



LEARN ENOUGH

TEXT EDITOR

TO BE DANGEROUS

FUNDAMENTALS 02

TUTORIAL BY
MICHAEL HARTL



LEARN ENOUGH

TEXT EDITOR

TO BE DANGEROUS

FUNDAMENTALS 02

TUTORIAL BY
MICHAEL HARTL

Learn Enough™ Text Editor to Be Dangerous

Michael Hartl

Contents

1 Vim

- 1.1 Starting Vim
- 1.2 Editing small files
- 1.3 Saving and quitting files
- 1.4 Deleting content
- 1.5 Editing large files
- 1.6 Summary

2 Introduction to modern text editors

- 2.1 Choosing a text editor
- 2.2 Opening
- 2.3 Moving
- 2.4 Selecting text
- 2.5 Cut, copy, paste
- 2.6 Deleting and undoing
- 2.7 Saving
- 2.8 Finding and replacing
- 2.9 Summary

3 Advanced text editing

- 3.1 Autocomplete and tab triggers
- 3.2 Writing source code
- 3.3 Writing an executable script
- 3.4 Editing projects
- 3.5 Customization
- 3.6 Summary

4 Conclusion

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*. 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.

Note: *Learn Enough™ Text Editor to Be Dangerous* is available for free online, but ebooks (EPUB, MOBI, PDF) and video screencasts are [available for purchase](#) as well.

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](#).³

Listing 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](#), the desired formatting options are indicated with special *tags* (such as the HTML emphasis tag `...`) 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](#)).

Box 1. Word processors vs. text editors

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 `emphasis` 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, automatic indentation, support for [regular expressions](#), and customization via packages and snippets. A good text editor is thus an essential tool in every technical person's toolkit.

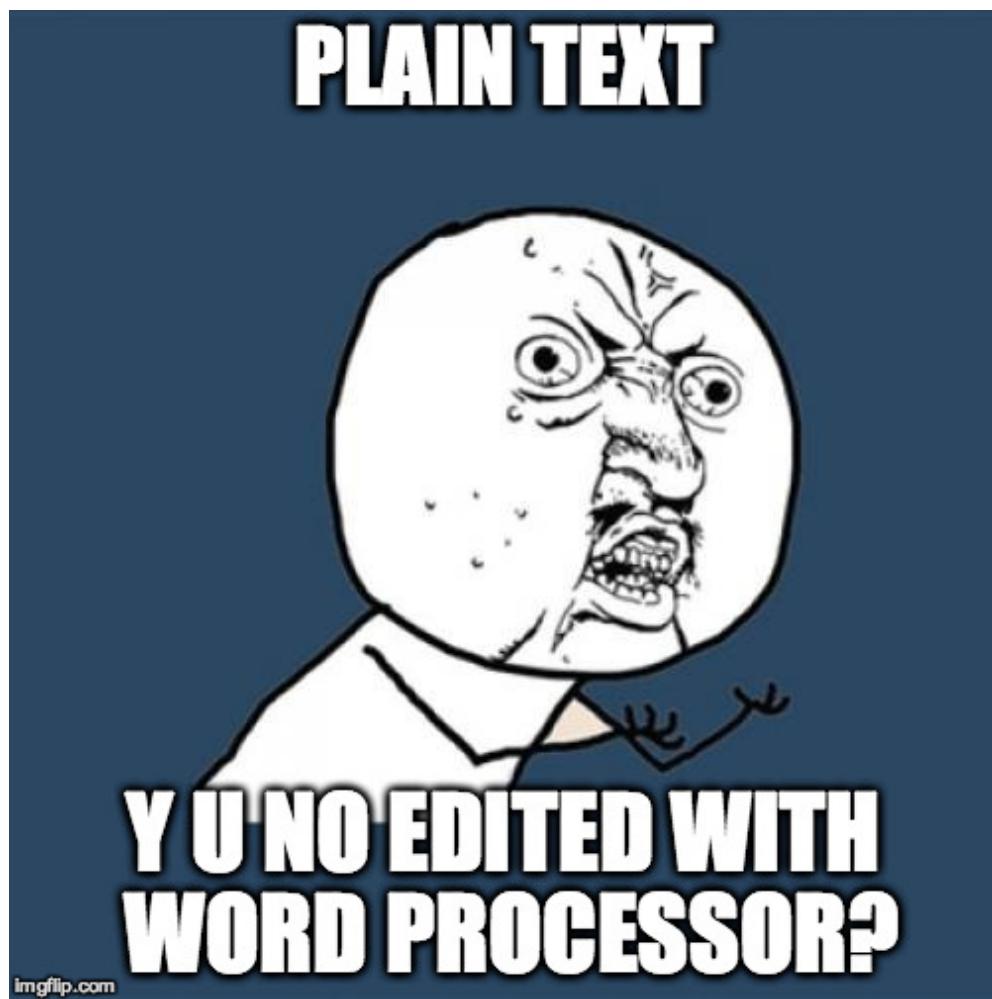


Figure 1: Why not edit plain text with Microsoft Word?

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](#)), 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 [Section 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 additional examples from the closely related [*Sublime Text*](#) editor and from [*Cloud9*](#), a cloud IDE.⁵

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 (Macintosh OS X, 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 strongly encouraged to run a free *Linux virtual machine* (as described in [Box 2](#)) in order to get the most out of this tutorial. Another good option is to use a cloud IDE, which comes with a built-in command line and text editor; to go this route, follow the “[Development environment](#)” section in the [*Ruby on Rails Tutorial book*](#).

Box 2. Running a virtual machine

In order to complete this tutorial, Windows users should install a couple of free programs to run a *virtual machine* (a simulation of a computer) that allows Windows to host a version of the Linux operating system. The steps 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 2](#)).

In the longer run, I recommend switching to a Mac as soon as possible. (*Warning:* As we’ll discuss later in this tutorial ([Box 5](#)), this could start a holy war.) You might have to save up a bit, as Macs are generally more expensive than Windows machines, but in most cases the increased productivity will quickly pay for the difference. (If you find yourself liking Linux, feel free to stick with it, but Macs are generally easier to use with a better user interface. Plus, you can always run Linux inside a VM, even on a Mac.)

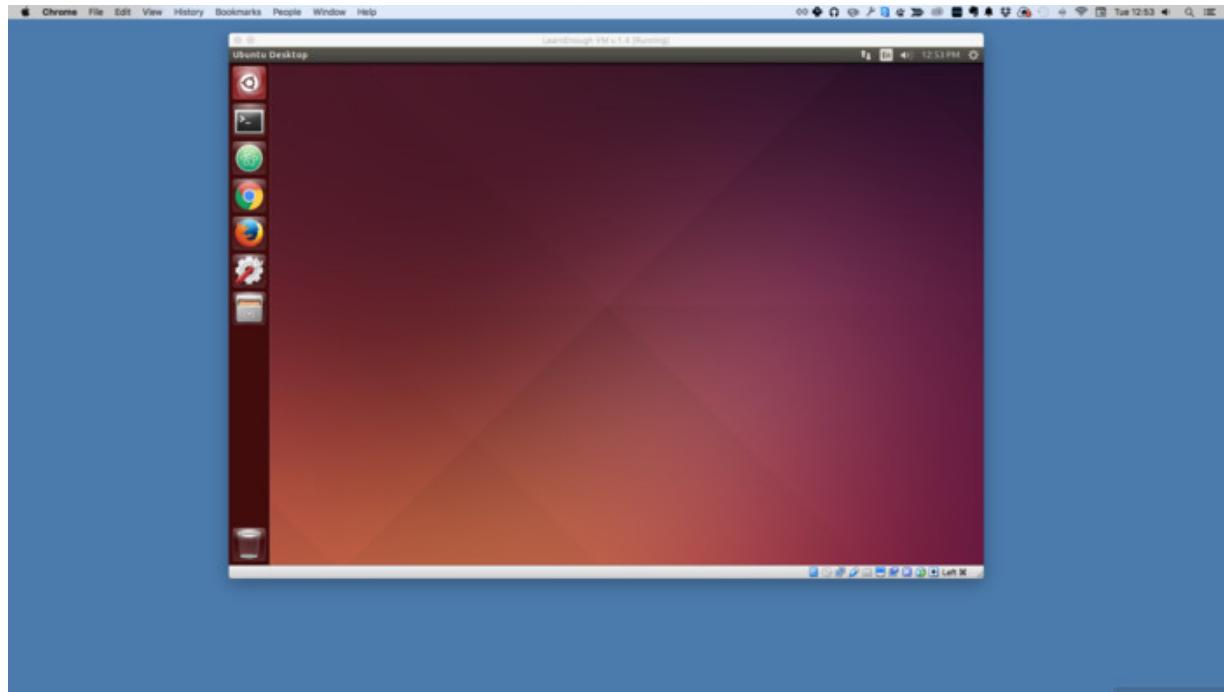


Figure 2: A Linux virtual machine running inside a host OS.

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 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 3. Technical sophistication

The phrase *technical sophistication*, [mentioned before](#) in *Learn Enough™ Command Line to Be Dangerous*, refers to the general ability to use

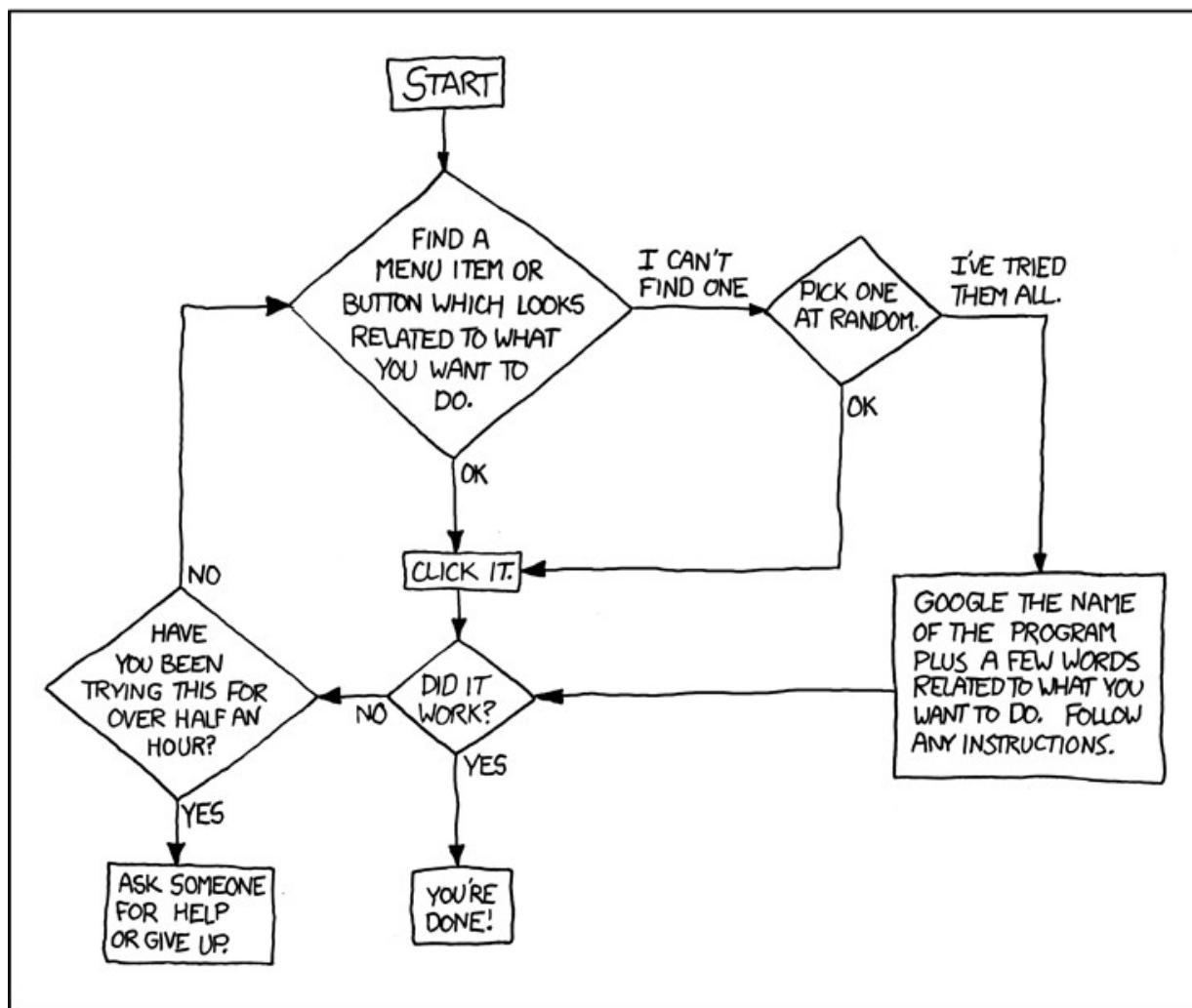
computers and other technical things. This includes both existing knowledge (such as familiarity with text editors and the Unix command line) and the ability to acquire *new* knowledge, as illustrated in [Figure 3](#). Unlike “hard skills” like coding and version control, this latter aspect of technical 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](#), 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 [Figure 3](#) who seem to have the [magical](#) ability to figure out technical things.⁶ (Warning: You might need a new [shirt](#) ([Figure 4](#)).)

DEAR VARIOUS PARENTS, GRANDPARENTS, CO-WORKERS,
AND OTHER "NOT COMPUTER PEOPLE."

WE DON'T MAGICALLY KNOW HOW TO DO EVERYTHING IN EVERY
PROGRAM. WHEN WE HELP YOU, WE'RE USUALLY JUST DOING THIS:



PLEASE PRINT THIS FLOWCHART OUT AND TAPE IT NEAR YOUR SCREEN.
CONGRATULATIONS; YOU'RE NOW THE LOCAL COMPUTER EXPERT!

Figure 3: “[Tech Support Cheat Sheet](#)” (via [xkcd](#)).



Figure 4: A [T-shirt](#) for the technically sophisticated.

1 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 3](#)) to reach its full potential. Vim also has a large and often obscure set of commands, 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 ([Section 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 be there.

This section includes only MVV (Minimum Viable Vim)—just enough to use Vim to do things like edit small configuration files or Git commits.⁷ It’s not even really enough to be *dangerous*. If you’re interested in digging deeper, and especially if you’d like to use Vim as your primary editor, I recommend taking a look at [*Learn Enough™ Vim 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 can impress your friends (or a job interviewer).

1.1 Starting Vim

Unlike most of the modern editors discussed starting in [Section 2](#), Vim can be run directly inside a terminal window, and requires no graphical

interface.⁸ All you do is type **vim** at the prompt:

```
$ vim
```

Typical results of running the `vim` command appear in [Listing 2](#) and [Figure 5](#). In both cases, the tildes (~) are not characters in the file but rather represent lines that have yet to be defined.

Listing 2: A textual representation of a Vim window (message and versions may differ).

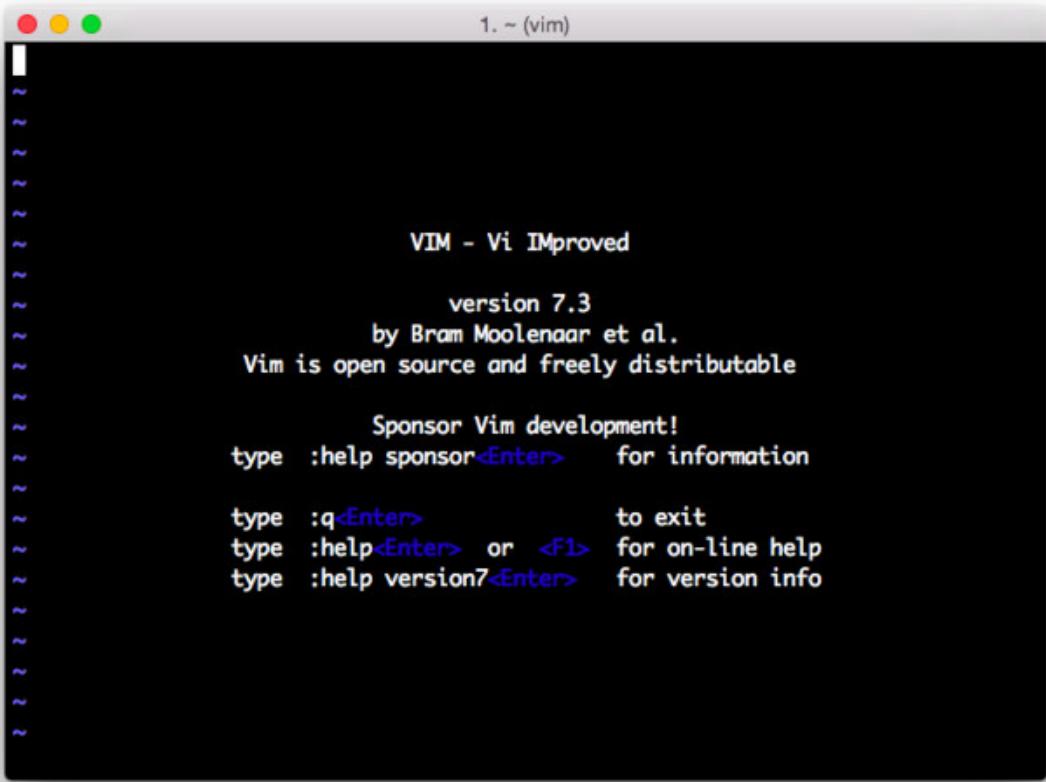


Figure 5: Vim running in a terminal window.

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 4](#)). 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.

Box 4. 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](#)).

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 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 5](#)), claimed the Most

Important Vim Command™ was the only one he ever wanted to learn. Here it is:

```
ESC:q!
```

This command means “Press the Escape key, then type ‘colon q exclamation-point.’.” 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 5. Holy wars: vi vs. Emacs

The [Jargon File](#) defines [*holy wars*](#) as follows:

holy wars: n.

[from [Usenet](#), but may predate it; common] [*flame wars*](#) 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 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 6](#)).



Figure 6: Watching a holy war play out ([Box 5](#)) can be entertaining.

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.

1.2 Editing small files

Now that we know the Most Important Vim Command™ ([Section 1.1](#)), we’ll start learning how to use Vim for real by opening and editing a small file. In [Section 1.1](#), 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, 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*.⁹ 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 1.3](#) 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 1.1](#), we learned that Vim starts in normal mode, which means that we 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 3](#).¹⁰ (There may be other existing content, which you should ignore for now.)

Listing 3: Adding some text after typing `i` to insert.

```
~/.bash_profile
```

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

After entering the text in [Listing 3](#), press `ESC` (the escape key) to switch from insertion mode back to normal mode.

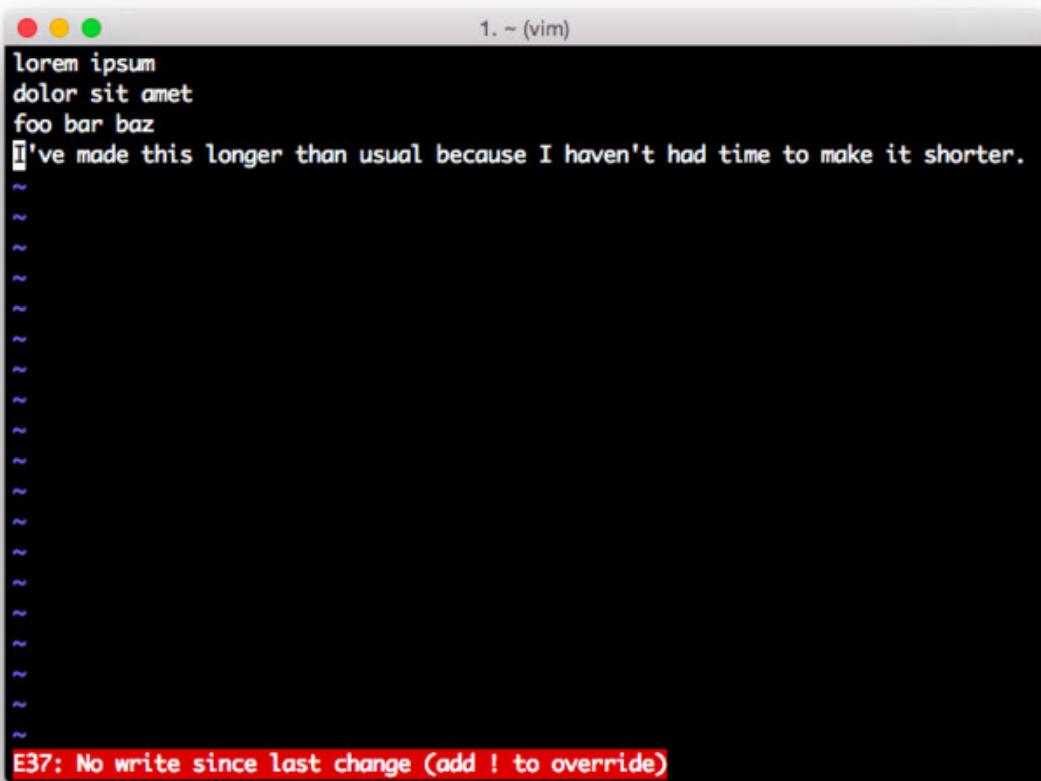
Now that we have some text on a few lines, we can learn some commands for moving around small files. (We'll cover some commands for navigating large files in [Section 1.5](#).) The easiest way to move around is to use the arrow keys—up, down, left, right—which is what I recommend.¹¹ Vim has literally *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.¹²

Exercises

1. Use the arrow keys to navigate to Line 4 in the file from [Listing 3](#).
2. Use the arrow keys to go to the end and then the beginning of Line 4. Cumbersome, eh?
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.

1.3 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 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`, 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 7](#).



The screenshot shows a terminal window titled "1. ~ (vim)". The Vim editor is open with the following text:

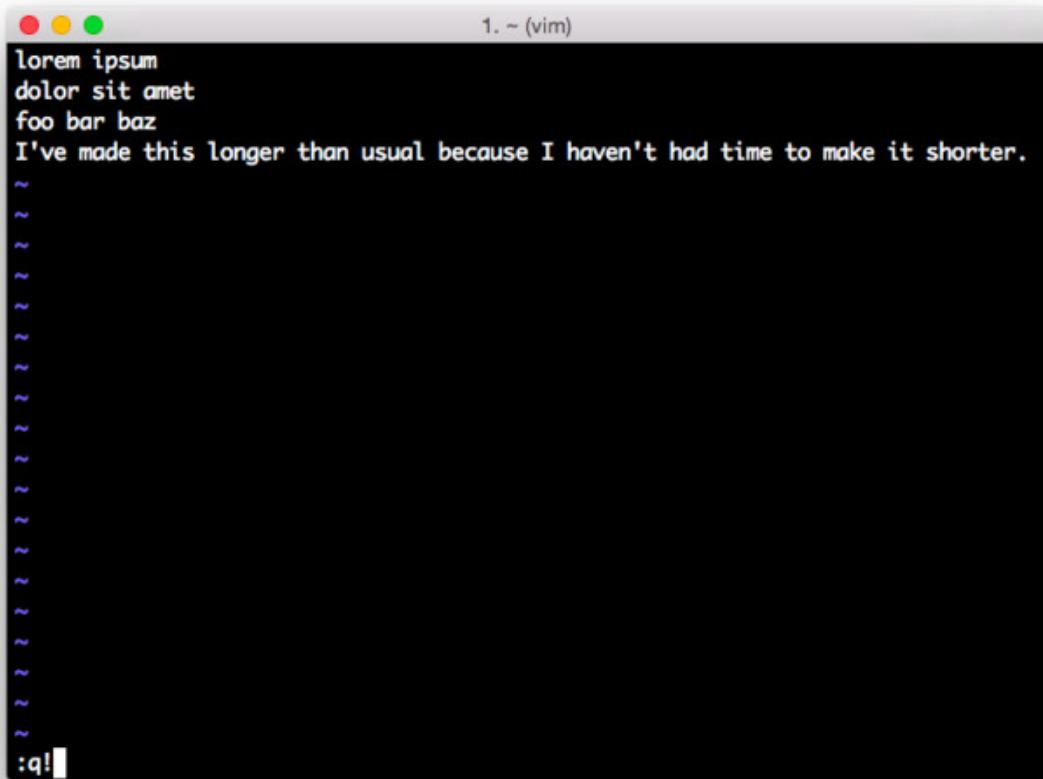
```
lorem ipsum
dolor sit amet
foo bar baz
I've made this longer than usual because I haven't had time to make it shorter.
```

At the bottom of the screen, a red status bar displays the error message:

E37: No write since last change (add ! to override)

Figure 7: Trying to quit a file with unsaved changes.

Following the message's advice, we can type `:q!` to force Vim to quit without saving any changes ([Figure 8](#)), which returns us to the command line.



```
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.
~
~
```

:q!

Figure 8: Forcing Vim to quit.

You may have noticed that we're now in a position to understand the Most Important Vim Command™ introduced in [Section 1.1](#): 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)¹³ followed by `:q!` (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 1.2](#), 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. 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 human-readable values for the sizes (e.g., 29K instead of 29592 for a 29-kilobyte file), including all of them (even hidden ones), ordered by reverse time, long 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 4](#)
3. Write the file using `:w`
4. Quit Vim by typing `:q`

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 [Section 2](#) all support native keybindings.)

