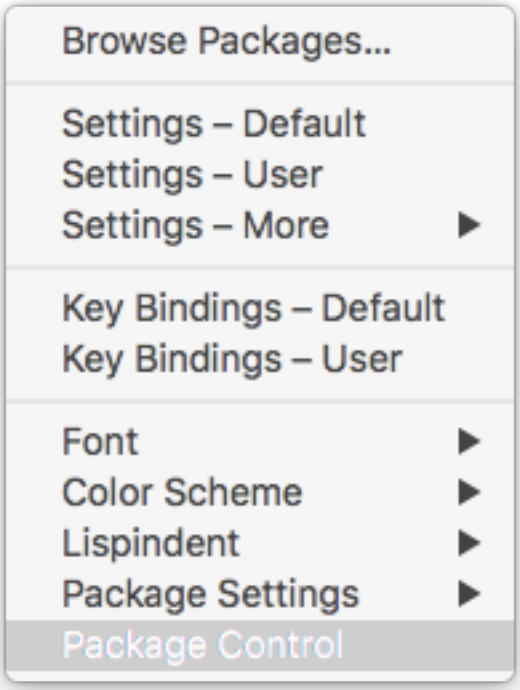

Figure 3.37: Sublime Text's Package Control menu item.

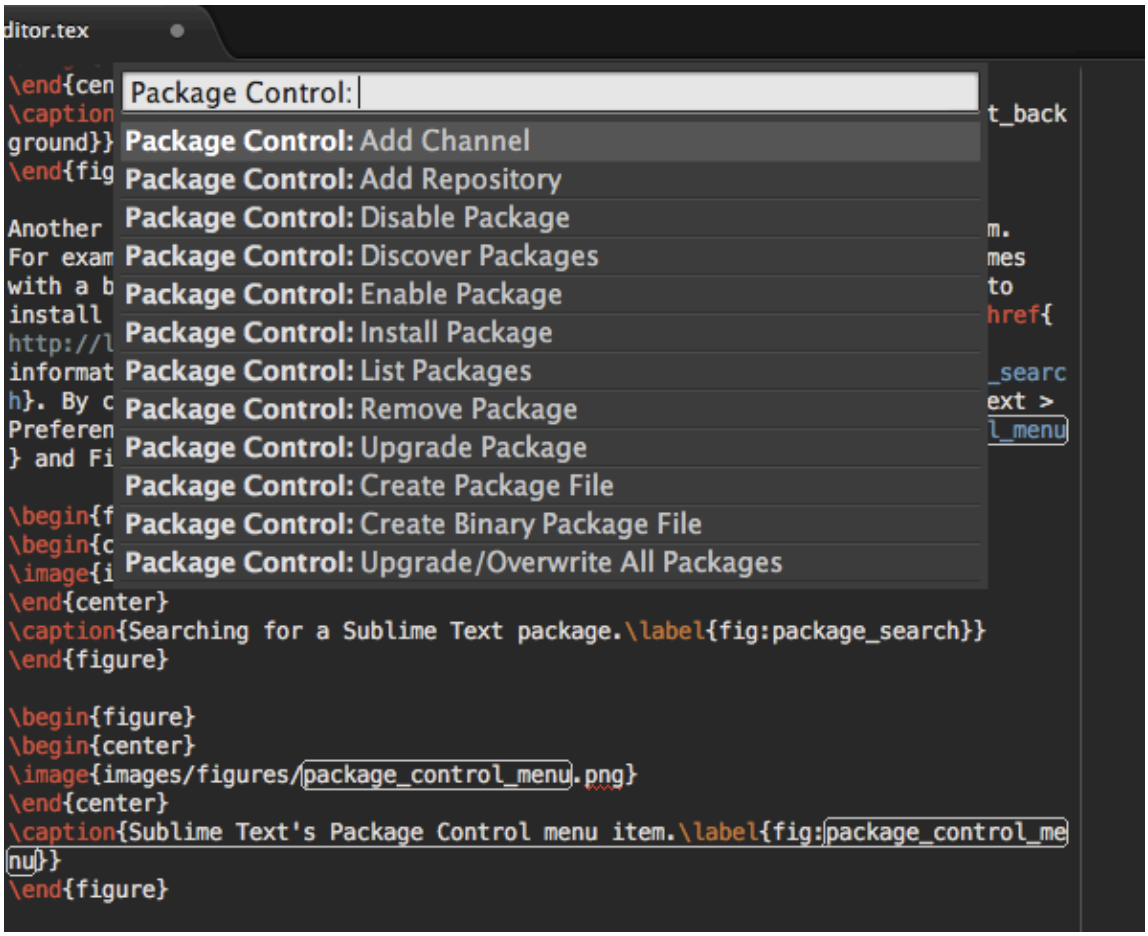

Figure 3.38: Sublime Text's Package Control.

commands to your editor. (The code in Listing 3.11 is generated using the custom Sublime Text tab trigger **`clist`** (for "code listing"), which I have also ported to Atom.)

Listing 3.11: The boilerplate for a code listing in this document.

```
\begin{codelisting}
\label{code:}
\codecaption{}
%= lang:
\begin{code}

\end{code}
\end{codelisting}
```

3.5.1 Exercises

1. Figure out how to change the syntax highlighting theme in your editor. Use the file from Listing 3.6 to confirm the change.

2. In Atom, figure out how to install the `minimap` package. What does this package do? The result for **`sonnets.txt`** should look something like Figure 3.39.