Listing 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 5](#).¹⁴

Listing 5: Activating the alias by sourcing the Bash profile.

```
$ source .bash_profile
```

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

```
$ lr  
.  
.br  
  
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.

Exercises

1. Define an alias `g` for the commonly used *case-insensitive grep* `grep -i`.
 - 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 6](#), which downloads a longer text file for use in [Section 1.5](#).

Listing 6: Downloading a longer text file for use in a future section.

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

1.4 Deleting content

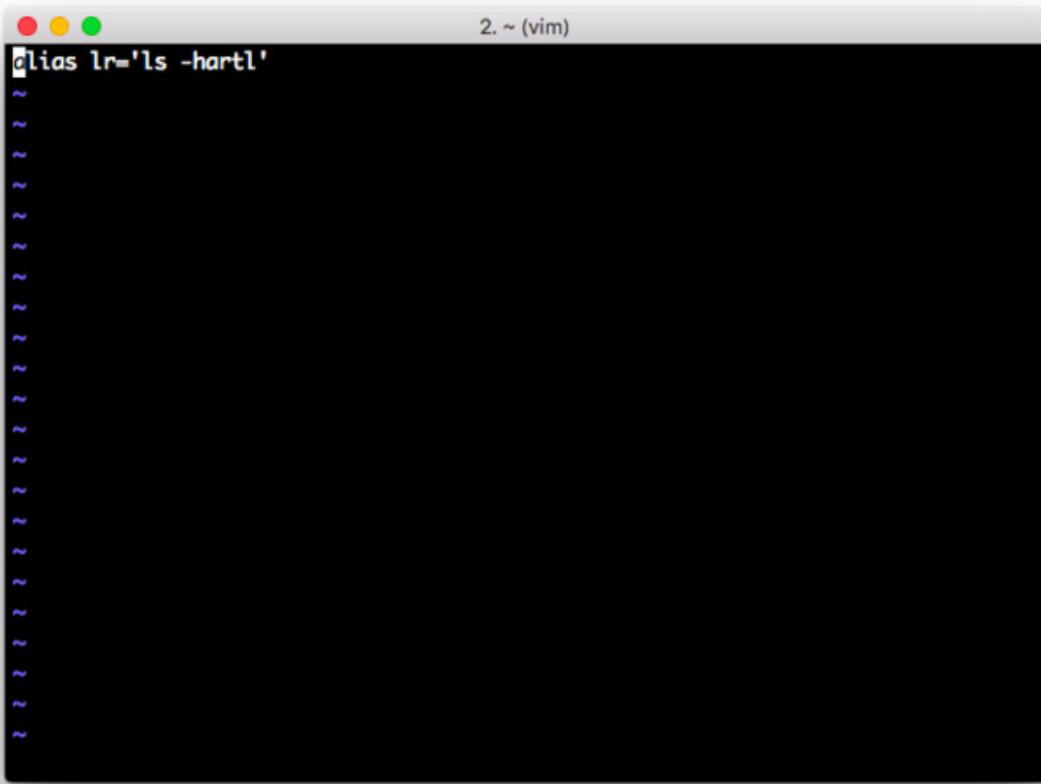
As with every category of text manipulation, Vim has an enormous number of commands for deleting content, but in this section we're just going to cover the absolute minimum. We'll start with deleting single characters, which we can do in normal mode using the `x` command:

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 9](#)) and press `x`

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

Figure 9: Preparing to delete a letter using **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 get rid of the extra **alias** we added by pressing **dd** to delete the line. Voilà ! It should be gone ([Figure 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.)



A screenshot of a terminal window titled "2. ~ (vim)". The window shows a single line of code: "alias lr='ls -hartl'". Below this line, there are approximately 20 blank lines, each consisting of a single tilde (~) character. This visual representation indicates that the line above has been deleted.

Figure 10: The result of deleting a line with `dd`.

Exercises

1. Using Vim, open a new file called `foo.txt`.
2. Insert the string “A leopard can’t change it’s spots.” ([Figure 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](#).)
4. Use `dd` to delete the line, then use `p` to paste it repeatedly into the document.

- Save the document and quit using a single command. *Hint:* See the final exercise in [Section 2.7.1](#).



Figure 11: This [animal](#)'s spot-changing abilities are frequently questioned.

1.5 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 1.3](#), you should so do now ([Listing 7](#)).¹⁶

Listing 7: Downloading Shakespeare's Sonnets.

```
$ curl -OL 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 12](#). Because of its length, this file is far too long to navigate conveniently by hand.

2. ~ (vim)
Shake-speare's Sonnets

I

From fairest creatures we desire increase,
That thereby beauty's Rose might never die,
But as the riper should by time decease,
His tender heir might bear his memory:
But thou contracted to thine own bright eyes,
Feed'st thy light's flame with self-substantial fuel,
Making a famine where abundance lies,
Thy self thy foe, to thy sweet self too cruel:
Thou that art now the world's fresh ornament,
And only herald to the gaudy spring,
Within thine own bud buriest thy content,
And tender churl mak'st waste in niggarding:
Pity the world, or else this glutton be,
To eat the world's due by the grave and thee.

II

When forty winters shall besiege thy brow,
And dig deep trenches in thy beauty's field,
"sonnets.txt" 2620L, 95635C

Figure 12: Some file stats about displayed upon starting Vim.

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 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.

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.

1.6 Summary

Important commands from this section are summarized in [Table 1](#).

Command	Description
ESC :q !	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	Save (write) a file
:q	Quit a file (must be saved)
:wq	Write and quit a file
:q !	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: Important Vim commands from [Section 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.

2 Introduction to 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](#) and [Atom](#), and editors in the cloud like [Nitrous](#) and [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 ([Section 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](#) shows the symbols for the various keys on a typical Macintosh keyboard. Apply your technical sophistication ([Box 3](#)) if your keyboard differs.

Key	Symbol
Command	⌘

Control	^
Shift	↑
Option	↖
Up, down, left, right	↑ ↓ ← →
Enter/Return	↵
Tab	→
Delete	⌫

Table 2: 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 are Atom and Sublime Text (sometimes just called “Sublime”).¹⁸ Each has its own advantages and disadvantages.

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, OS X, Linux)

- 4. Easy to set up command-line tools
- 5. Backed by collaboration powerhouse [GitHub](#)
- Disadvantages
 - 1. Slower than Sublime
 - 2. Based on browser technologies, which are largely untested in editors
 - 3. Not as mature as some alternatives

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, OS X, Linux)
 - 5. Backed by a profitable company that has a good track record of support and development
- Disadvantages
 - 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

It's hard to go wrong with either of these choices, and my main day-to-day editor is currently Sublime Text, but because Atom is 100% free I think it's

probably the best choice for new users. The good news is that most of the skills in the sections that follow are universal, and if you learn Atom but decide to switch to Sublime Text (or even a cloud IDE) for your daily editing, most of the core ideas will translate easily.

Exercises

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

1. Download and install either [Atom](#) or [Sublime Text](#). Note: Atom comes pre-installed on the Linux virtual machine described in [Box 2](#), so if you're using the VM you can skip this step.
2. If using Atom, go to Atom > Install shell commands to enable the **atom** command at the command line ([Figure 13](#)).
3. 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 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*.

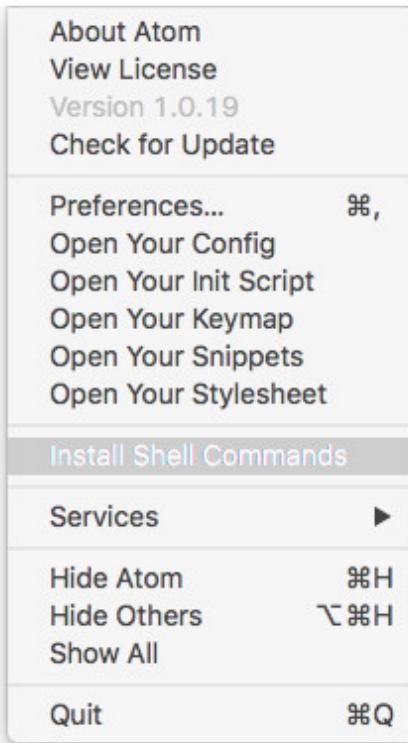


Figure 13: Installing Atom’s shell commands.

2.2 Opening

To open files, we’re going to use the command configured in [Section 2.1.3](#) to launch the editor and open the file at the same time (a method we used with `vim` in [Section 1.2](#)). 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 1.5](#), we’ll use `curl` command to download the file at the command line:

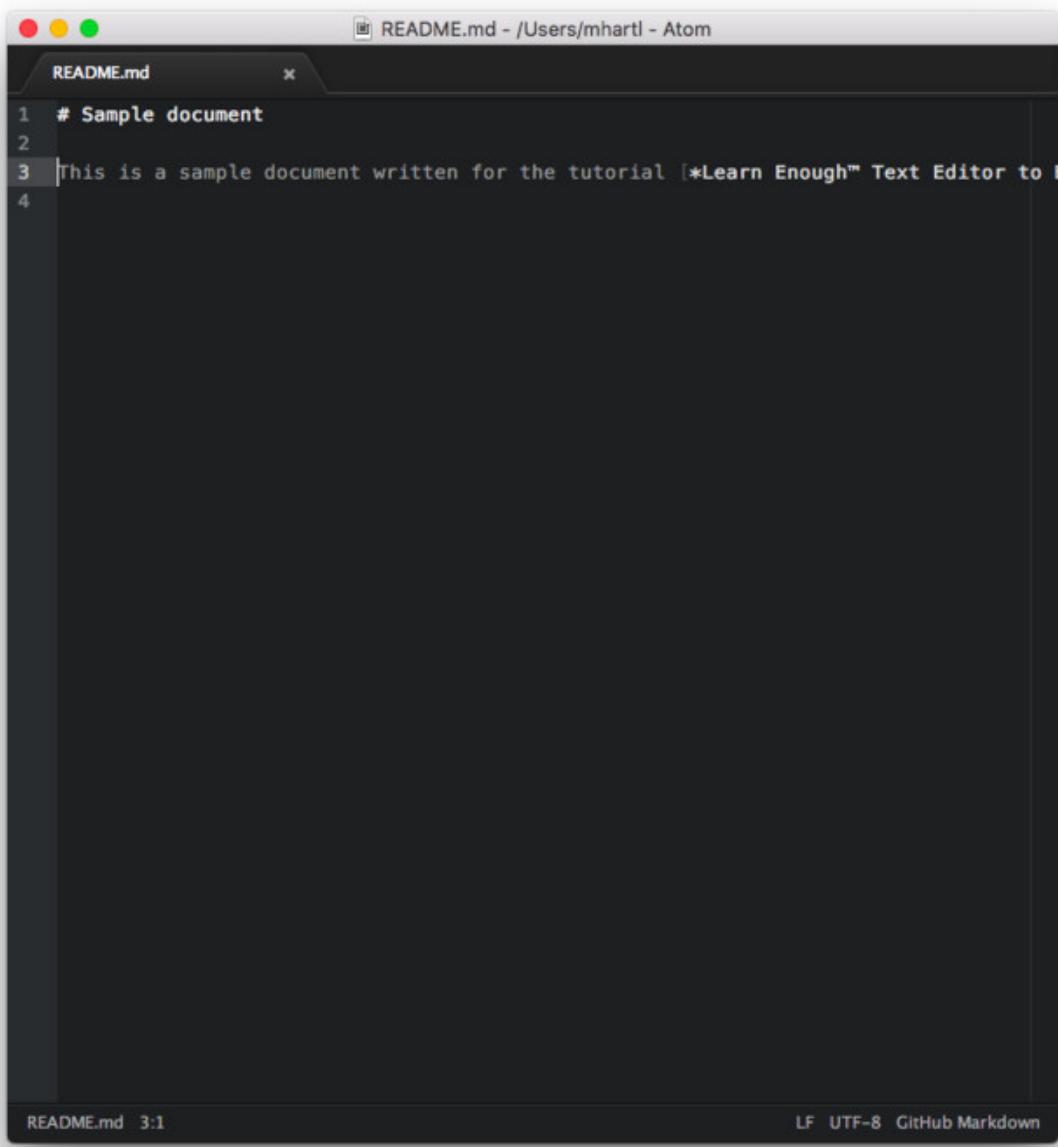
```
$ curl -OL cdn.learnenough.com/README.md
```

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
```

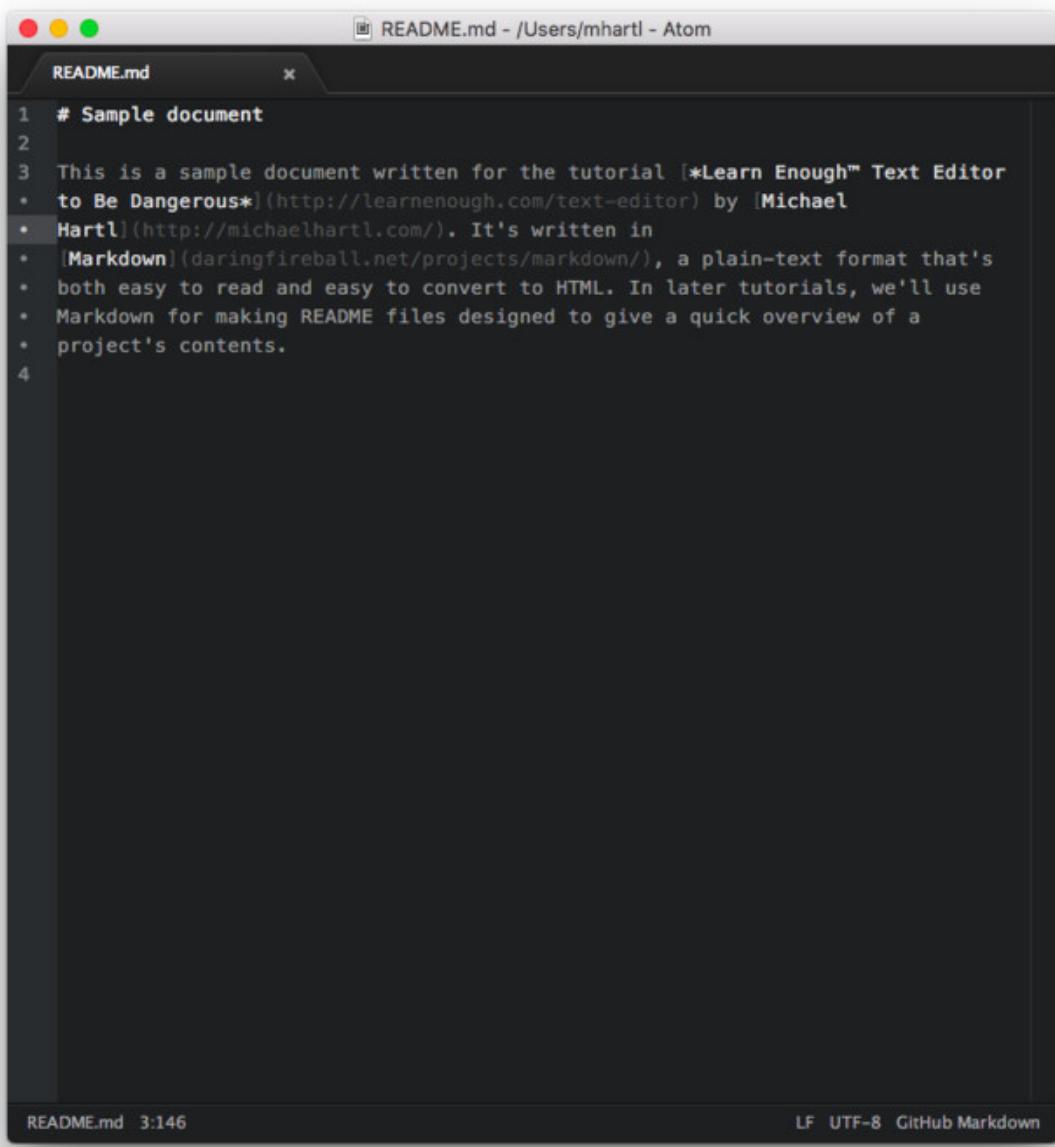
(If this doesn't work, be sure you've installed the Atom shell commands as shown in [Figure 13](#).) The result of opening `README.md` in Atom should look something like [Figure 14](#) or [Figure 15](#). (If this is your first time opening Atom, it's also possible you'll see a one-time greeting screen. As usual, apply [Box 3](#).) [Figure 14](#) 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 16](#).

A screenshot of the Atom text editor interface. The window title is "README.md ~ /Users/mhartl - Atom". The tab bar shows "README.md" and an "x". The main editor area contains the following text:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor to I
4
```

The status bar at the bottom left shows "README.md 3:1" and at the bottom right shows "LF UTF-8 GitHub Markdown".

Figure 14: The sample file with word wrap off.



A screenshot of the Atom text editor interface. The window title is "README.md ~ /Users/mhartl - Atom". The tab bar shows "README.md" and an "x". The main editor area contains the following Markdown content:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
* to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
* Hartl](http://michaelhartl.com/). It's written in
* [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
* both easy to read and easy to convert to HTML. In later tutorials, we'll use
* Markdown for making README files designed to give a quick overview of a
* project's contents.
4
```

The status bar at the bottom left shows "README.md 3:146" and at the bottom right shows "LF UTF-8 GitHub Markdown".

Figure 15: The sample file with word wrap on.



Figure 16: The menu item to [toggle](#) word wrap.

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 can be used to open files at the Cloud9 command line).¹⁹ Double-clicking on **README.md** in the filesystem navigator ([Figure 17](#)) opens the file in Cloud9's editor, as shown in [Figure 18](#). [Figure 19](#) shows the file after clicking "Navigate" to close the filesystem navigator, and we see that, as in [Figure 14](#), the line extends inconveniently off the screen. We can fix this using View > Wrap Lines as shown in [Figure 20](#), with the word-wrapped result appearing as in [Figure 21](#). (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 3](#))).

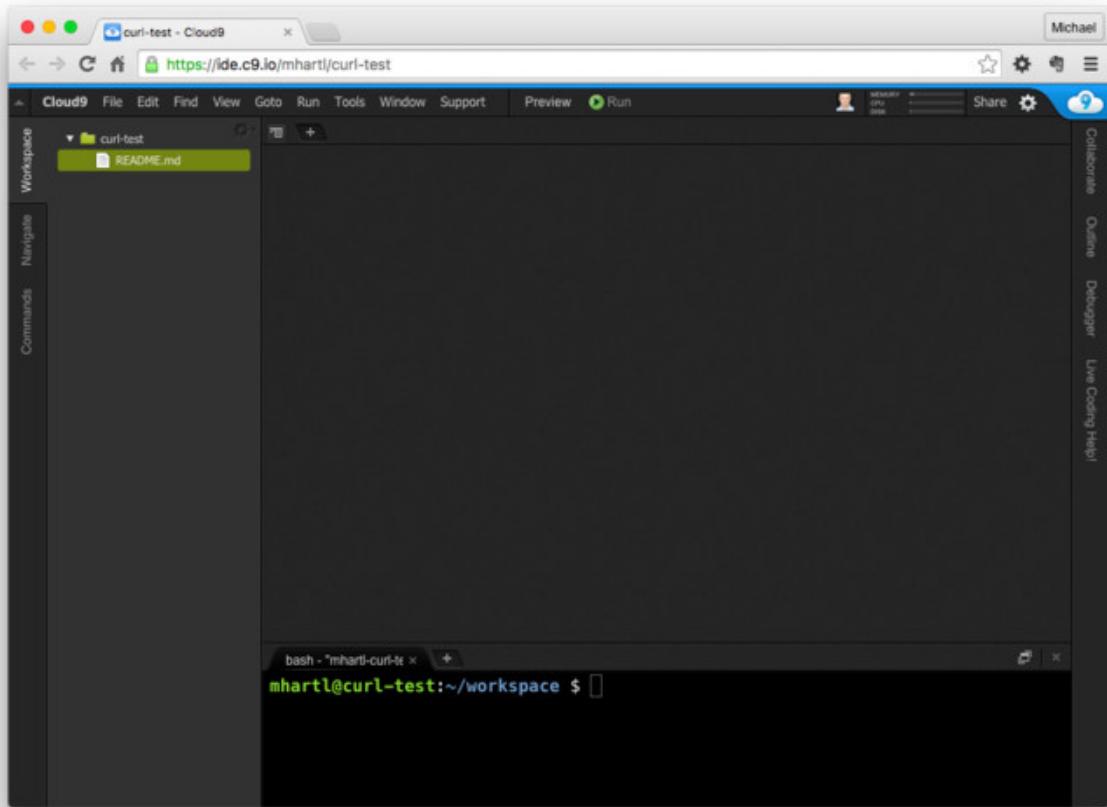


Figure 17: The Cloud9 filesystem navigator.

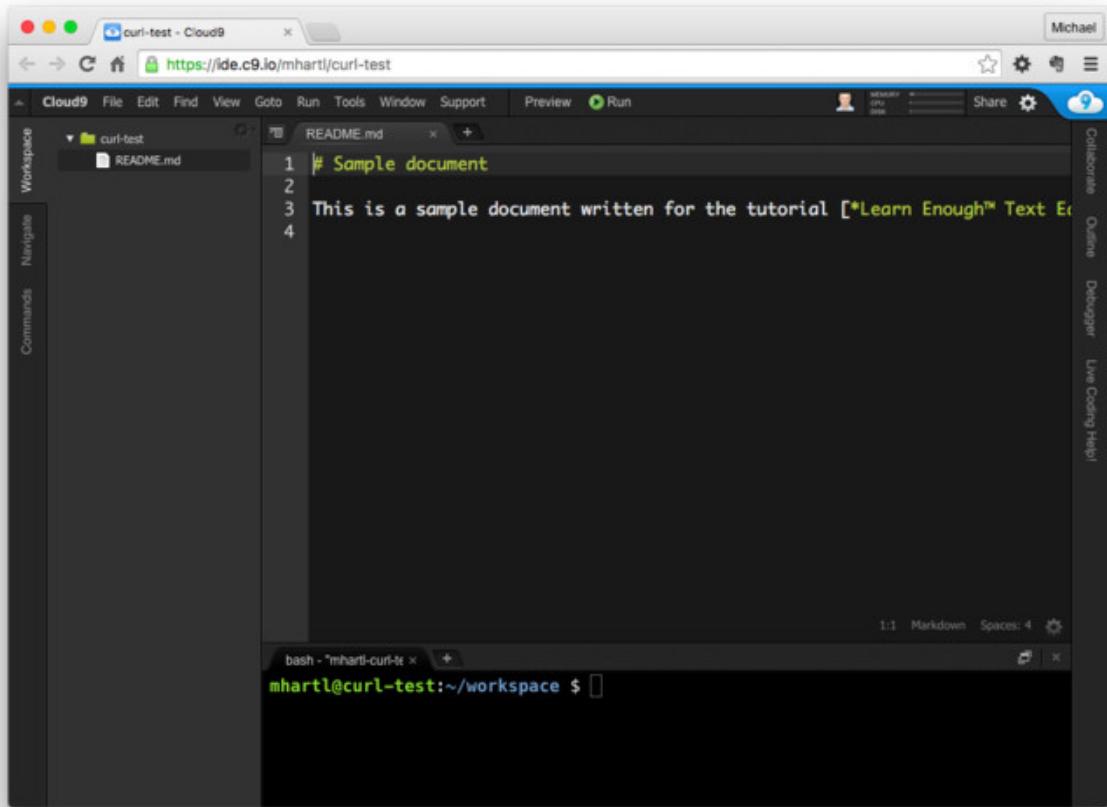


Figure 18: Cloud9 after double-clicking on `README.md`.

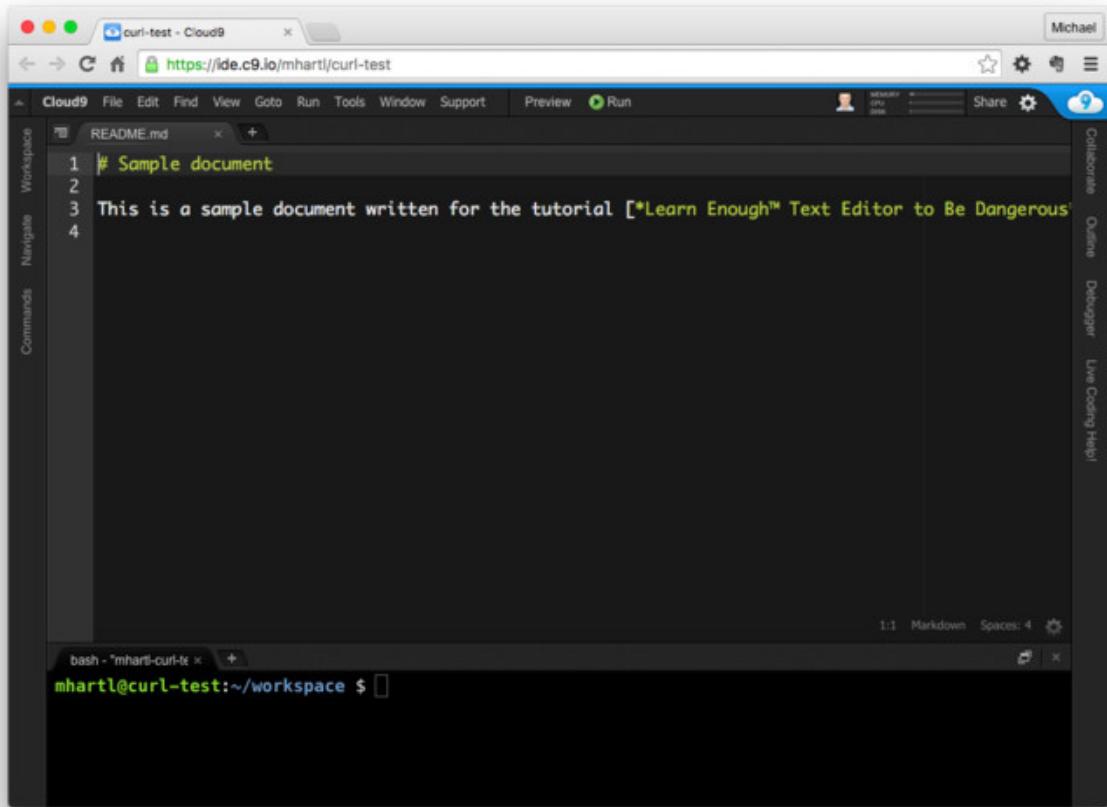


Figure 19: Cloud9 with word wrap off.

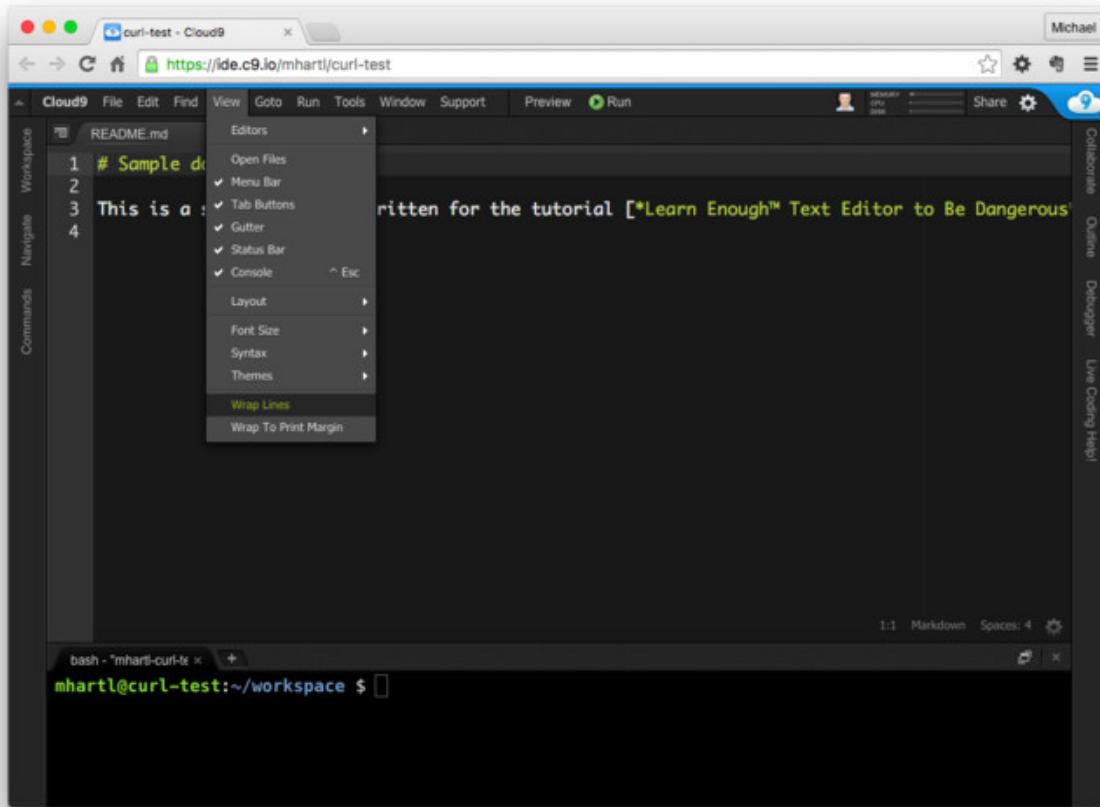


Figure 20: Turning word wrap on in Cloud9.

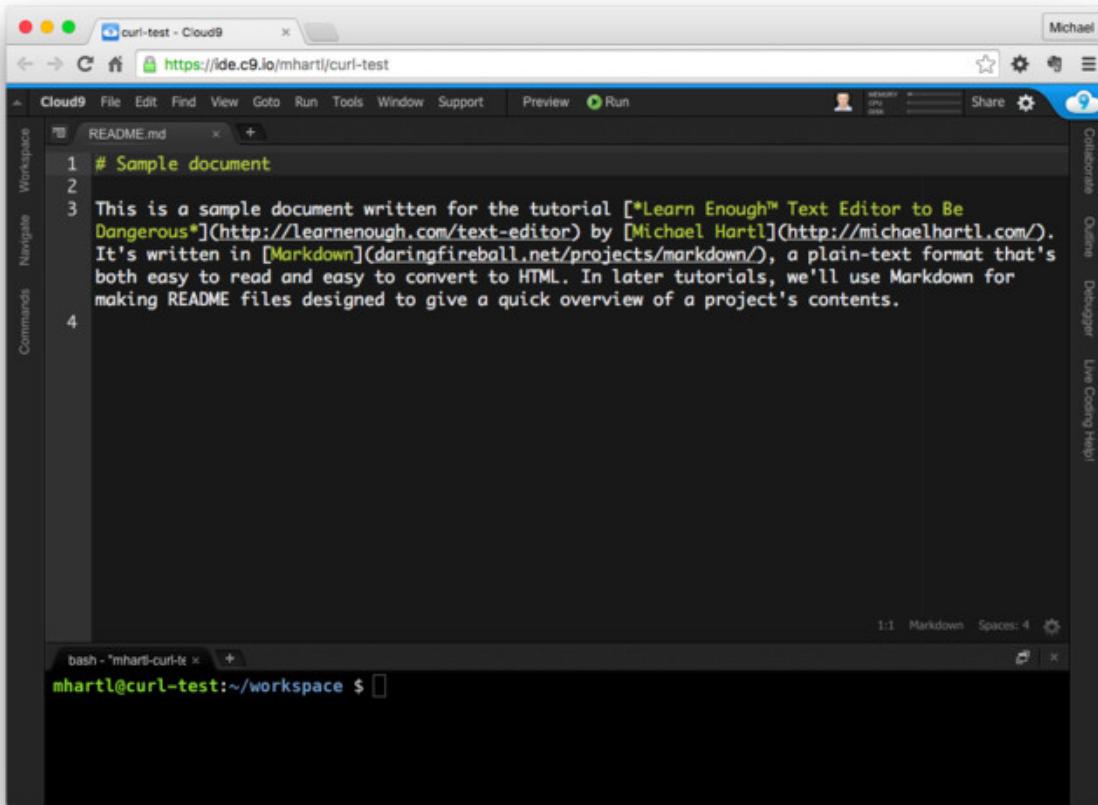


Figure 21: Cloud9 with word wrap on.

Syntax highlighting

One thing you may have noticed from inspecting [Figure 15](#) and [Figure 21](#) 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](#).

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 3](#)), in this case by clicking on the Help menu and searching for "Preview" ([Figure 22](#)). The result is a built-in package called Markdown Preview, which converts Markdown to HTML and shows the result, as seen in [Figure 23](#). In this context, it's convenient to work with an expanded width so that both the source and the preview are wide enough to view easily, as seen in [Figure 24](#). 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](#).

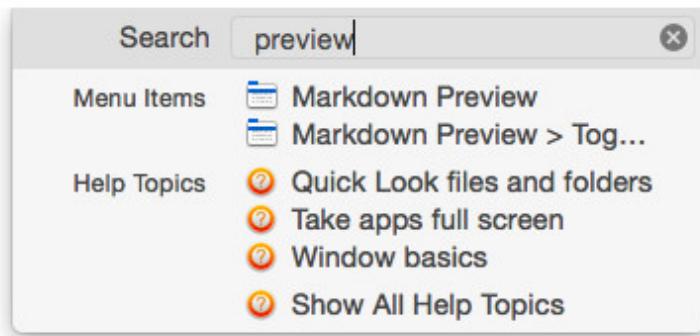


Figure 22: Using the Help menu to learn how to preview Markdown.

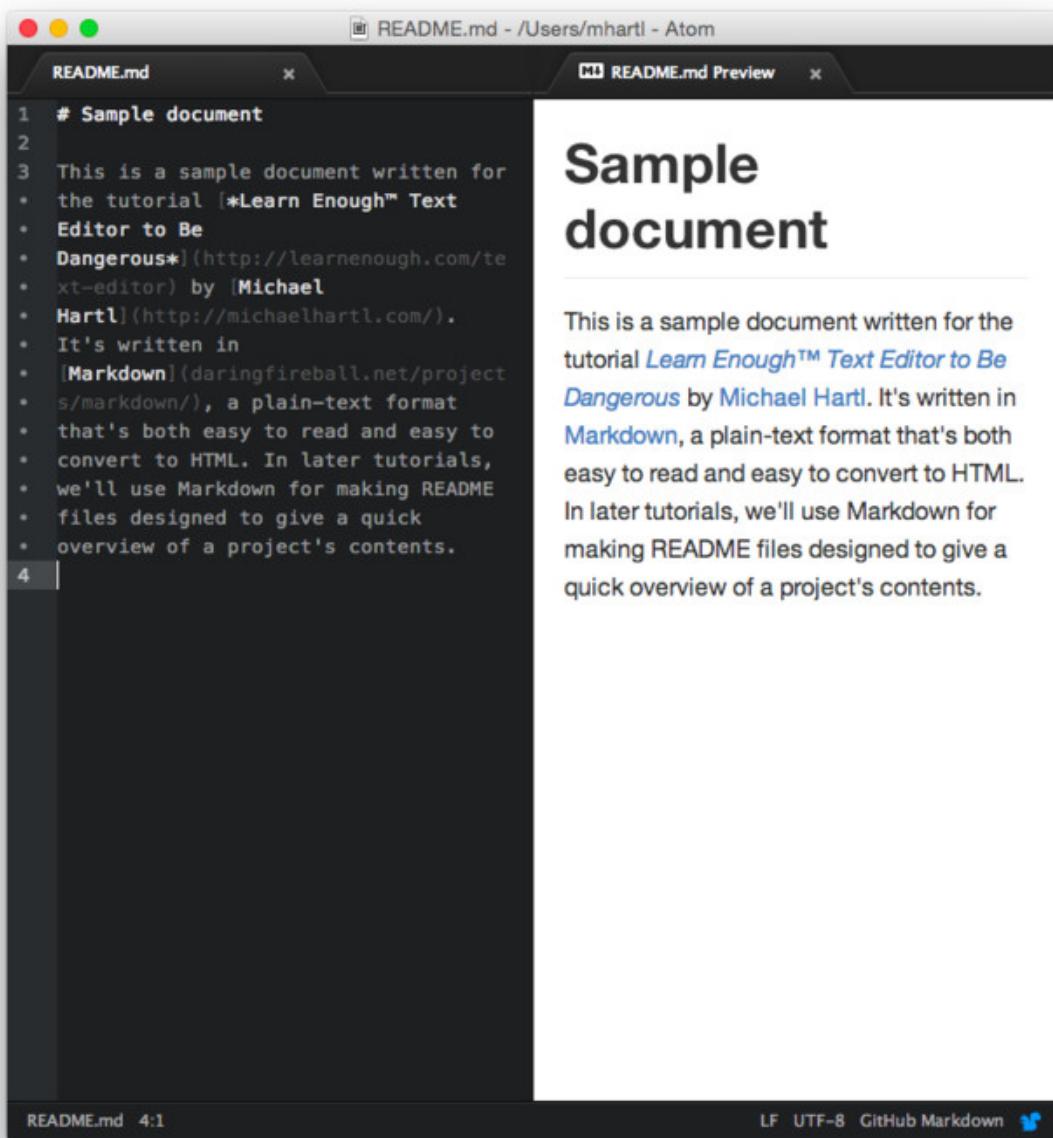


Figure 23: A Markdown preview in Atom.

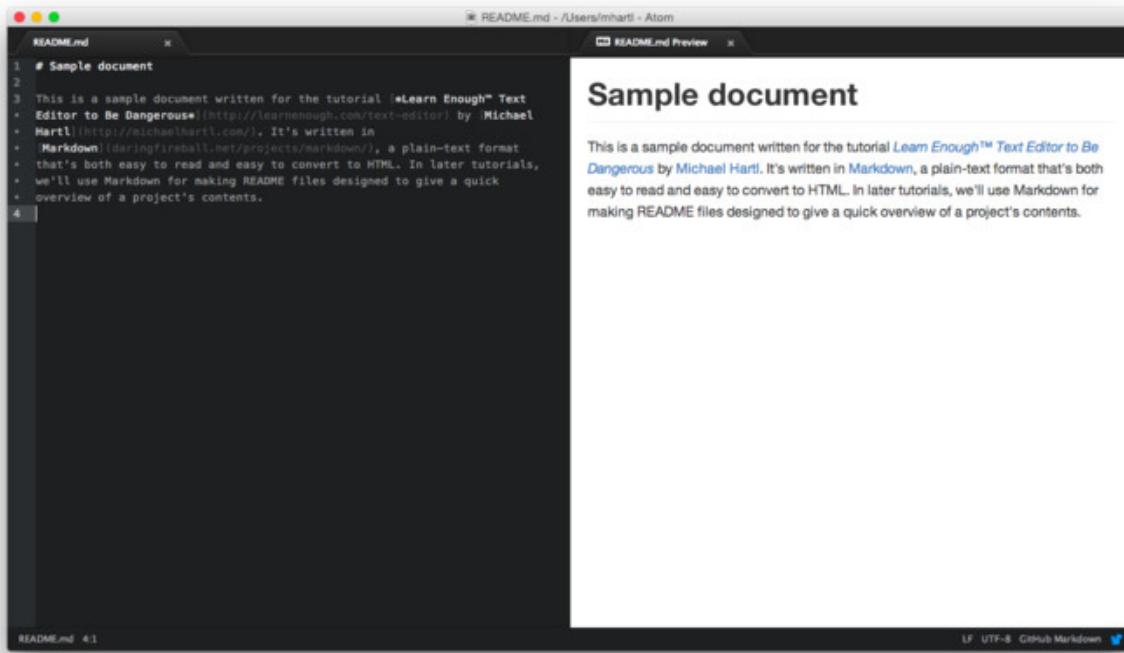


Figure 24: Using a wider window for the source and preview.

Exercises

1. By applying the methods in [Box 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 23](#)?
2. Open a new document called `lorem.txt` and fill it with the text shown in [Listing 8](#). Does the result have syntax highlighting?
3. Open a new document called `test.rb` and fill it with the text shown in [Listing 9](#). Does the result have syntax highlighting?

Listing 8: Some lorem ipsum text.

`~/lorem.txt`

```
  Lorem ipsum dolor sit amet
```

Listing 9: A test file.

```
~/test.rb
```

```
puts "test"
```

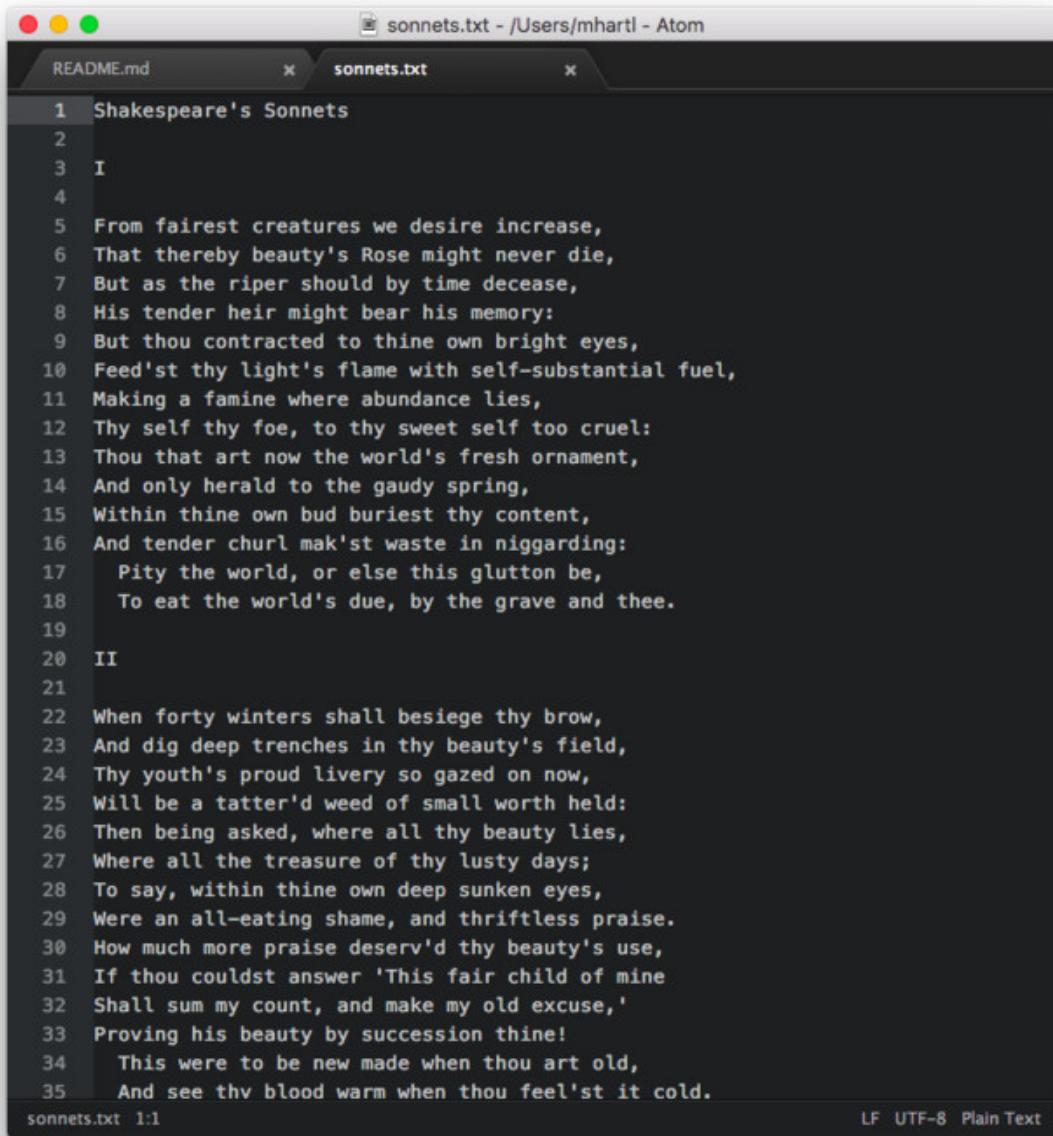
2.3 Moving

Unlike the commands for moving around in Vim ([Section 1](#), summarized in [Table 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 1.5](#) 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 7](#), and you should also verify that you're in the right directory.) The result appears in [Figure 25](#). Note that [Figure 25](#) 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](#).



The screenshot shows the Atom code editor interface. The title bar reads "sonnets.txt - /Users/mhartl - Atom". There are two tabs open: "README.md" and "sonnets.txt". The "sonnets.txt" tab is active, displaying the first 35 lines of Shakespeare's Sonnets. The text is in a monospaced font, with line numbers on the left. The content includes the opening sonnets, starting with "Shakespeare's Sonnets" and sections I and II. The bottom right corner of the editor window shows file statistics: "sonnets.txt 1:1" and encoding information "LF UTF-8 Plain Text".

```
1 Shakespeare's Sonnets
2
3 I
4
5 From fairest creatures we desire increase,
6 That thereby beauty's Rose might never die,
7 But as the riper should by time decease,
8 His tender heir might bear his memory:
9 But thou contracted to thine own bright eyes,
10 Feed'st thy light's flame with self-substantial fuel,
11 Making a famine where abundance lies,
12 Thy self thy foe, to thy sweet self too cruel:
13 Thou that art now the world's fresh ornament,
14 And only herald to the gaudy spring,
15 Within thine own bud buriest thy content,
16 And tender churl mak'st waste in niggarding:
17 Pity the world, or else this glutton be,
18 To eat the world's due, by the grave and thee.
19
20 II
21
22 When forty winters shall besiege thy brow,
23 And dig deep trenches in thy beauty's field,
24 Thy youth's proud livery so gazed on now,
25 Will be a tatter'd weed of small worth held:
26 Then being asked, where all thy beauty lies,
27 Where all the treasure of thy lusty days;
28 To say, within thine own deep sunken eyes,
29 Were an all-eating shame, and thriftless praise.
30 How much more praise deserv'd thy beauty's use,
31 If thou couldst answer 'This fair child of mine
32 Shall sum my count, and make my old excuse,'
33 Proving his beauty by succession thine!
34 This were to be new made when thou art old,
35 And see thy blood warm when thou feel'st it cold.
```

Figure 25: Opening Shakespeare's *Sonnets* in Atom.

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 subtle in Atom, so [Figure 26](#) shows the scrollbar for Sublime Text. [Figure 26](#) also shows the sort of two-pane view mentioned briefly in [Figure 24](#), which we'll discuss more in [Section 3.4](#).

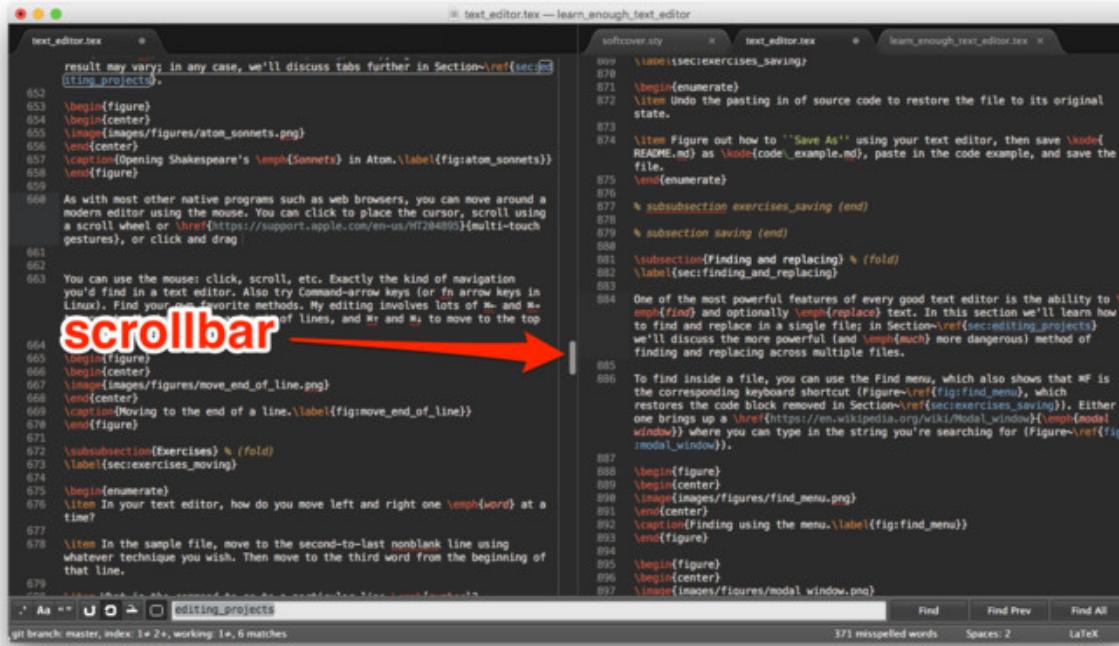
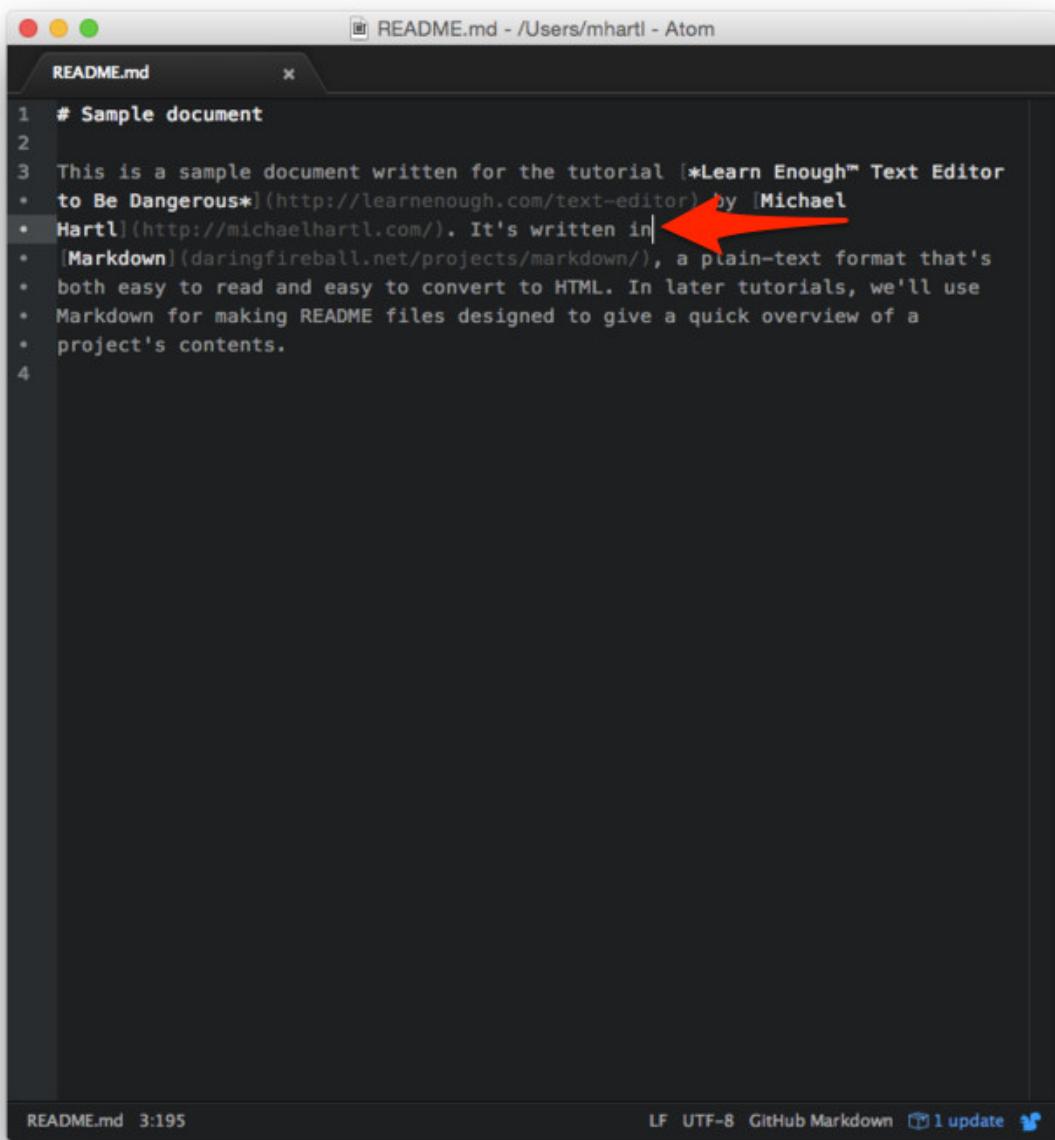


Figure 26: The Sublime Text scrollbar.

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](#)). (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 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 27](#), and an example of moving to the end of the file in [sonnets.txt](#) appears in [Figure 28](#).



A screenshot of the Atom text editor interface. The window title is "README.md - /Users/mhartl - Atom". The tab bar shows "README.md". The main editor area contains the following text:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
• Hartl](http://michaelhartl.com/). It's written in|
```

The cursor is positioned at the end of the line starting with "It's written in". A red arrow points from the left towards this cursor position. The status bar at the bottom of the editor shows "README.md 3:195" on the left and "LF UTF-8 GitHub Markdown 1 update" on the right.