3.6 Summary

- Autocomplete and tab triggers make it easy to type lots of text quickly.
- All good text editors have special features to support writing computer source code, including syntax highlighting, commenting out, indenting & dedenting, and goto line number.
- Many programmers think it's perfectly fine to have lines that are more than 80 columns across, but they are wrong. (Speaking of which, there's nothing quite so entertaining as a holy war (Box 1.6)…)

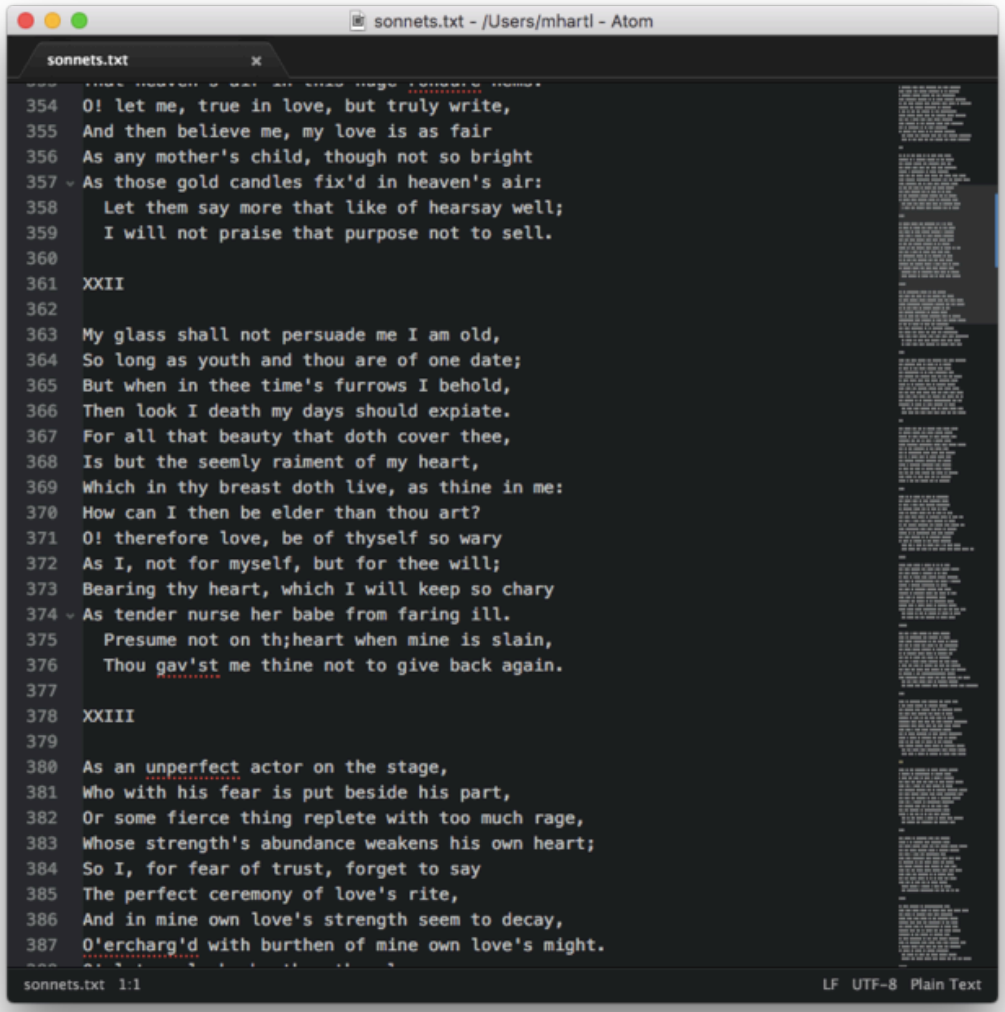

Figure 3.39: The result of adding `minimap` to Atom.

Command	Description
Select + ⌘/	Toggle commenting out
Select + →|	Indent
Select + ⇧ →|	Dedent
^G	Goto line number
⌘W	Close a tab
`$ echo $PATH`	Show the current path variable
`$ chmod +x <filename>`	Make `filename` executable
`$ unzip <filename>.zip`	Unzip a ZIP archive
⌘P	Fuzzy opening
⌘1	Switch focus to tab #1
⇧⌘F	Global find and replace

Table 3.1: Important commands from Chapter 3.

- Once you know how to use the command line and a text editor, it's easy to add custom shell scripts to your system.
- It's common to open entire projects (such as Ruby on Rails applications) all at once using the command line.
- Fuzzy opening is useful when editing projects with large numbers of files.
- Using multiple panes allows the editor to display more than one file at a time.
- Global find and replace is dangerous but powerful.
- All good programmer's editors are extensible and customizable.

Important commands from this section are summarized in Table 3.1.

3.7 Conclusion

Congratulations! You now know enough text editor to be *dangerous*. If you continue down this technical path, you'll keep getting better at using text editors

for years to come, but with the material in this tutorial you've got a great start. For now, you're probably best off working with what you've got, applying your technical sophistication (Box 1.3) when necessary. Once you've got a little more experience under your belt, I recommend seeking out resources specific to your editor of choice. To get you started, here are some links to documentation for the editors mentioned in this tutorial:

- Atom docs
- Sublime Text docs
- Cloud9 editor docs

As a reminder, *Learn Enough Text Editor to Be Dangerous* is just one in a series of tutorials designed to teach the fundamentals of software development. The next step in the series is *Learn Enough Git to Be Dangerous*, and the full sequence appears as follows:

1. **Developer Fundamentals**
   (a) *Learn Enough Command Line to Be Dangerous*
   (b) *Learn Enough Text Editor to Be Dangerous* (you are here)
   (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!