Figure 27: Moving to the end of a line with $\mathbb{M} \rightarrow$.

```
sonnets.txt - Atom
sonnets.txt ×
2587
2588 CLIII
2589
2590 Cupid laid by his brand and fell asleep:
2591 A maid of Dian's this advantage found,
2592 And his love-kindling fire did quickly steep
2593 In a cold valley-fountain of that ground;
2594 Which borrow'd from this holy fire of Love,
2595 A dateless lively heat, still to endure,
2596 And grew a seething bath, which yet men prove
2597 Against strange maladies a sovereign cure.
2598 But at my mistress' eye Love's brand new-fired,
2599 The boy for trial needs would touch my breast;
2600 I, sick withal, the help of bath desired,
2601 And thither hied, a sad distemper'd guest,
2602 But found no cure, the bath for my help lies
2603 Where Cupid got new fire; my mistress' eyes.
2604
2605 CLIV
2606
2607 The little Love-god lying once asleep,
2608 Laid by his side his heart-inflaming brand,
2609 Whilst many nymphs that vow'd chaste life to keep
2610 Came tripping by; but in her maiden hand
2611 The fairest votary took up that fire
2612 Which many legions of true hearts had warm'd;
2613 And so the general of hot desire
2614 Was, sleeping, by a virgin hand disarm'd.
2615 This brand she quenched in a cool well by,
2616 Which from Love's fire took heat perpetual,
2617 Growing a bath and healthful remedy,
2618 For men diseas'd; but I, my mistress' thrall,
2619 Came thither for cure and this by that I prove,
2620 Like fire heats water, water cools not love.
2621
/Users/mhartl/sonnets.txt 2621:1 LF UTF-8 Plain Text
```

Figure 28: Moving to the end of the file with ⌘↓.

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 the **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 29](#). How do you get to the true beginning of the line?

sonnets.txt - /Users/mhartl - Atom

```
sonnets.txt x
2587
2588 CLIII
2589
2590 Cupid laid by his brand and fell asleep:
2591 A maid of Dian's this advantage found,
2592 And his love-kindling fire did quickly steep
2593 In a cold valley-fountain of that ground;
2594 Which borrow'd from this holy fire of Love,
2595 A dateless lively heat, still to endure,
2596 And grew a seething bath, which yet men prove
2597 Against strange maladies a sovereign cure.
2598 But at my mistress' eye
2599 The boy for trial need
2600 I, sick withal, the help of bath desired,
2601 And thither hied, a sad distemper'd guest,
2602 But found no cure, the foun
2603 Where Cupid got new
2604
2605 CLIV
2606
2607 The little Love-god lying once asleep,
2608 Laid by his side his heart-inflaming brand,
2609 Whilst many nymphs that vow'd chaste life to keep
2610 Came tripping by; but in her maiden hand
2611 The fairest votary took up that fire
2612 Which many legions of true hearts had warm'd;
2613 And so the general of hot desire
2614 Was, sleeping, by a virgin hand disarm'd.
2615 This brand she clenched in a cool well by,
2616 Which from Love's fire took heat perpetual,
2617 Growing health and healthful remedy,
2618 For me aseas'd; but I, my mistress' thrall,
2619 Call here for cure and this by that I prove,
2620 Love's fire heats water, water cools not love.
2621
```

sonnets.txt 2620:3

LF UTF-8 Plain Text

Figure 29: When using ⌘ ←, the cursor stops on whitespace.

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 30](#). Another closely related technique is to click on one location, and then Shift-click on another location to select all the text in between.

The screenshot shows a window titled "sonnets.txt - Atom". The file contains a list of sonnets from William Shakespeare. The text is in a monospaced font, with line numbers preceding each sonnet. The cursor is positioned over the word "lov'd" in the 1975 line of sonnet CXVI. The status bar at the bottom shows the file path "/Users/mhartl/sonnets.txt" and the file statistics "1975:39 (14,585)".

```
sonnets.txt  x
1953 Alas! why fearing of Time's tyranny,
1954 Might I not then say, 'Now I love you best,'  

1955 When I was certain o'er incertainty,
1956 Crowning the present, doubting of the rest?  

1957 Love is a babe, then might I not say so,  

1958 To give full growth to that which still doth grow?  

1959  

1960 CXVI  

1961  

1962 Let me not to the marriage of true minds  

1963 Admit impediments. Love is not love  

1964 Which alters when it alteration finds,  

1965 Or bends with the remover to remove:  

1966 O, no! it is an ever-fixed mark,  

1967 That looks on tempests and is never shaken;  

1968 It is the star to every wandering bark,  

1969 Whose worth's unknown, although his height be taken.  

1970 Love's not Time's fool, though rosy lips and cheeks  

1971 Within his bending sickle's compass come;  

1972 Love alters not with his brief hours and weeks,  

1973 But bears it out even to the edge of doom.  

1974 If this be error and upon me prov'd,  

1975 I never writ, nor no man ever lov'd.  

1976  

1977 CXVII  

1978  

1979 Accuse me thus: that I have scanted all,  

1980 Wherein I should your great deserts repay,  

1981 Forgot upon your dearest love to call,  

1982 Whereto all bonds do tie me day by day;  

1983 That I have frequent been with unknown minds,  

1984 And given to time your own dear-purchas'd right;  

1985 That I have hoisted sail to all the winds  

1986 Which should transport me farthest from your sight.
```

Figure 30: The result of clicking and dragging the mouse cursor.

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:

- Click and drag the mouse cursor over the word
- Double-click the word with the mouse
- Press ⌘D (system-dependent; see [Box 3](#))

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:

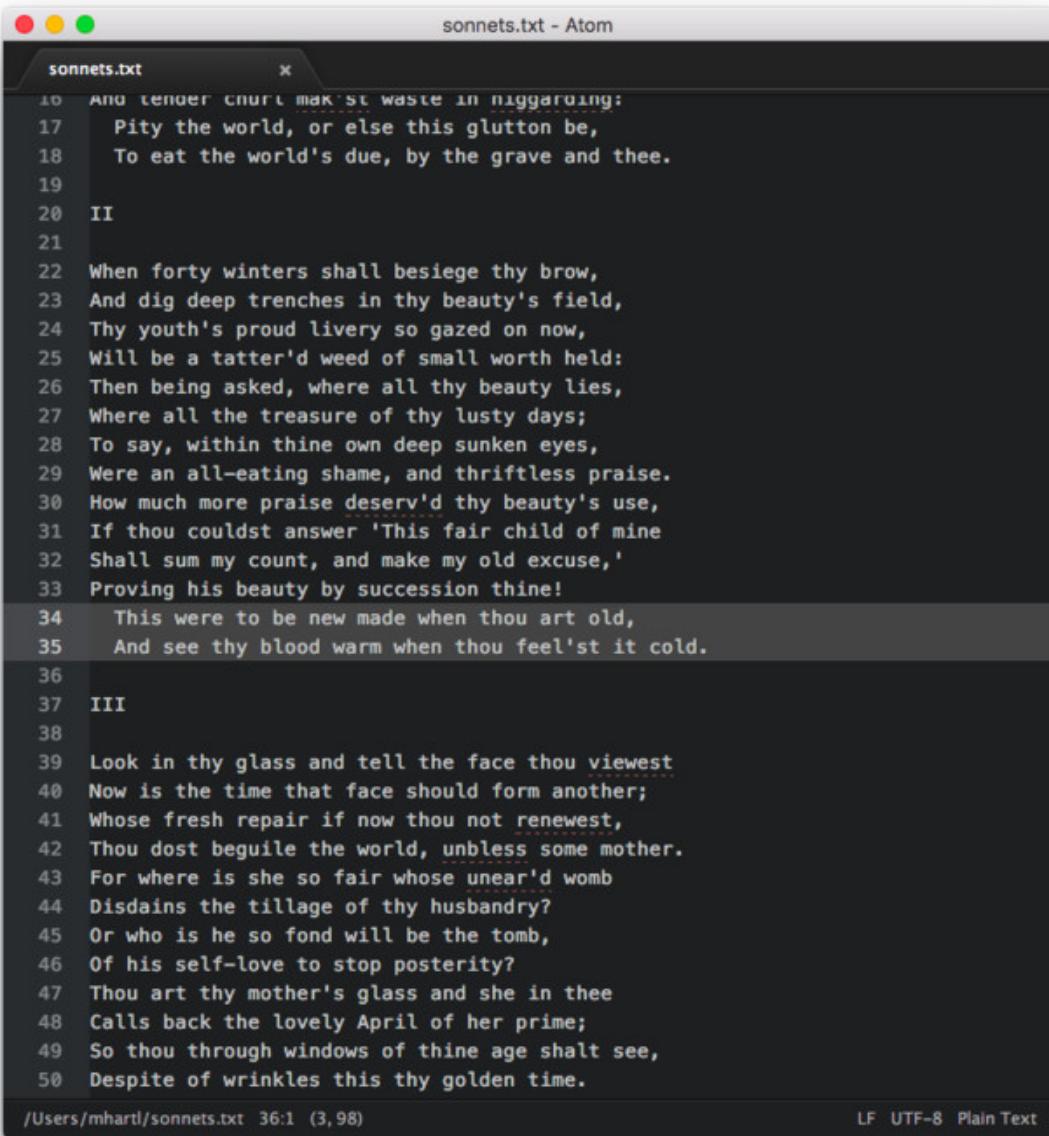
- Click the beginning of the line and drag the cursor to the end
- Click the end of the line and drag the cursor to the beginning
- Press ⌘← (twice) to get to the beginning of line, then press ↑ ⌘→ to select to the end of line
- Press ⌘→ to get to the end of line, then press ↑ ⌘← (twice) to select to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

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 $\mathbf{\mathbb{M} \leftarrow}$ to go to the beginning of the first line I want to select and then hitting $\mathbf{\mathbb{U} \downarrow}$ repeatedly until I've selected all the lines I want ([Figure 31](#)). (As noted in [Section 2.3.1](#), in many editors $\mathbf{\mathbb{M} \leftarrow}$ stops on whitespace, so moving to the beginning of the line actually requires two uses of $\mathbf{\mathbb{M} \leftarrow}$ in succession. Being able to figure out details and [edge cases](#) like this is a hallmark of growing technical sophistication ([Box 3](#)).



A screenshot of the Atom text editor showing the file "sonnets.txt". The window title is "sonnets.txt - Atom". The text in the editor is a collection of sonnets, specifically Shakespeare's Sonnets 10 through 50. The text is formatted with line numbers on the left and some sections labeled II and III. A selection is highlighted around Sonnet 36, which begins with "When forty winters shall besiege thy brow, And dig deep trenches in thy beauty's field, Thy youth's proud livery so gazed on now, Will be a tatter'd weed of small worth held: Then being asked, where all thy beauty lies, Where all the treasure of thy lusty days; To say, within thine own deep sunken eyes, Were an all-eating shame, and thriftless praise. How much more praise deserved thy beauty's use, If thou couldst answer 'This fair child of mine Shall sum my count, and make my old excuse,' Proving his beauty by succession thine! This were to be new made when thou art old, And see thy blood warm when thou feel'st it cold.

36
37 III
38
39 Look in thy glass and tell the face thou viewest
40 Now is the time that face should form another;
41 Whose fresh repair if now thou not renewest,
42 Thou dost beguile the world, unbless some mother.
43 For where is she so fair whose unear'd womb
44 Disdains the tillage of thy husbandry?
45 Or who is he so fond will be the tomb,
46 Of his self-love to stop posterity?
47 Thou art thy mother's glass and she in thee
48 Calls back the lovely April of her prime;
49 So thou through windows of thine age shalt see,
50 Despite of wrinkles this thy golden time.

Figure 31: Selecting a Shakespearean couplet using $\mathbb{H} \leftarrow$ and $\mathbb{U} \downarrow$.

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 32](#) shows the use of the Selection menu in Sublime Text, while [Figure 33](#) shows the use of the Edit menu in Atom.
- Press ⌘A

Note from [Figure 32](#) 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.

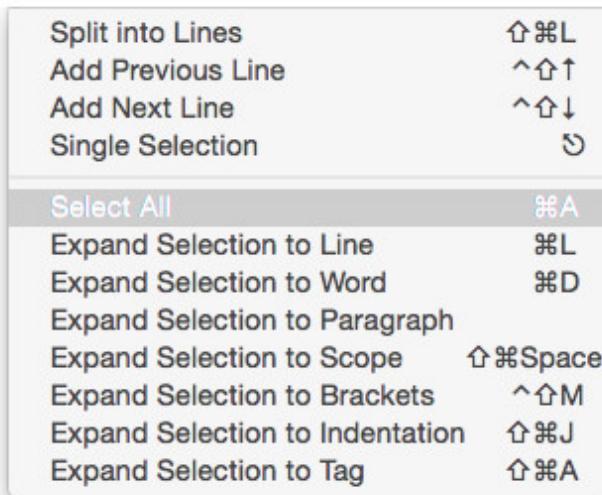


Figure 32: Selecting the entire document using the Selection menu (Sublime Text).

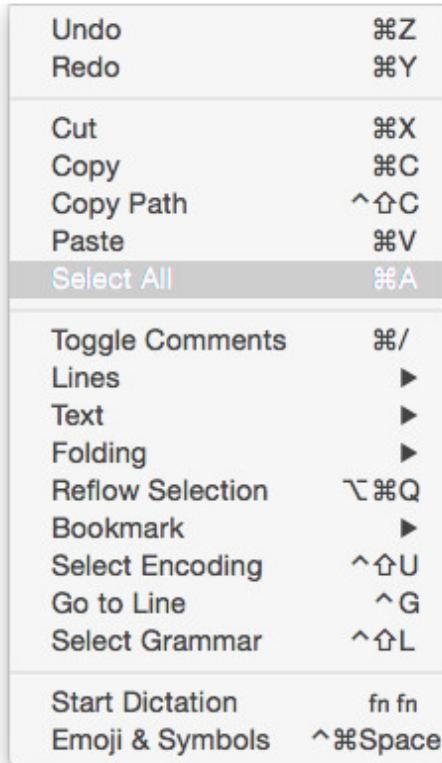


Figure 33: Selecting the entire document using the Edit menu (Atom).

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 34](#)), but the operations are so common that I strongly recommend 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 35](#)).

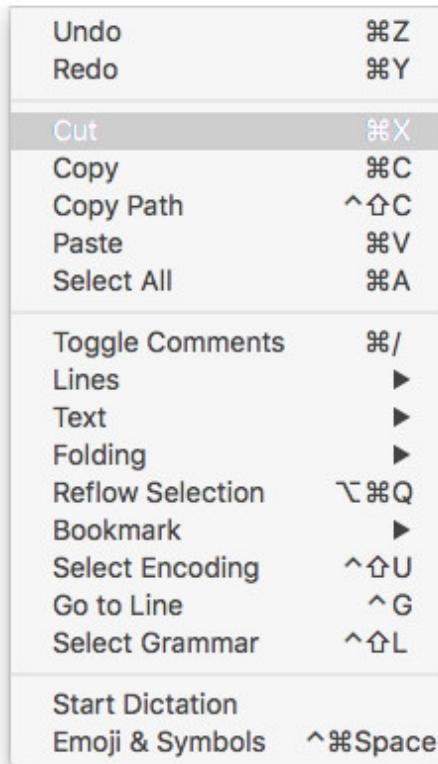


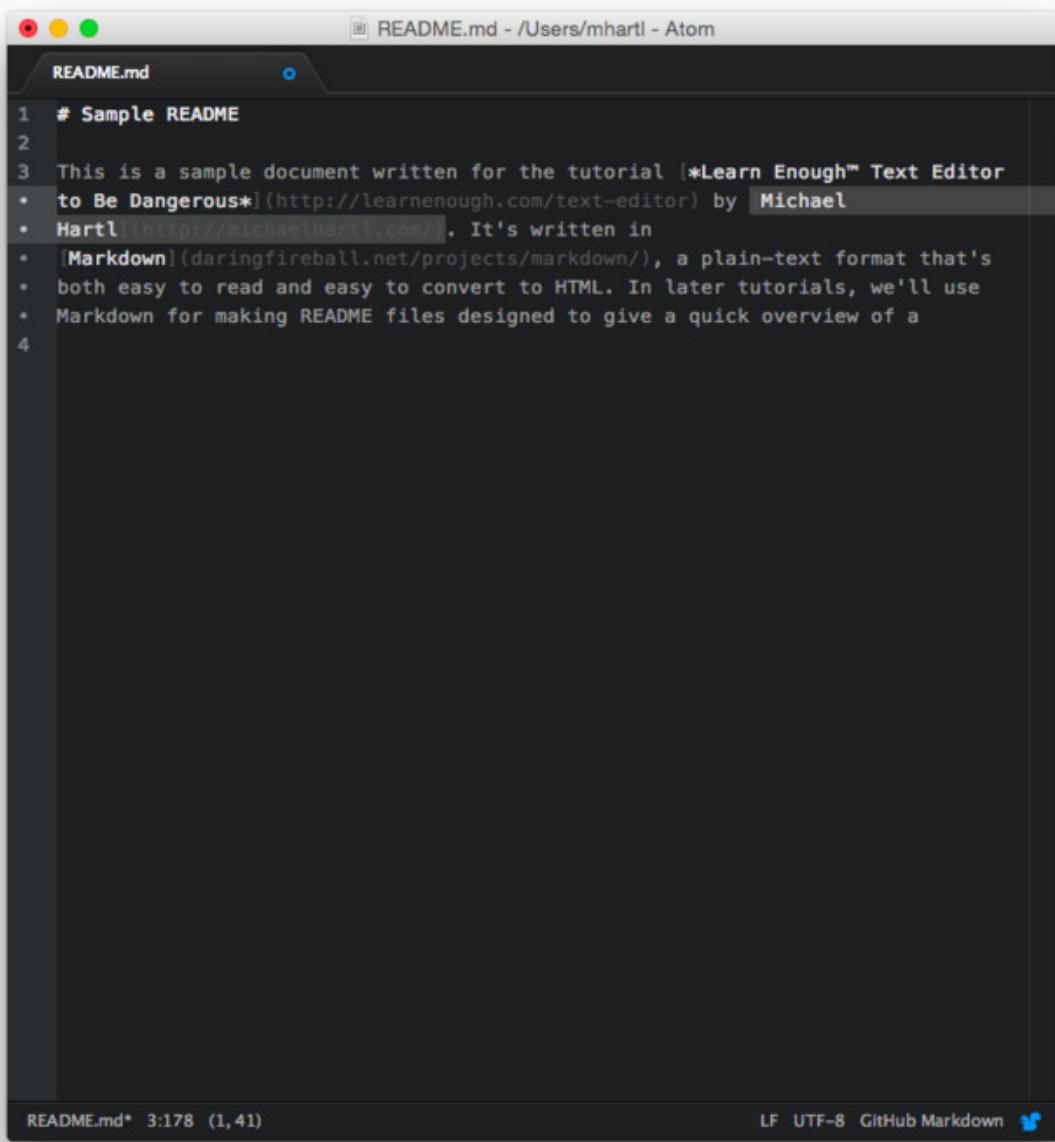
Figure 34: The Cut/Copy/Paste menu items (which you should never use).



Figure 35: The XCV keys on a standard QWERTY keyboard.

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 36](#). 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 37](#). Finally, [Figure 38](#) shows the result of cutting `README` from the main text and pasting it in at the end of the file.



A screenshot of the Atom text editor interface. The window title is "README.md - /Users/mhartl - Atom". The file content is a sample README document. A portion of the text is selected, specifically the line starting with "•". The status bar at the bottom shows "README.md* 3:178 (1,41)" on the left and "LF UTF-8 GitHub Markdown" on the right.

```
1 # Sample README
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by Michael
• Hartl (http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making README files designed to give a quick overview of a
4
```

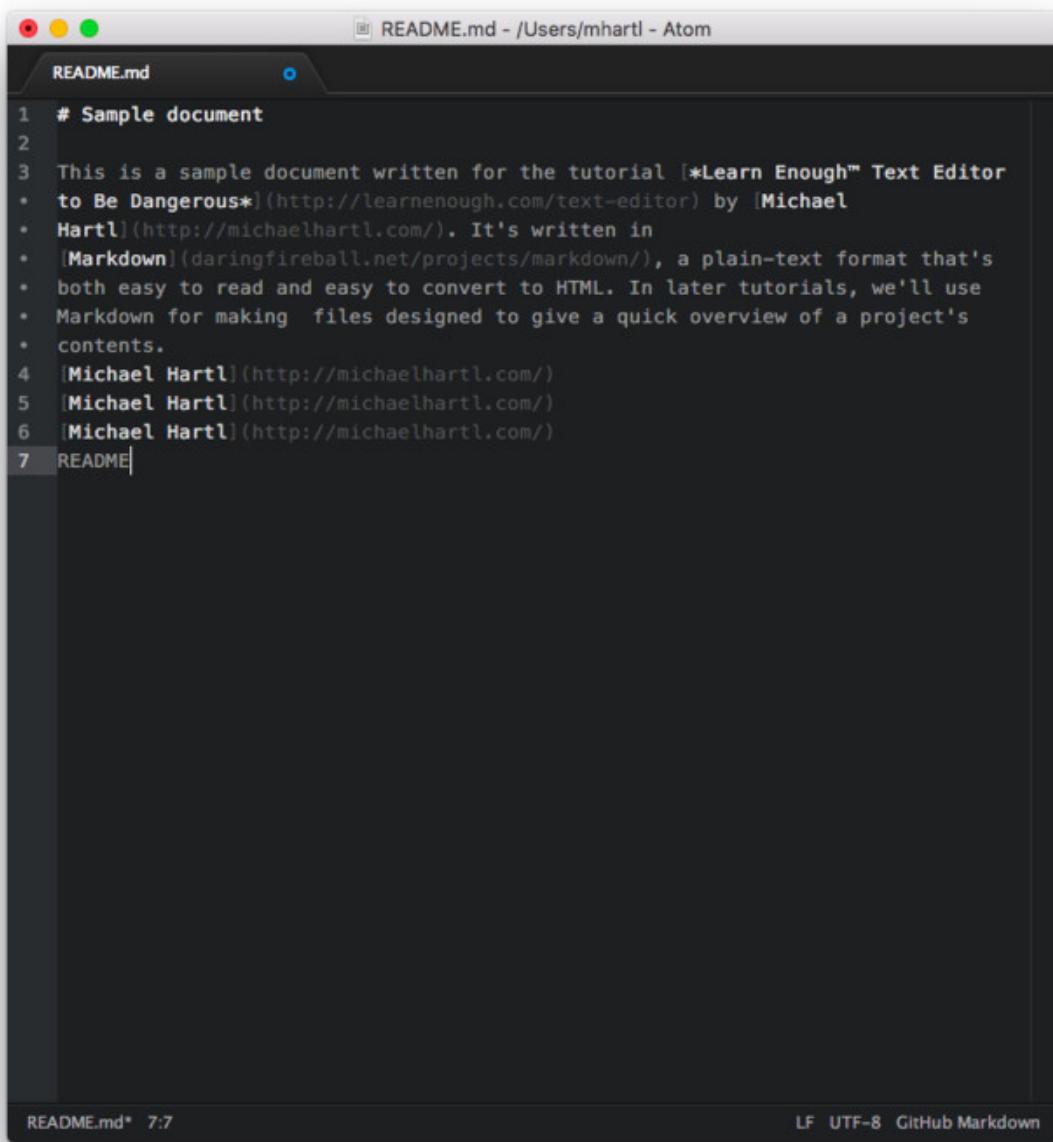
Figure 36: Selecting a Markdown link

A screenshot of the Atom text editor interface. The window title is "README.md - /Users/mhartl - Atom". The file name "README.md" is visible in the tab bar. The main editor area contains the following text:

```
1 # Sample README
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
4 * to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
5 Hartl](http://michaelhartl.com/). It's written in
6 [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
7 both easy to read and easy to convert to HTML. In later tutorials, we'll use
8 Markdown for making README files designed to give a quick overview of a
9 [Michael Hartl](http://michaelhartl.com/)
10 [Michael Hartl](http://michaelhartl.com/)
11 [Michael Hartl](http://michaelhartl.com/)
```

The status bar at the bottom shows "README.md* 6:42" on the left and "LF UTF-8 GitHub Markdown" on the right, along with a small icon.

Figure 37: Pasting link text several times (with returns in between).



A screenshot of the Atom text editor interface. The window title is "README.md ~ /Users/mhartl - Atom". The tab bar shows "README.md". The main editor area contains the following Markdown content:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
* to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
* Hartl](http://michaelhartl.com/). It's written in
* [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
* both easy to read and easy to convert to HTML. In later tutorials, we'll use
* Markdown for making files designed to give a quick overview of a project's
* contents.
4 [Michael Hartl](http://michaelhartl.com/)
5 [Michael Hartl](http://michaelhartl.com/)
6 [Michael Hartl](http://michaelhartl.com/)
7 README|
```

The status bar at the bottom left shows "README.md* 7:7" and the right side shows "LF UTF-8 GitHub Markdown".

Figure 38: The result of cutting “README” and pasting at the end of the file.

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 39](#)) or the keyboard shortcuts $\text{^}\text{ }\text{\texttildelow}\text{V}$ (cycle forward) and $\text{\texttildelow}\text{ }\text{^}\text{ }\text{\texttildelow}\text{V}$ (cycle backward). I use Jumpcut dozens or even hundreds of times a day, and I strongly suggest giving it a try.

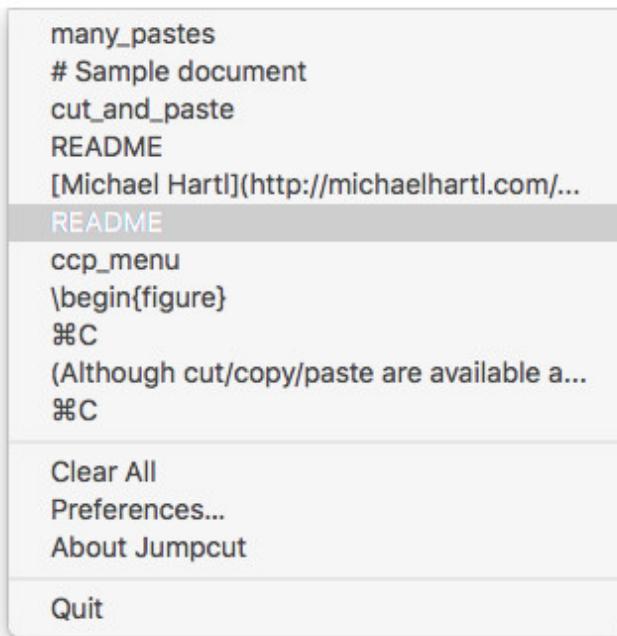
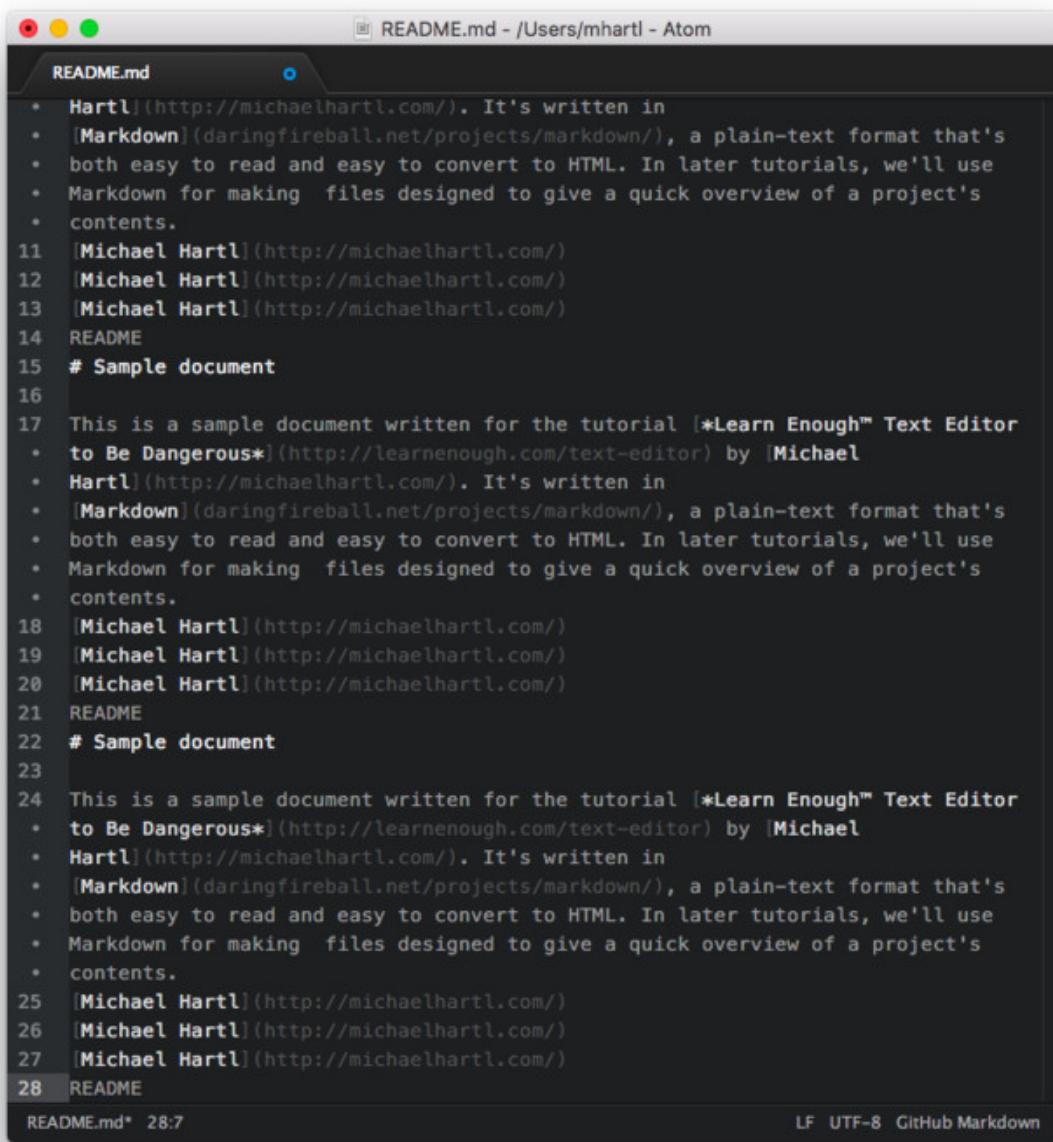


Figure 39: Jumpcut expands the copy-and-paste buffer to include a longer history.

Exercises

1. Select the entire document, Copy it, and Paste several times. The result should look something like [Figure 40](#).
2. Select the entire document and Cut it. Why might this be preferable to deleting it?
3. Select 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?



The screenshot shows the Atom text editor window with the file 'README.md' open. The title bar indicates the file path is '/Users/mhartl - Atom'. The code editor displays three identical sections of text, each consisting of a list item, a link to Michael Hartl's website, and a line of text. The sections are numbered 11, 17, and 24. The text is as follows:

```
• [Hartl](http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making files designed to give a quick overview of a project's
• contents.
11 [Michael Hartl](http://michaelhartl.com/)
12 [Michael Hartl](http://michaelhartl.com/)
13 [Michael Hartl](http://michaelhartl.com/)
14 README
15 # Sample document
16
17 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
• Hartl](http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making files designed to give a quick overview of a project's
• contents.
18 [Michael Hartl](http://michaelhartl.com/)
19 [Michael Hartl](http://michaelhartl.com/)
20 [Michael Hartl](http://michaelhartl.com/)
21 README
22 # Sample document
23
24 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
• Hartl](http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making files designed to give a quick overview of a project's
• contents.
25 [Michael Hartl](http://michaelhartl.com/)
26 [Michael Hartl](http://michaelhartl.com/)
27 [Michael Hartl](http://michaelhartl.com/)
28 README
```

The status bar at the bottom shows 'README.md* 28:7' on the left and 'LF UTF-8 GitHub Markdown' on the right.

Figure 40: The result of pasting the whole document several times.

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 $\leftarrow \times$ ([Table 2](#)). As with Cut/Copy/Paste ([Section 2.5](#)), deletion is especially useful when combined with the selection techniques from [Section 2.4](#).

In addition to the obvious technique of selecting and deleting text, on a Mac I especially like using $\text{⌘}\leftarrow \times$ 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 \times repeatedly, as context-switching to use $\text{⌘}\leftarrow \times$ 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 $\text{⌘}Z$ or $\text{^}Z$. Its inverse, Redo, is usually something like $\text{↑ ⌘}Z$ or $\text{⌘}Y$. 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.



Figure 41: Undo and Redo in the editor menu.

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 ($\text{⌘Z}/\text{↑ ⌘Z}$ or $\text{⌘Z}/\text{⌘Y}$), which is guaranteed to find the cursor without making any undesired changes.

Exercises

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

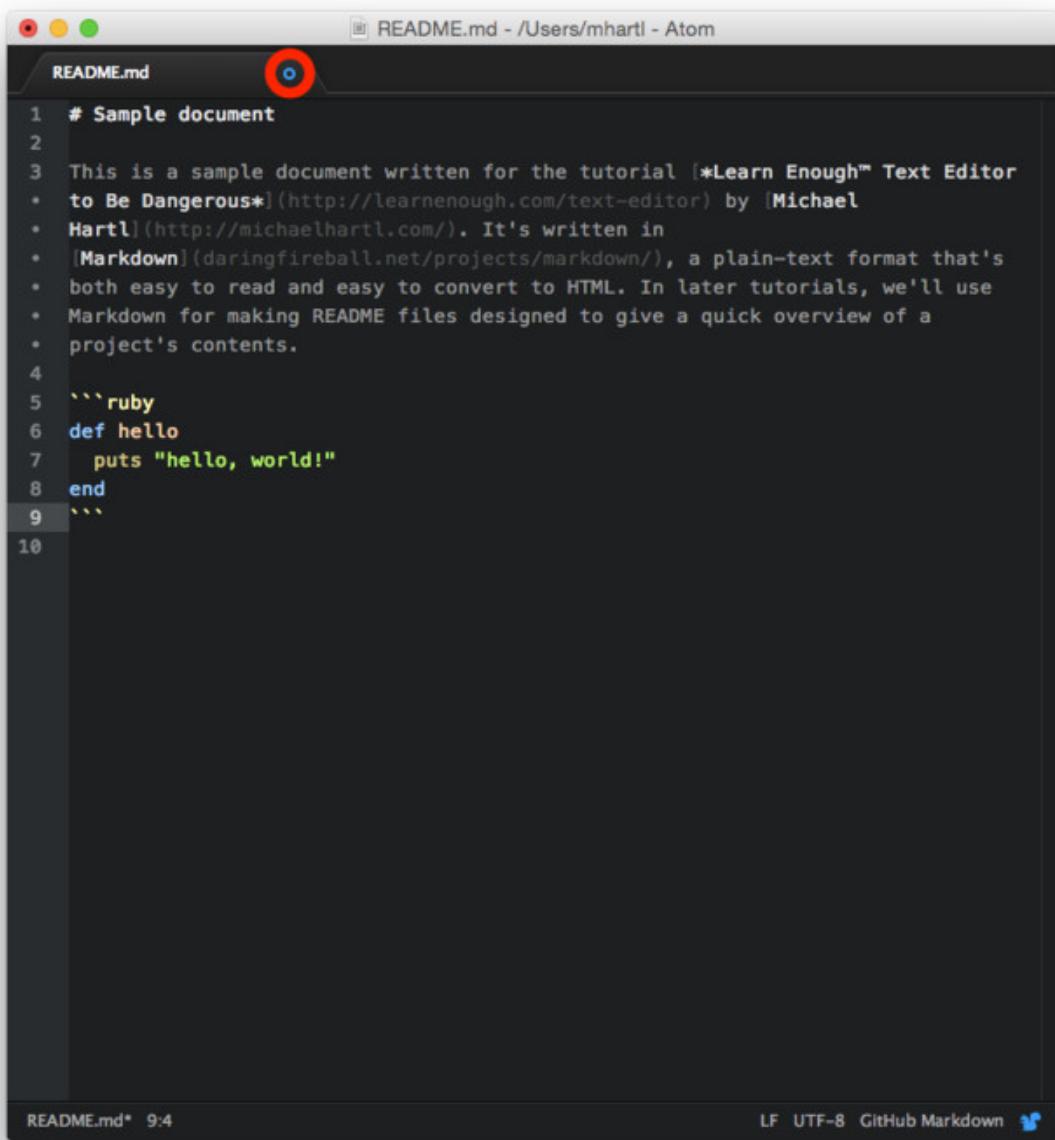
Once we've made some edits to a file, we can save it using the menu or with ⌘S . I strongly recommend using the keyboard shortcut, which among other things makes it easier to save the file whenever you reach a temporary pause in your writing or coding—a valuable habit to cultivate. Basically, if you're 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 10](#), as shown in [Figure 42](#) (which includes some nice high-contrast syntax highlighting). As you can see from the circled indicator in [Figure 42](#), 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 43](#)).

Listing 10: A code snippet.

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

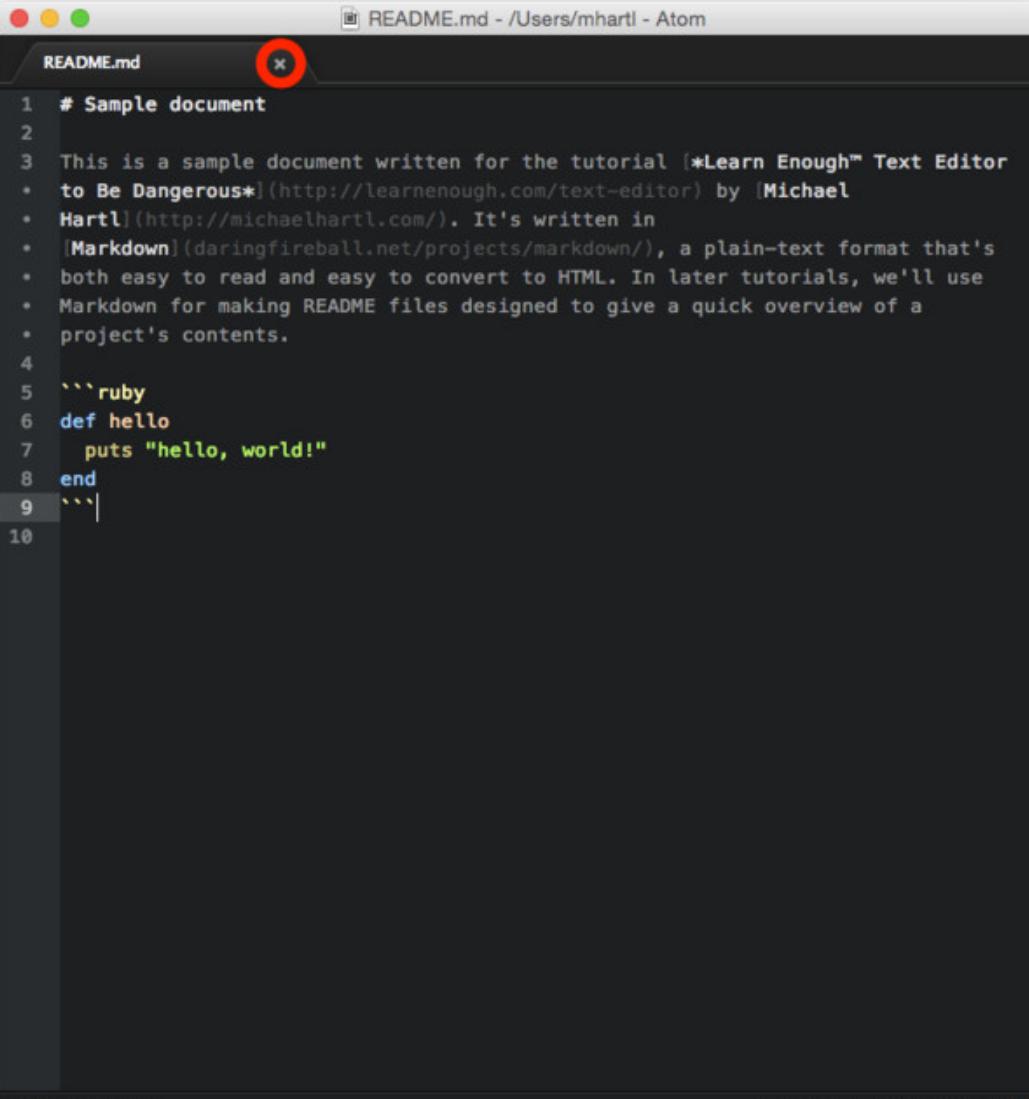


A screenshot of the Atom text editor interface. The window title is "README.md - /Users/mhartl - Atom". The main editor area displays the following content:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
4 * to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
5 * Hartl](http://michaelhartl.com/). It's written in
6 * [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
7 * both easy to read and easy to convert to HTML. In later tutorials, we'll use
8 * Markdown for making README files designed to give a quick overview of a
9 * project's contents.
10
```

The status bar at the bottom shows "README.md* 9:4" on the left and "LF UTF-8 GitHub Markdown" on the right, along with a small GitHub icon.

Figure 42: An unsaved file.



The screenshot shows the Atom text editor interface. The title bar reads "README.md - /Users/mhartl - Atom". The main window displays the content of a README.md file. The file contains the following text:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
• Hartl](http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making README files designed to give a quick overview of a
• project's contents.
4
5 ```ruby
6 def hello
7   puts "hello, world!"
8 end
9 ```
10
```

A red circle highlights the close button (X) in the top right corner of the Atom window.

Figure 43: The file from [Figure 42](#) after saving.

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 11](#), but I prefer the more compact prompt shown in [Listing 12](#). 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 13](#). Source the Bash profile as in [Listing 5](#) and confirm that the prompt on your system matches the one shown in [Listing 12](#).

Listing 11: The default terminal prompt on my system.

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

Listing 12: My preferred, more compact prompt.

```
[~]$
```

Listing 13: The Bash lines needed to customize the prompt as shown in [Listing 12](#).

```
~/.bash_profile

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

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 44](#), which also shows that \mathcal{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 45](#)).

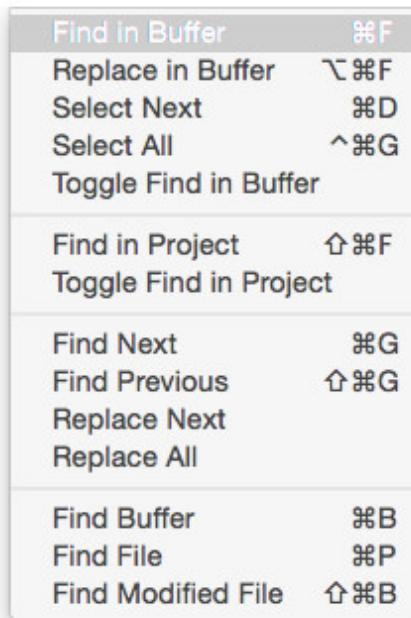


Figure 44: Finding using the menu.

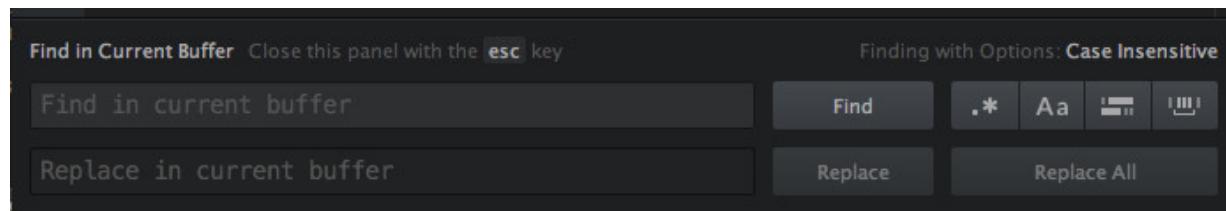


Figure 45: A modal window for finding and replacing.

For example, suppose we search for the string “sample”. As seen in [Figure 46](#), 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).

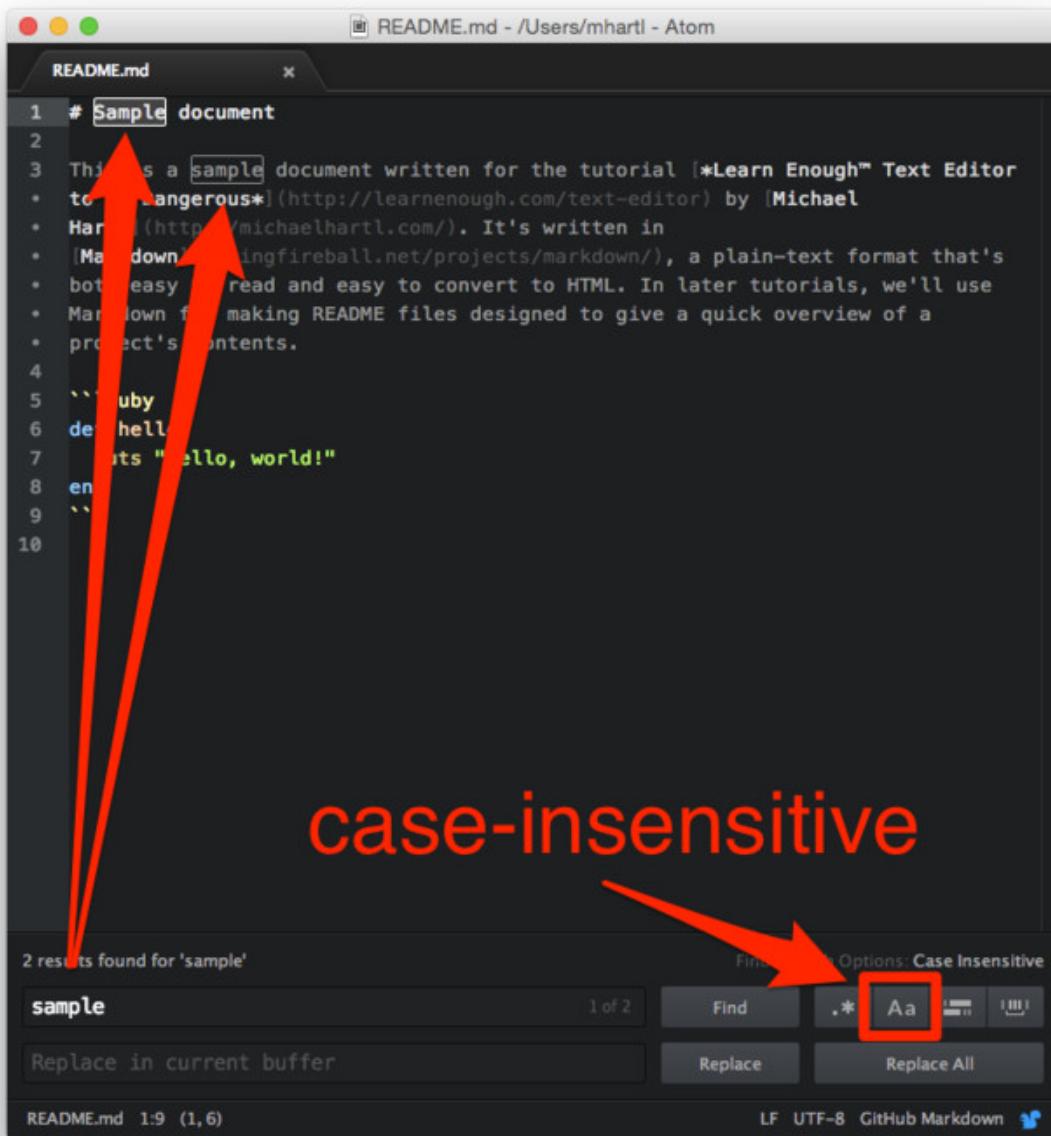


Figure 46: Finding the string “sample”.

[Figure 47](#) 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 48](#)). (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](#).)

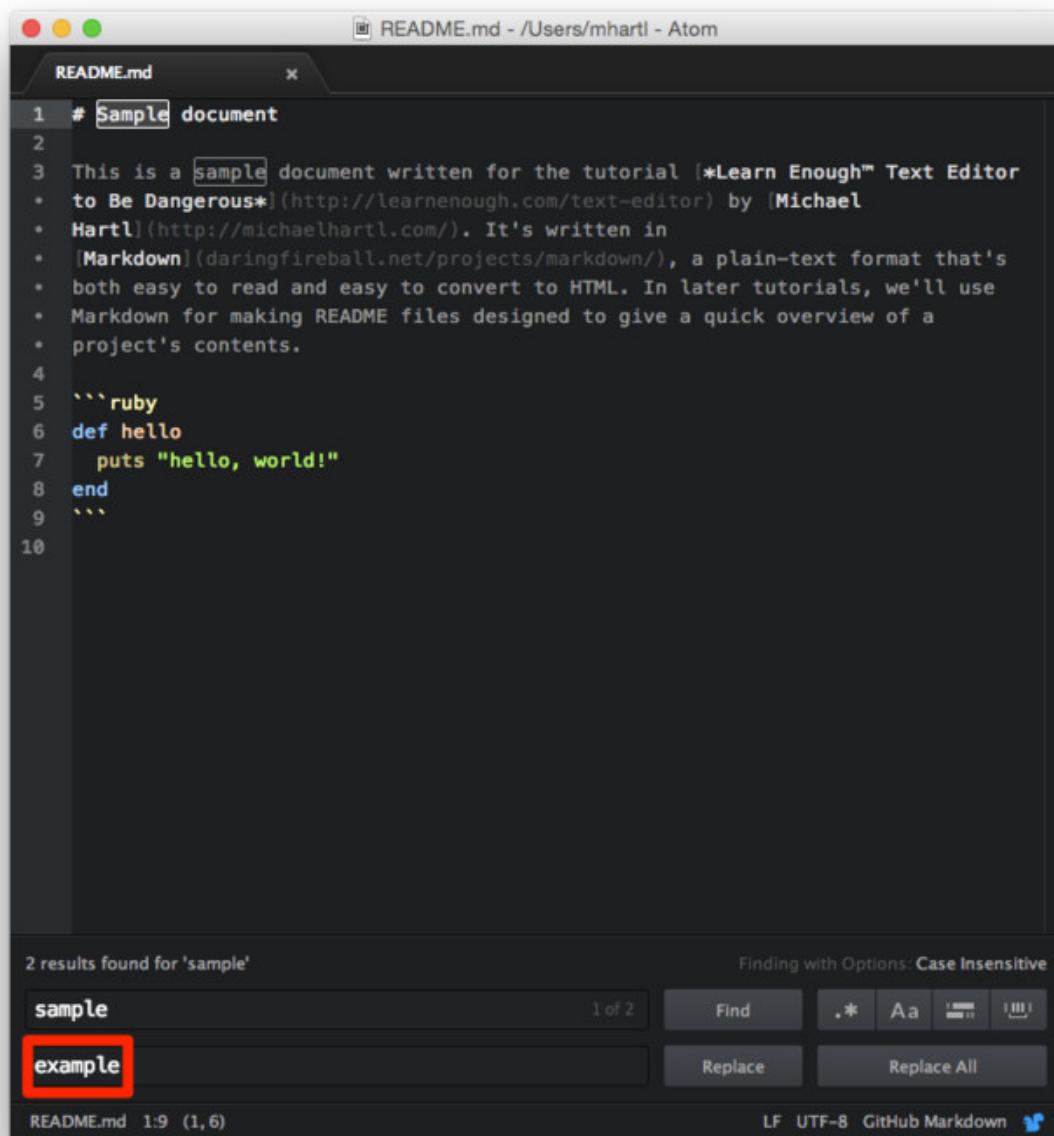


Figure 47: Finding and replacing.

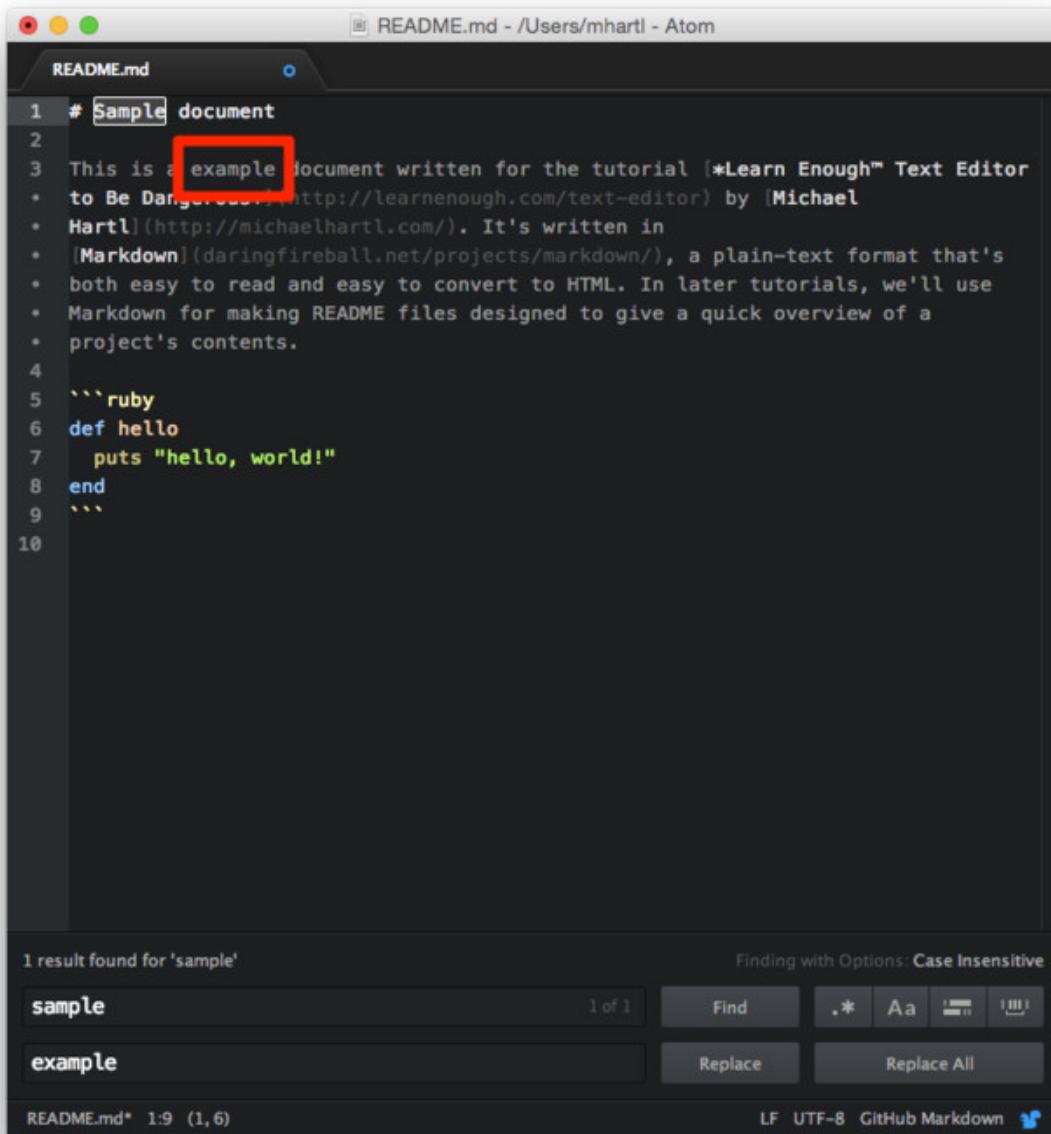


Figure 48: The result of replacing “sample” with “example”.

As seen in [Figure 44](#), you can also type ⌘G to find the next match using a keyboard shortcut. This $\text{⌘F}/\text{⌘G}$ combination also works in many other applications, such as word processors and web browsers.

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 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 49](#)).²⁰



Figure 49: Undo can save your bacon.

Important commands from this section are summarized in [Table 3](#).

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

Table 3: Important commands from [Section 2.9](#).

3 Advanced text editing

Having covered the basic functions of modern text editors in [Section 2](#), in this section we'll learn about a few of the most common advanced topics. Even more than in [Section 2](#), details will vary based on the exact editor you choose, so use your growing technical sophistication ([Box 3](#)) 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.

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 50](#).

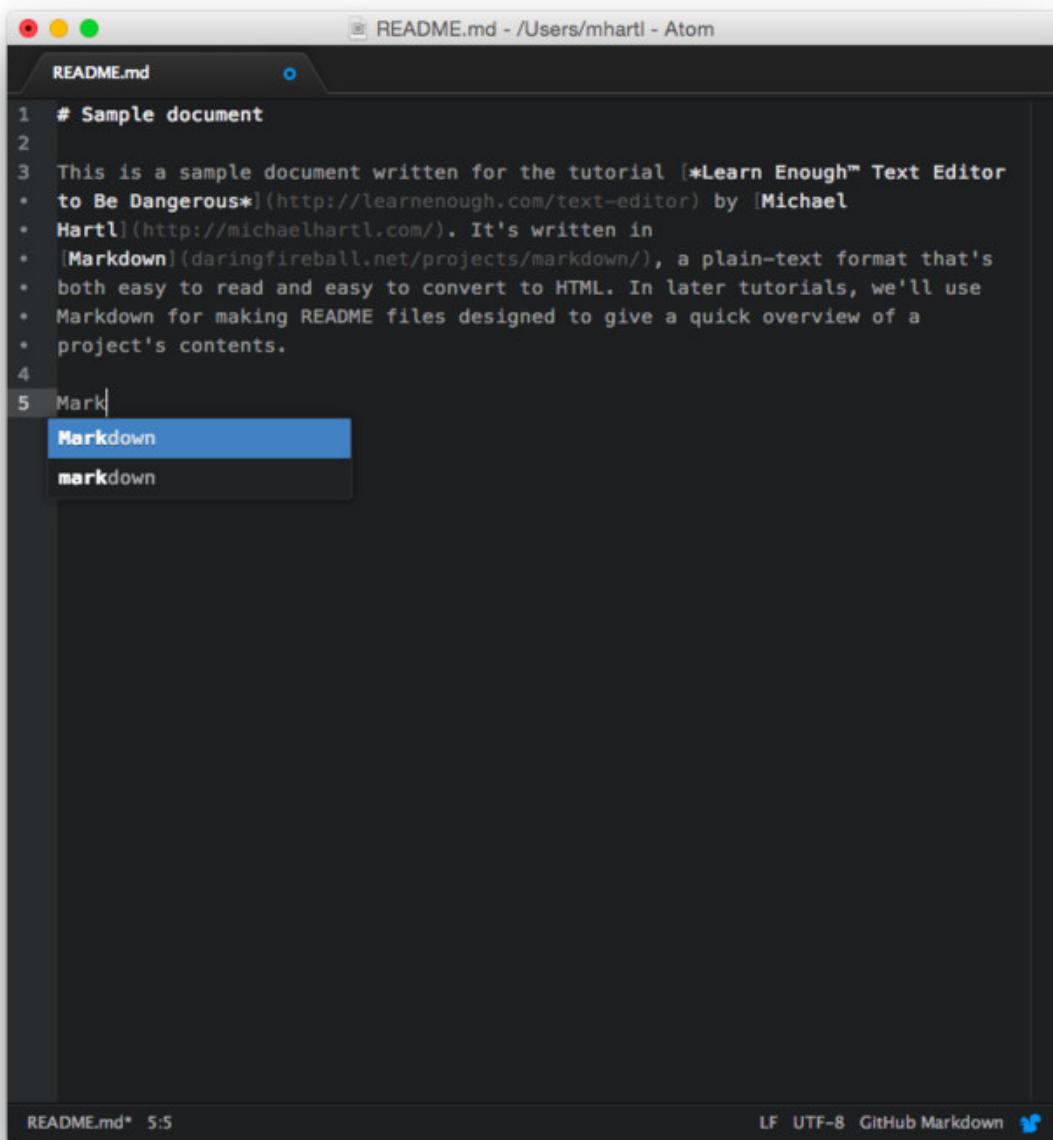


Figure 50: Autocomplete for “Markdown”.

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 L^AT_EX) 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 3](#), whose source looks like [Listing 14](#).

Listing 14: A cross-reference with a label I usually autocomplete.

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

When writing the string `technical_sophistication` in [Listing 14](#), I nearly always use autocomplete instead of typing it out in full.²¹ (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:

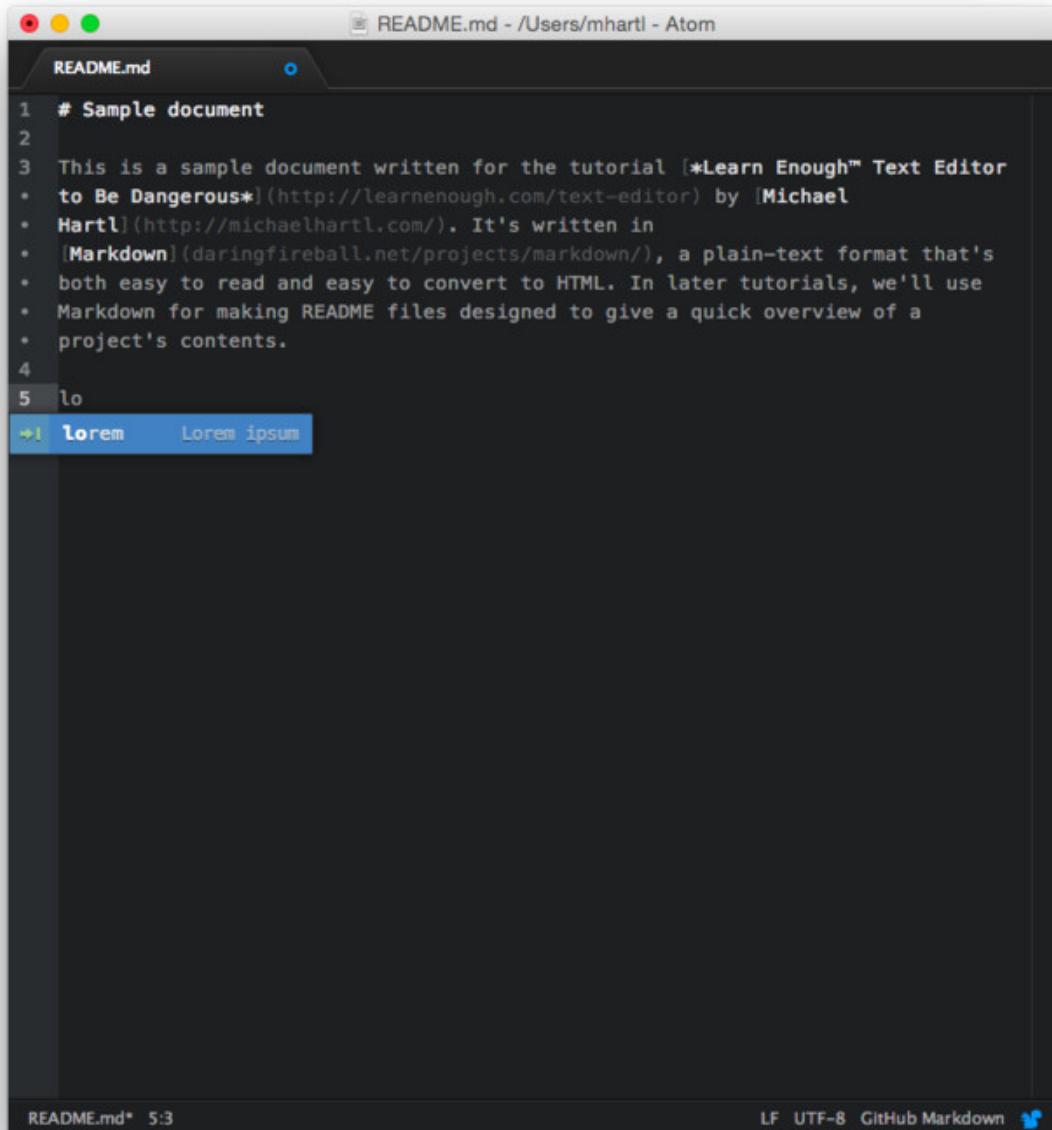
```
ReallyLongClassName < ReallyLongBaseClassName
```

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

Tab triggers

Tab triggers are similar to autocomplete 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 pre-defined with the editor, with the exact triggers typically based on the particular type of document we're editing. For example, in Markdown and other markup files (HTML, L^AT_EX, etc.), typing `lorem→` or `lo→` yields so-called *lorem ipsum* text, a Latin fragment from [Cicero](#) that is [often used as dummy text](#) in programming and design. We saw *lorem ipsum* briefly before in [Listing 8](#); a second example appears in [Figure 51](#), which shows the

result of typing `lo` in Atom. A closeup appears in [Figure 52](#). After hitting → to invoke the tab trigger, the full *lorem ipsum* text appears as in [Figure 53](#).



The screenshot shows the Atom text editor interface. The window title is "README.md - /Users/mhartl - Atom". The file content is a sample Markdown document. Line 5 contains the text "lo". Below the text area, a tab completion dropdown is open, showing two suggestions: "lorem" and "Lorem ipsum". The "lorem" suggestion is highlighted with a blue selection bar. At the bottom of the screen, the status bar displays "README.md* 5:3" on the left and "LF UTF-8 GitHub Markdown" along with a small icon on the right.

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
4 * to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
5 Hartl](http://michaelhartl.com/). It's written in
6 [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
7 both easy to read and easy to convert to HTML. In later tutorials, we'll use
8 Markdown for making README files designed to give a quick overview of a
9 project's contents.
10
11 lo
12 +1 lorem    Lorem ipsum
```

Figure 51: Typing “lo” in Atom prepares to activate a tab trigger.

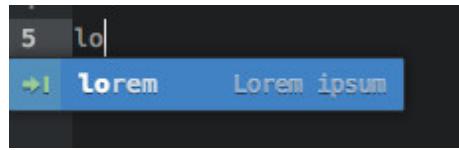
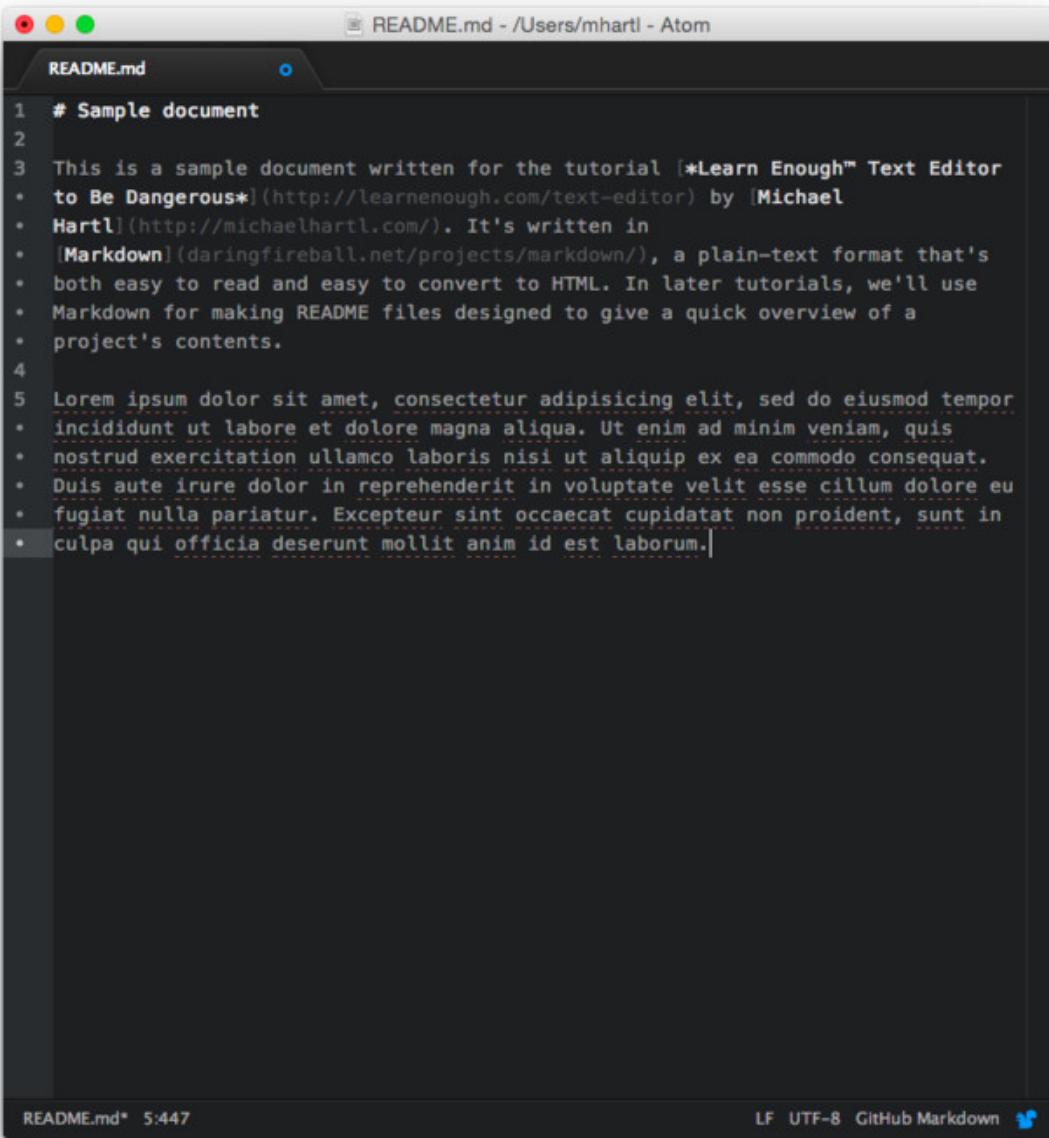


Figure 52: A more detailed view of the trigger in [Figure 51](#).



The screenshot shows the Atom text editor interface. The title bar reads "README.md ~ /Users/mhartl - Atom". The main editor area displays a sample README.md file with the following content:

```
1 # Sample document
2
3 This is a sample document written for the tutorial [*Learn Enough™ Text Editor
• to Be Dangerous*](http://learnenough.com/text-editor) by [Michael
• Hartl](http://michaelhartl.com/). It's written in
• [Markdown](daringfireball.net/projects/markdown/), a plain-text format that's
• both easy to read and easy to convert to HTML. In later tutorials, we'll use
• Markdown for making README files designed to give a quick overview of a
• project's contents.
4
5 Lorem ipsum dolor sit amet, consectetur adipisicing elit, sed do eiusmod tempor
• incididunt ut labore et dolore magna aliqua. Ut enim ad minim veniam, quis
• nostrud exercitation ullamco laboris nisi ut aliquip ex ea commodo consequat.
• Duis aute irure dolor in reprehenderit in voluptate velit esse cillum dolore eu
• fugiat nulla pariatur. Excepteur sint occaecat cupidatat non proident, sunt in
• culpa qui officia deserunt mollit anim id est laborum.|
```

The bottom status bar shows "README.md* 5:447" and icons for LF, UTF-8, GitHub Markdown, and a Twitter icon.

Figure 53: The result of the tab trigger in [Figure 51](#).

Tab triggers are especially useful when editing source code. For instance, 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 [cognitive load](#) of programming. We'll learn more about using tab triggers in [*Learn Enough™ Ruby to Be Dangerous*](#).

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 14](#), instead of typing

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

by hand I used the custom tab trigger `bref` (for “box reference”). (Of course, I then completed the part of the label after `aside:` using autocomplete.) Custom tab triggers are beyond the scope of this tutorial, but some hints about how to figure it out for yourself appear in [Section 3.5](#).

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.

3.2 Writing source code

In addition 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 15](#), which shows a variant of a “[hello, world](#)” program written in the Ruby programming language. (You are not expected to understand this program.)

Listing 15: A variant of “hello, world” in Ruby.

```
1 # Prints a greeting.
2 def hello(location)
3   puts "hello, #{location}!"
4 end
5
6 hello("world")
```

To see the contents from [Listing 15](#) in a text editor, we can fire up Atom as follows:

```
$ atom hello.rb
```

Upon pasting in the content of [Listing 15](#), we get the result shown in [Figure 54](#). (For extra credit, type in [Listing 15](#) by hand using the **def** tab trigger discussed in [Section 3.1](#).)

The screenshot shows a window titled "hello.rb - /Users/mhartl - Atom". The file content is:

```
1 # Prints a greeting.
2 def hello(location)
3   puts "hello, #{location}!"
4 end
5
6 hello("world")
7
```

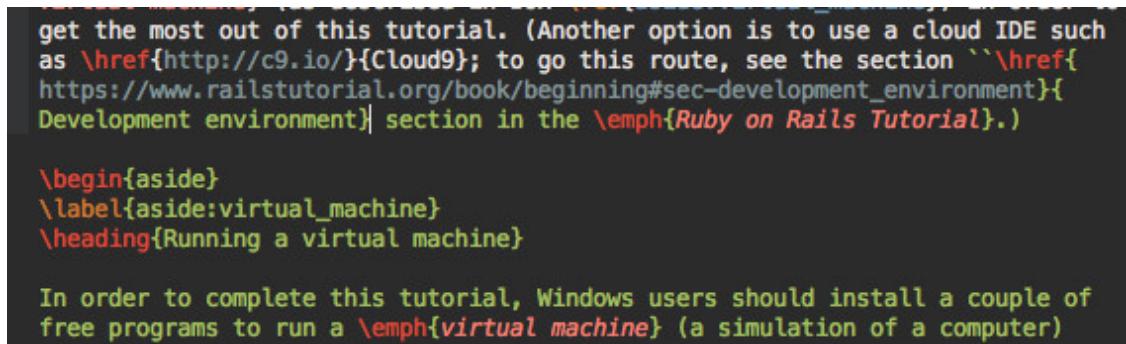
The status bar at the bottom left shows "hello.rb 1:1" and at the bottom right shows "LF UTF-8 Ruby".

Figure 54: A Ruby program in Atom.

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 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 (i.e., 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 L^AT_EX closing quote (which consists of the two single quotes ' '), with the result shown in [Figure 55](#). 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 56](#).



The screenshot shows a dark-themed LaTeX editor interface. The main text area displays the following LaTeX code:

```
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 55: An error in L^AT_EX source caught by syntax highlighting.

```

get the most out of this tutorial. (Another option is to use a cloud IDE such
as \ref{http://c9.io/}{Cloud9}; to go this route, see the section ``\ref{
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 56: Error fixed, syntax highlighting as expected.

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 ignored by the programming language itself.²² An example of a Ruby comment appears in the first line of [Listing 15](#):

```
# 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")
```

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 57](#)) and hitting `⌘/`, as shown in [Figure 58](#). (Note from [Figure 58](#) that the subtle save indicator shown in [Figure 43](#) has been filled in; this is because I habitually press `⌘S` after making changes, as recommended in [Section 2.7](#).)

A screenshot of the Atom code editor window. The title bar reads "hello.rb - /Users/mhartl - Atom". The main editor area shows the following Ruby code:

```
1 # Prints a greeting.
2 def hello(location)
3   puts "hello, #{location}!"
4 end
5
6 hello("world")
7
```

The status bar at the bottom left shows "hello.rb 5:1 (4,53)" and the status bar at the bottom right shows "LF UTF-8 Ruby".

Figure 57: Preparing to comment out some lines.

The screenshot shows a dark-themed window for the Atom code editor. The title bar reads "hello.rb - /Users/mhartl - Atom". The main pane displays the following Ruby code:

```
1 # Prints a greeting.
2 # def hello(location)
3 #   puts "hello, #{location}!"
4 # end
5
6 hello("world")
7
```

The file path "hello.rb" is also visible at the bottom left, along with the line number "5:1" and column "4,53". At the bottom right, the encoding is listed as "LF UTF-8 Ruby".

Figure 58: Commented-out lines.

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 57](#)).

This is useful when restoring some commented-out text after, for example, doing some debugging.

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.²³ For example, Ruby typically uses two spaces for indentation:

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

In most languages, this is equivalent to the following:

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

This second example is harder to read, though, and it's generally important to indent properly for the sake of humans reading the code even if the programming language doesn't care.²⁴

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 15](#) and typing in the following two lines:

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

The result appears in [Figure 59](#).

The screenshot shows a window titled "hello.rb - /Users/mhartl - Atom". The file content is:

```
1 # Prints a greeting.
2 def hello(location)
3   puts "Hello, #{location}!"
4   puts "Uh, oh."
5   puts "Goodbye, #{location}!"
6 end
7
8 hello("world")
9
```

The status bar at the bottom left says "hello.rb 5:31" and the bottom right says "LF UTF-8 Ruby".

Figure 59: Adding two indented lines.

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 59](#) six extra spaces, making eight spaces total. As with commenting out, the first step is to select the text we want to indent ([Figure 60](#)). 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 3](#)) to figure out how to change it.) The result of applying three tabs in succession is shown in [Figure 61](#).

The screenshot shows a window titled "hello.rb - /Users/mhartl - Atom". The file content is:

```
1 # Prints a greeting.
2 def hello(location)
3   puts "Hello, #{location}!"
4   puts "Uh, oh."
5   puts "Goodbye, #{location}!"
6 end
7
8 hello("world")
9
```

The status bar at the bottom left shows "hello.rb 6:1 (4, 77)". The status bar at the bottom right shows "LF UTF-8 Ruby".

Figure 60: Preparing to indent some lines.

The screenshot shows a window titled "hello.rb - /Users/mhartl - Atom". The file content is:

```
1 # Prints a greeting.
2 def hello(location)
3     puts "Hello, #{location}!"
4     puts "Uh, oh."
5     puts "Goodbye, #{location}!"
6 end
7
8 hello("world")
9
```

The status bar at the bottom left shows "hello.rb 6:1 (4, 77)". The status bar at the bottom right shows "LF UTF-8 Ruby".

Figure 61: 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 62](#). (By the way, many editors (including Atom) support the alternate keyboard shortcuts ⌘] and ⌘[for indenting and dedenting, respectively.)

```
hello.rb - /Users/mhartl - Atom
hello.rb
1 # Prints a greeting.
2 def hello(location)
3   puts "Hello, #{location}!"
4   puts "Uh, oh."
5   puts "Goodbye, #{location}!"
6 end
7
8 hello("world")
9
```

Figure 62: Dedenting the code block in [Figure 61](#).

Goto line number

It's often important to be able to go to a particular line number, such as when debugging program that an error on (say) line 187. We saw this feature in [Section 1.5](#), 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 63](#). (Incidentally, the syntax `:<number>` shown in [Figure 63](#), which is for Sublime Text, also works in Vim.)

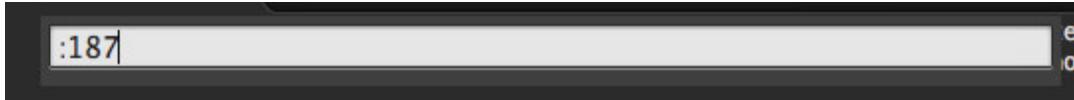


Figure 63: The modal box for going to a particular line number.

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. 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. (The main exception to the 80-column rule is markup like HTML, Markdown, or L^AT_EX, which is why in these cases we often activate word wrap as in [Section 2.2](#).) 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 horizontal line showing where the limit is, as shown in [Figure 64](#).²⁵ If the 80-column limit indicator isn't shown by default in your editor, flex your

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

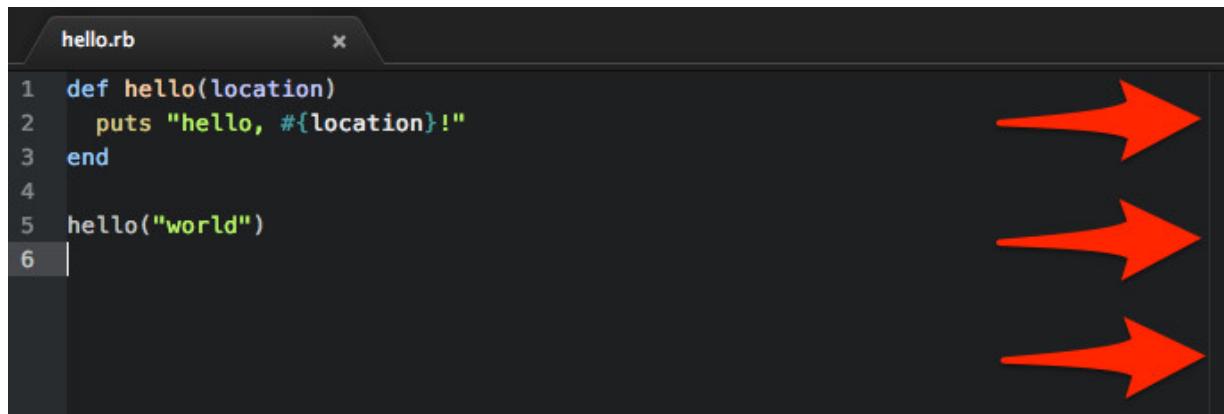


Figure 64: Unsubtle arrows pointing at the subtle 80-column indicator in Atom.

Exercises

1. Create the file **foo.rb**, then define the class **FooBar** ([Listing 16](#)) using a tab trigger. *Hint:* Chances are the trigger is something like `c la->`.
2. Referring to [Listing 17](#), 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 18](#).
4. By cutting and pasting the text for the **hello** definition and indenting the block, transform [Listing 18](#) into [Listing 19](#).

Listing 16: Creating a class using a tab trigger.

`~/foo.rb`

```
class FooBar
end
```

Listing 17: Using autocomplete to make a class name.

```
~/foo.rb
```

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

FooBar.new.bazquux
```

Listing 18: A proto-Greeter class in Ruby.

```
~/greeter.rb
```

```
class Greeter
end

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

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

Listing 19: 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 20](#).

Listing 20: 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 “she-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. By default, this program is the Bourne shell, or Bash, and in this context a shell script is often called a *Bash script*. 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 21](#).

Listing 21: A custom escalating kill script.

```
~/bin/ekill
```

```
1 #!/bin/bash  
2  
3 # Kill a process as safely as possible.  
4 # Tries to kill a process using a series of signals with escalating urgency.  
5 # usage: ekill <pid>  
6  
7 # Assign the process id to the first argument.  
8 pid=$1  
9 kill -15 $pid || kill -2 $pid || kill -1 $pid || kill -9 $pid
```

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 20](#). 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 21](#) into the script file, one thing you might notice is that the result has no syntax highlighting, as seen in [Figure 65](#). This is because, unlike `README.md` in [Section 2.2](#) and `hello.rb` in [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.

A screenshot of the Atom code editor interface. The title bar shows "ekill - /Users/mhartl - Atom". There are two tabs open: "README.md" and "ekill". The "ekill" tab contains the following text:

```
1 #!/bin/sh
2
3 # Kill a process as safely as possible.
4 # Tries to kill a process using a series of signals with escalating urgency.
5 # usage: ekill <pid>
6
7 # Assign the process id to the first argument.
8 pid=$1
9 kill -15 $pid || kill -2 $pid || kill -1 $pid || kill -9 $pid
10
```

A large red arrow points from the bottom right towards the status bar. The status bar displays "bin/ekill 9:62" on the left, "LF UTF-8" in the center, and "Plain Text" on the right, which is highlighted with a red box.

Figure 65: The `ekill` script with no syntax highlighting.

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 65](#)) 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 21](#), 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 66](#)).

The screenshot shows the Atom code editor interface. At the top, there are two tabs: "README.md" and "ekill". The "ekill" tab is active, displaying a shell script. The script content is as follows:

```
1 #!/bin/sh
2
3 # Kill a process as safely as possible.
4 # Tries to kill a process using a series of signals with escalating urgency.
5 # usage: ekill <pid>
6
7 # Assign the process id to the first argument.
8 pid=$1
9 kill -15 $pid || kill -2 $pid || kill -1 $pid || kill -9 $pid
10
```

A large red arrow points downwards from the top of the image towards the status bar at the bottom of the Atom window. The status bar displays the file path "Dropbox/unix_config/bin/ekill" and the line number "1:1". On the right side of the status bar, there are several icons: "LF", "UTF-8", a red-bordered "Shell Script" button, and a small Twitter icon.

Figure 66: 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; 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 1.2](#). Open `~/.bash_profile` as follows:

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

Then add the `export` line shown in [Listing 22](#).

Listing 22: 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. (Again, don't worry about the details; at this point, focus on the mechanics of text-editing.) To use it, we need to use `source` as in [Section 1.3](#):

```
$ 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:

```
/Users/mhartl/bin/ekill
```

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 23](#), 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 23: 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 23](#), 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](#).)

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 67](#)).
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 21](#) can be used in a string like this: "`Hello, $1`"

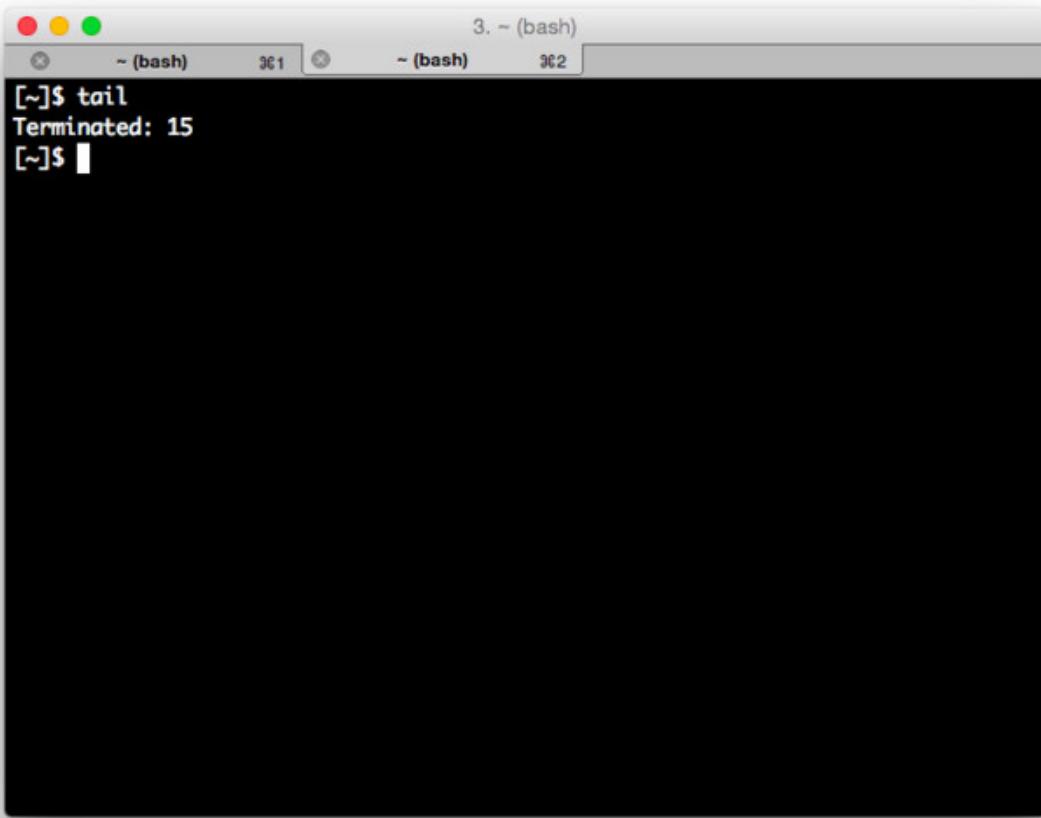


Figure 67: The result of using `ekill` to kill a `tail` process.

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 1.5](#) and [Section 2.2](#), we'll use the `curl` command to download the file to our local disk:

```
$ cd  
$ curl -OL 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/
```

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 68](#). We can toggle its display using the View menu or a keyboard shortcut ([Figure 69](#)).

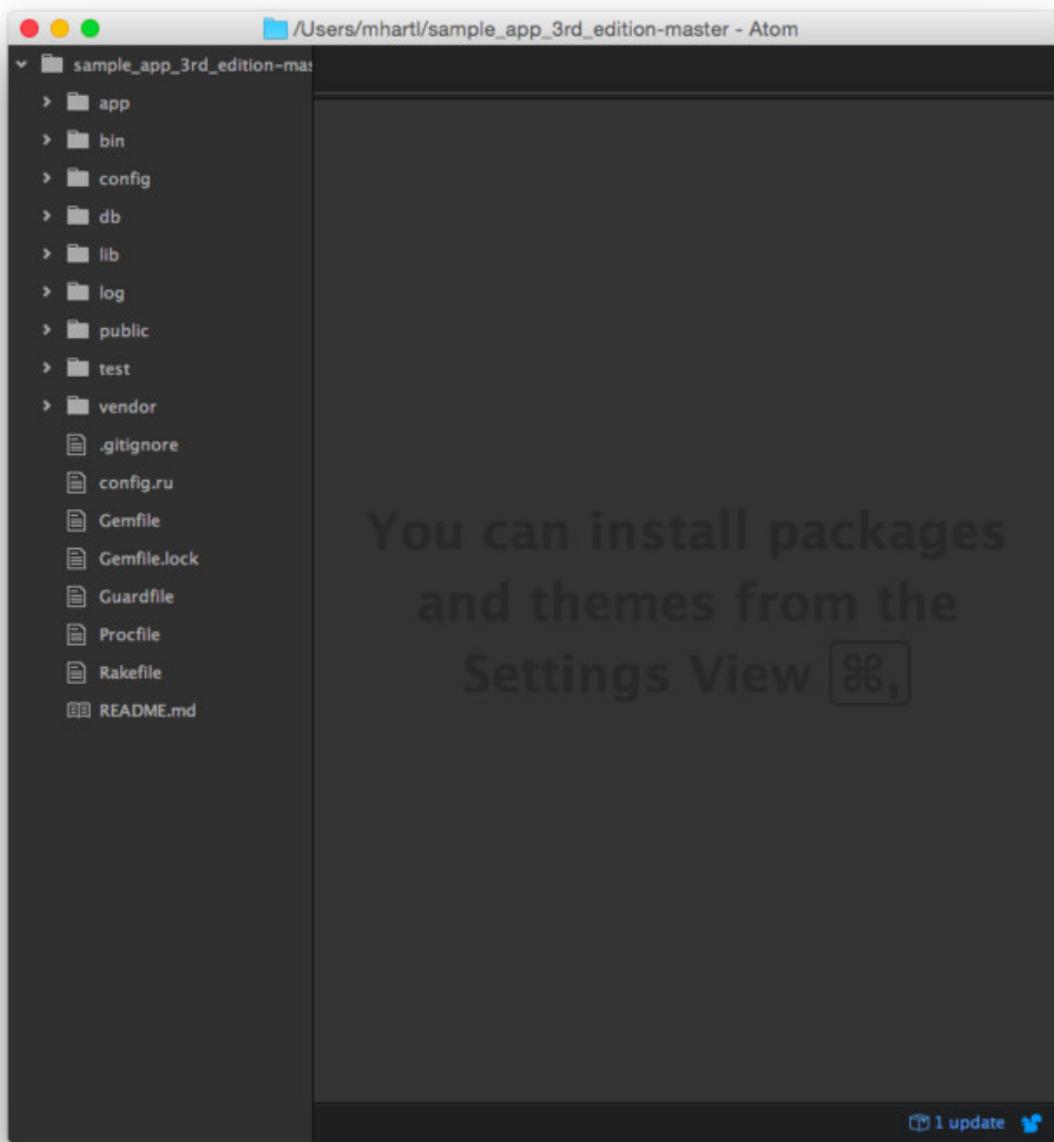


Figure 68: The Rails Tutorial sample app in Atom.

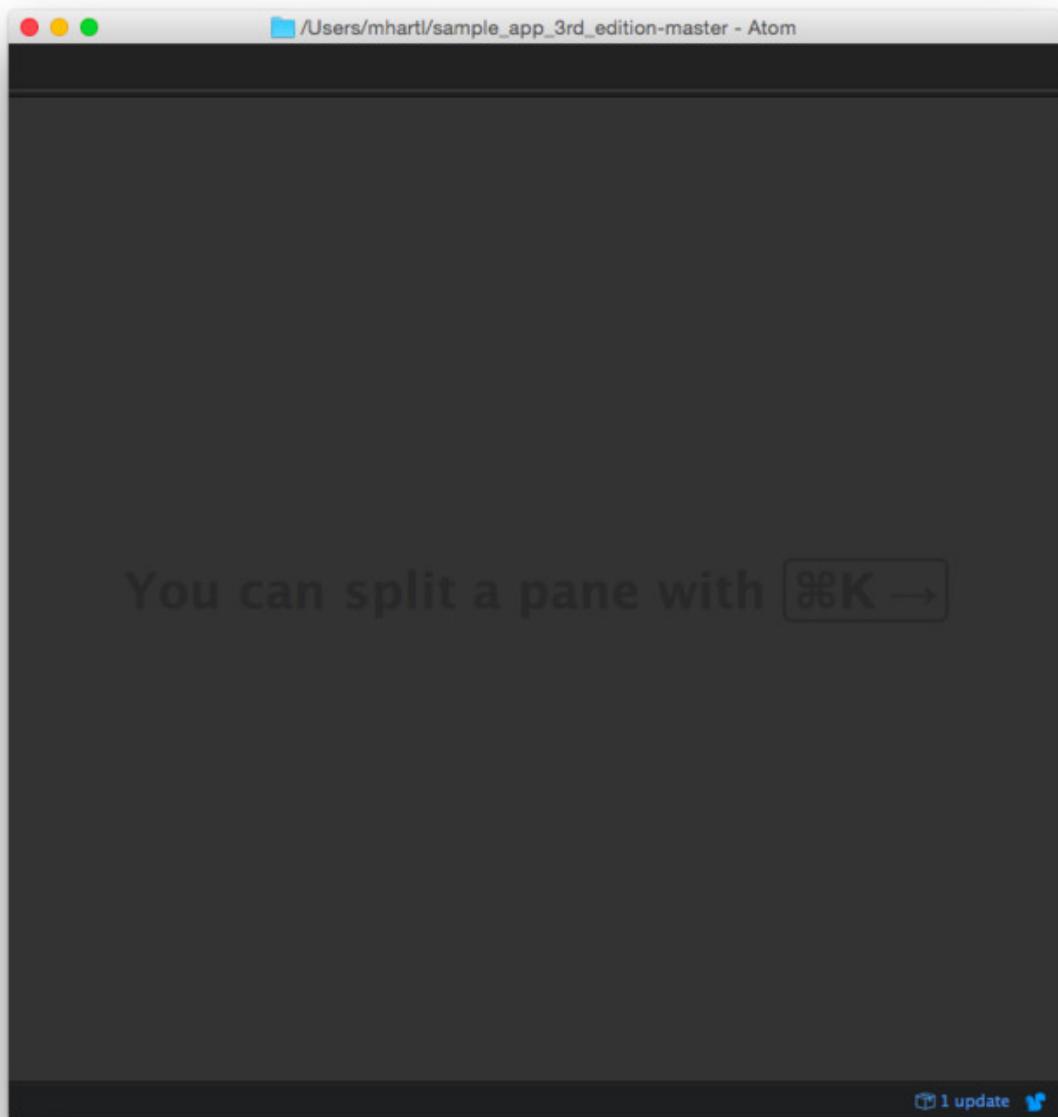


Figure 69: Toggling the tree view.

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 70](#). The letters don't have to be sequential in the filename, though, so typing “uctt” (for `users controller test`) will also work, as seen in [Figure 71](#).

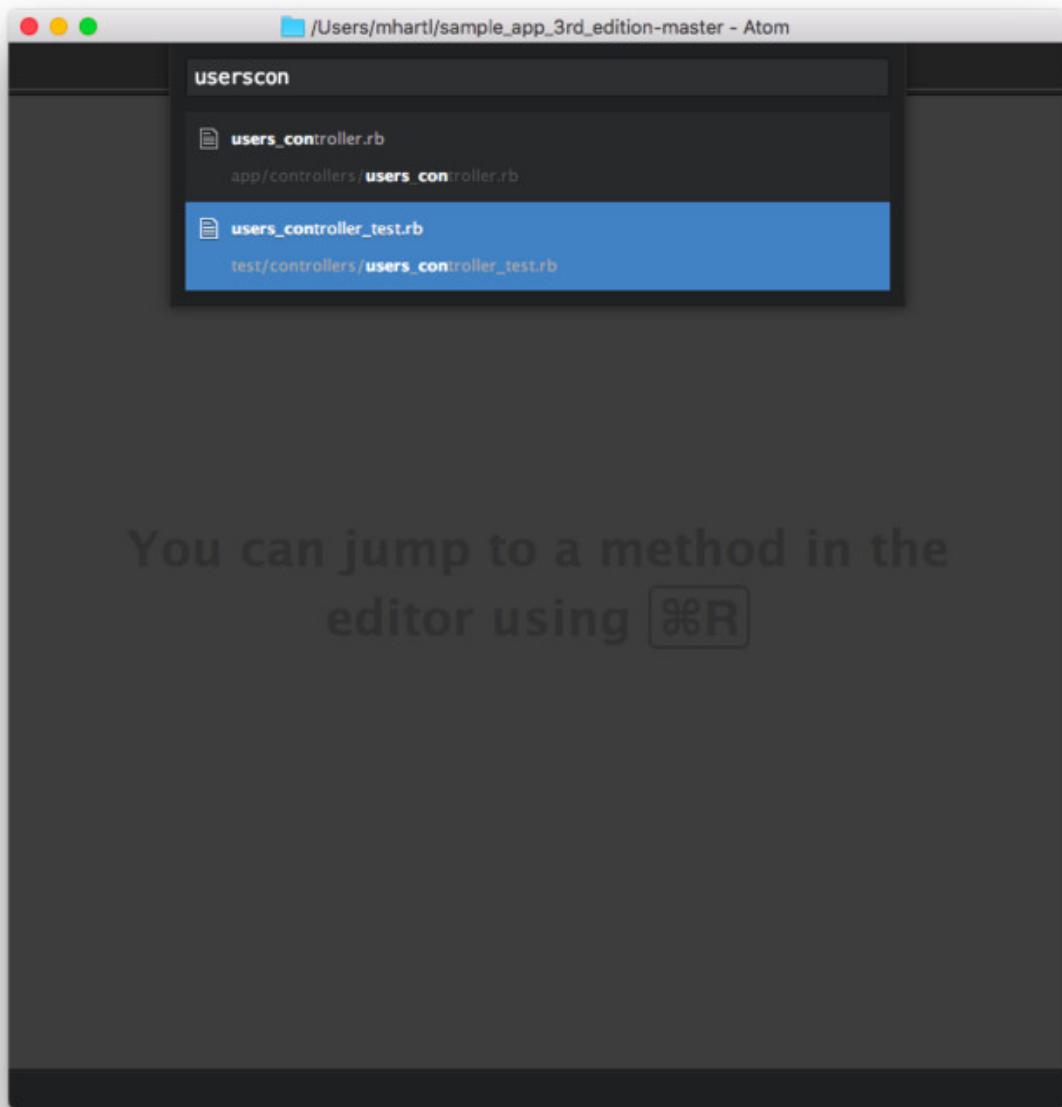


Figure 70: One way to open a file with fuzzy opening.

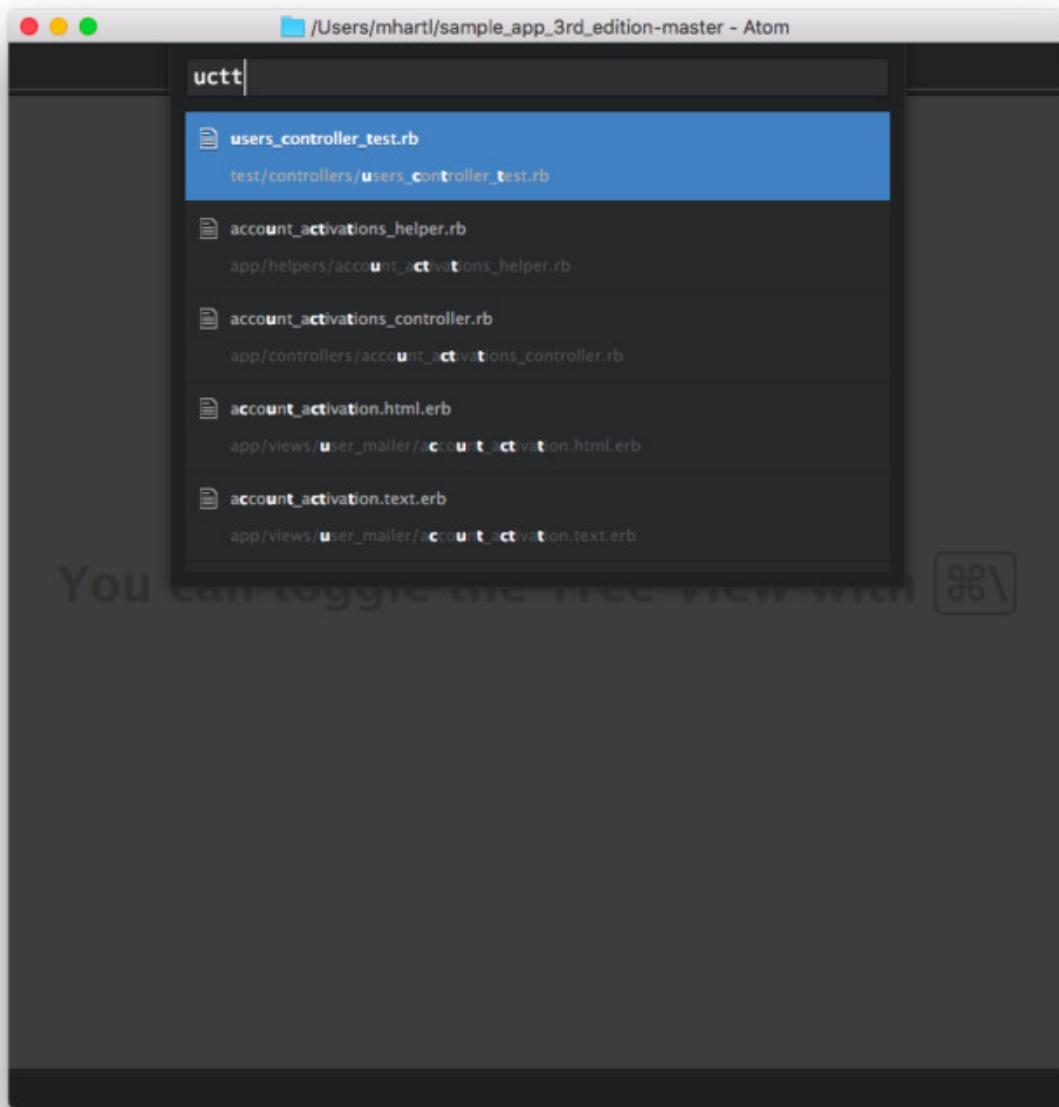
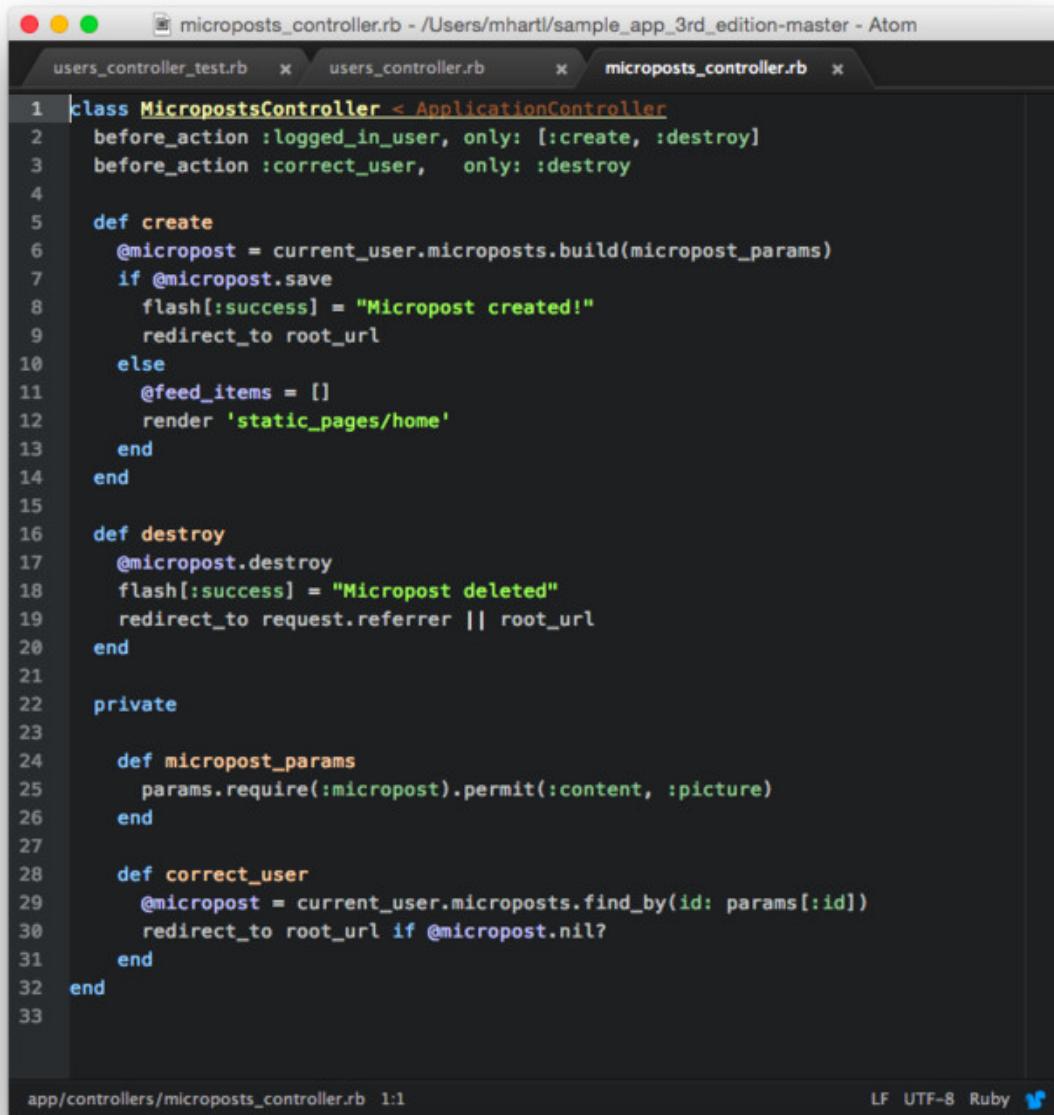


Figure 71: A second way to open a file with fuzzy opening.

As a result of opening multiple files in a project, you will generally have multiple tabs open in your editor ([Figure 72](#)). 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.)



A screenshot of the Atom code editor interface. The title bar shows the path: microposts_controller.rb - /Users/mhart/sample_app_3rd_edition-master - Atom. Below the title bar, there are three tabs: users_controller_test.rb, users_controller.rb, and microposts_controller.rb. The microposts_controller.rb tab is active and displays the following Ruby code:

```
1 class MicropostsController < ApplicationController
2   before_action :logged_in_user, only: [:create, :destroy]
3   before_action :correct_user,   only: :destroy
4
5   def create
6     @micropost = current_user.microposts.build(micropost_params)
7     if @micropost.save
8       flash[:success] = "Micropost created!"
9       redirect_to root_url
10    else
11      @feed_items = []
12      render 'static_pages/home'
13    end
14  end
15
16  def destroy
17    @micropost.destroy
18    flash[:success] = "Micropost deleted"
19    redirect_to request.referrer || root_url
20  end
21
22  private
23
24  def micropost_params
25    params.require(:micropost).permit(:content, :picture)
26  end
27
28  def correct_user
29    @micropost = current_user.microposts.find_by(id: params[:id])
30    redirect_to root_url if @micropost.nil?
31  end
32 end
33
```

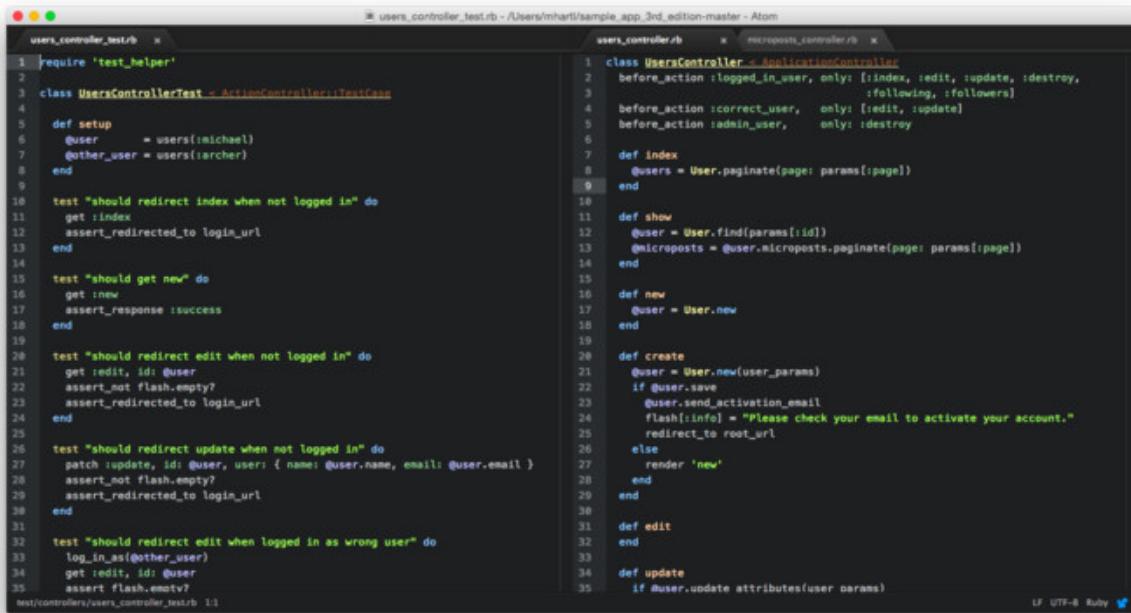
The status bar at the bottom of the editor shows the file path app/controllers/microposts_controller.rb 1:1, the encoding LF, the character set UTF-8, the language Ruby, and the Atom logo.

Figure 72: Opening multiple tabs.

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 73](#)). I especially like to use different panes for different types of files, such as using the left pane for test code and the right pane for application code. It's also often useful to open the same file in two different panes ([Figure 74](#)); 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.



```
users_controller_test.rb
1 require 'test_helper'
2
3 class UsersControllerTest < ActionController::TestCase
4
5   def setup
6     @user      = users(:michael)
7     @other_user = users(:archer)
8   end
9
10 test "should redirect index when not logged in" do
11   get :index
12   assert_redirected_to login_url
13 end
14
15 test "should get new" do
16   get :new
17   assert_response :success
18 end
19
20 test "should redirect edit when not logged in" do
21   get :edit, id: @user
22   assert_not flash.empty?
23   assert_redirected_to login_url
24 end
25
26 test "should redirect update when not logged in" do
27   patch :update, id: @user, user: { name: @user.name, email: @user.email }
28   assert_not flash.empty?
29   assert_redirected_to login_url
30 end
31
32 test "should redirect edit when logged in as wrong user" do
33   log_in_as(@other_user)
34   get :edit, id: @user
35   assert flash.empty?
36
users_controller.rb
1 class UsersController < ApplicationController
2   before_action :logged_in_user, only: [:index, :edit, :update, :destroy]
3   before_action :correct_user,   only: [:edit, :update]
4   before_action :admin_user,    only: [:destroy]
5
6   def index
7     @users = User.paginate(page: params[:page])
8   end
9
10  def show
11    @user = User.find(params[:id])
12    @microposts = @user.microposts.paginate(page: params[:page])
13  end
14
15  def new
16    @user = User.new
17  end
18
19  def create
20    @user = User.new(user_params)
21    if @user.save
22      @user.send_activation_email
23      flash[:info] = "Please check your email to activate your account."
24      redirect_to root_url
25    else
26      render 'new'
27    end
28  end
29
30  def edit
31  end
32
33  def update
34    if @user.update_attributes(user_params)
35
test/controllers/users_controller_test.rb 1:1
```

Figure 73: Using multiple panes.

```
* text_editor.tex — learn_enough_text_editor

text_editor.tex
1637 \begin{center}
1638 \caption{Opening multiple tabs.\label{fig:multiple_tabs}}
1639 \end{figure}
1640
1641 % subsection fuzzy_opening (end)
1642
1643 \subsubsection{Multiple panes} % (fold)
1644 \label{sec:multipane_panes}
1645
1646 The default editor view we've seen in most of the previous examples consists of a single \mpn[pane] (like a 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. This is useful for comparing different versions of a file, such as using the left pane for test code and the right pane for application code (\Figure{\ref{fig:multipane_panes}}). It's also often useful to open the same file in two different panes (\Figure{\ref{fig:same_file_different_panes}}); as I wrote in http://www.learnenough.com/tutorial-writing-tutorial:
1647 Learn Enough™ Tutorial Writing to Be Dangerous));
1648
1649 \begin{quote}
1650 When searching through the document for whatever reason (to fix an error, look up a label for a cross-reference, expand a triple star, 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\ldots. 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.
1651 \end{quote}
1652
1653 \begin{figure}
1654 \begin{center}
1655 \Image{images/figures/multiple_panes.png}
1656 \end{center}
1657 \caption{Using multiple panes.\label{fig:multipane_panes}}
1658 \end{figure}
1659
1660 \begin{figure}
1661 \begin{center}
1662 \Image{images/figures/same_file_different_panes.png}
1663 \end{center}
1664 \caption{Opening the same file in two different panes.\label{fig:same_file_diff_panes}}
1665 \end{figure}
1666
1667 \subsubsection{global_find_and_replace} % (fold)
1668 \label{sec:global_find_and_replace}
1669
1670 global find and replace lets you find or replace across multiple files. Very dangerous, because it's difficult to undo if you make a mistake. Best applied in concert with version control. Will learn about in \vref.
1671
1672 \Figure{\ref{fig:find_in_project}}, \Figure{\ref{fig:replace_in_project}}
1673
1674 \begin{figure}
1675 \begin{center}
1676 \Image{images/figures/find_in_project.png}
1677 \end{center}
1678 \caption{The result of find in project.\label{fig:find_in_project}}
1679 \end{figure}
1680
1681 \begin{figure}
1682 \begin{center}
1683 \Image{images/figures/replace_in_project.png}
1684 \end{center}
1685 \caption{The result of replace in project.\label{fig:replace_in_project}}
1686 \end{figure}
1687
1688 For really advanced replacing, can use regexes. Often indicated with a menu item that looks like \Mode{+}. Play with \vref{http://rubular.com/}{Rubular} to learn more about regexes.
1689
1690 % subsection global_find_and_replace (end)
1691
1692 \subsubsection{Exercises} % (fold)
1693 \label{sec:exercises_project}
1694
1695 \begin{enumerate}
1696 \item What is the keyboard shortcut in your editor for toggling the tree view?
1697 \item What is the keyboard shortcut in your editor for splitting panes horizontally?
1698 \end{enumerate}
1699
1700 % subsection exercises_project (end)
1701
1702 % subsection editing_projects (end)
1703
1704
1705
1706
1707
1708
1709
1710
1711
1712
1713
1714
```

Figure 74: Opening the same file in two different panes.

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 75](#)) and a keyboard shortcut (often ⌘F).

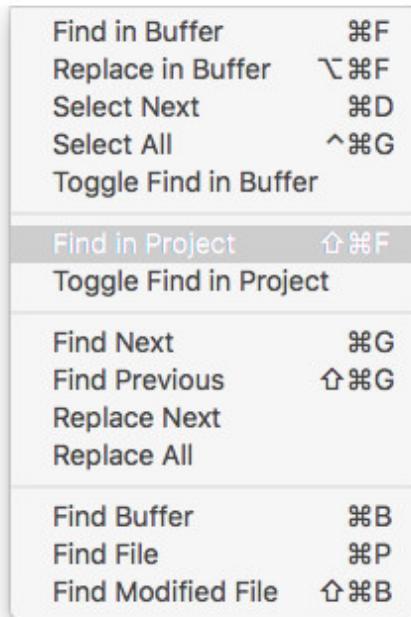


Figure 75: A menu item for global find and replace.

An example of global find appears in [Figure 76](#), which searches for the string “@user” in all project files. The command to globally replace this with “@person” appears in [Figure 77](#).

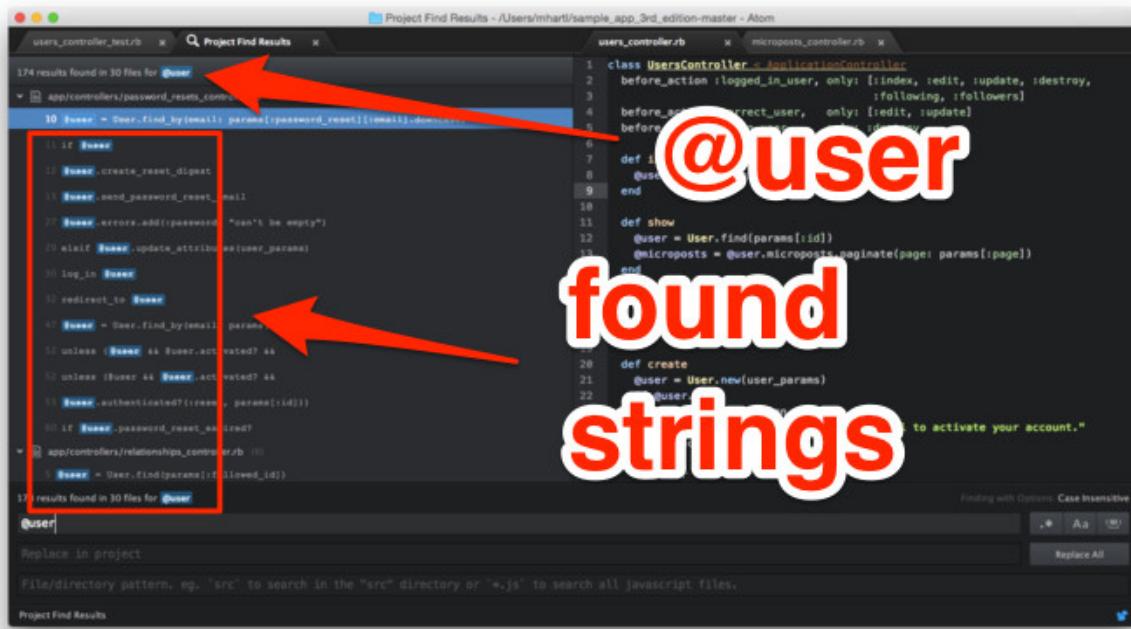


Figure 76: The result of finding in project.

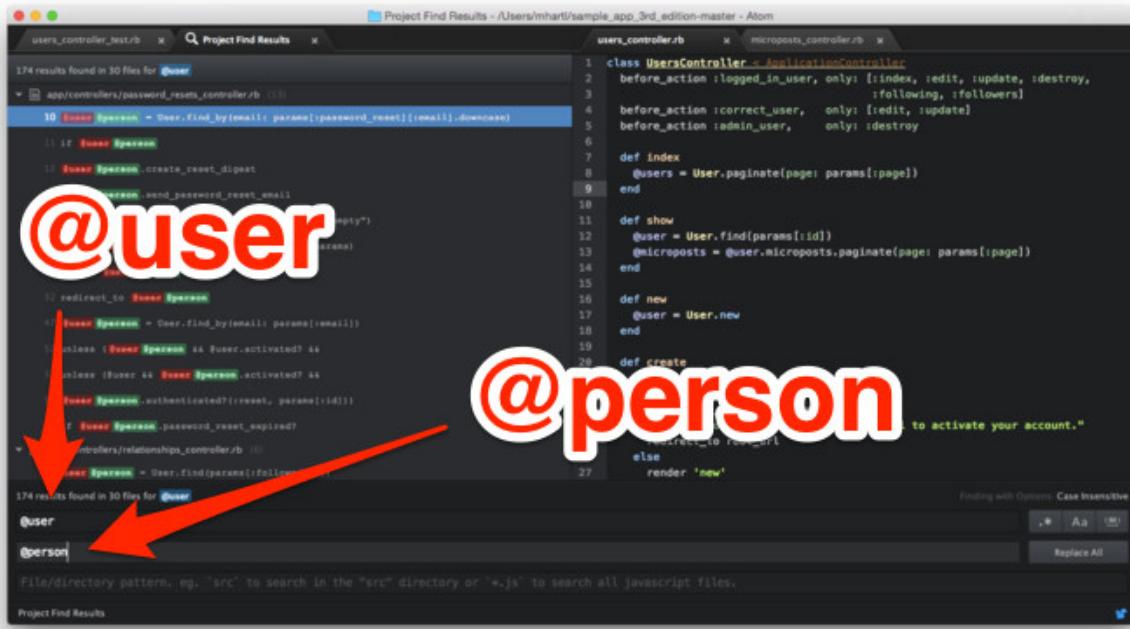


Figure 77: The result of replacing in project.

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 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.)

By the way, for really advanced replacing we can use *regular expressions* (or *regexes*), which are often indicated with a menu item that looks like `.*`. Regexes are covered in more detail in [Learn Enough™ Regular Expressions to Be Dangerous](#). If you decide to experiment with regular expression

searches, I also recommend playing around with the outstanding [Rubular](#) regular expression builder.

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?
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`.

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 3](#)) to figure out how to make the desired changes.

For example, one student of the [Ruby on Rails Tutorial](#) wrote in asking about the dark background in the Cloud9 editor (e.g., [Figure 17](#)), 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 [Figure 3](#)), I was able to discover the

answer (Preferences > Themes > Syntax Theme > Cloud9 Day), as shown in [Figure 78](#).

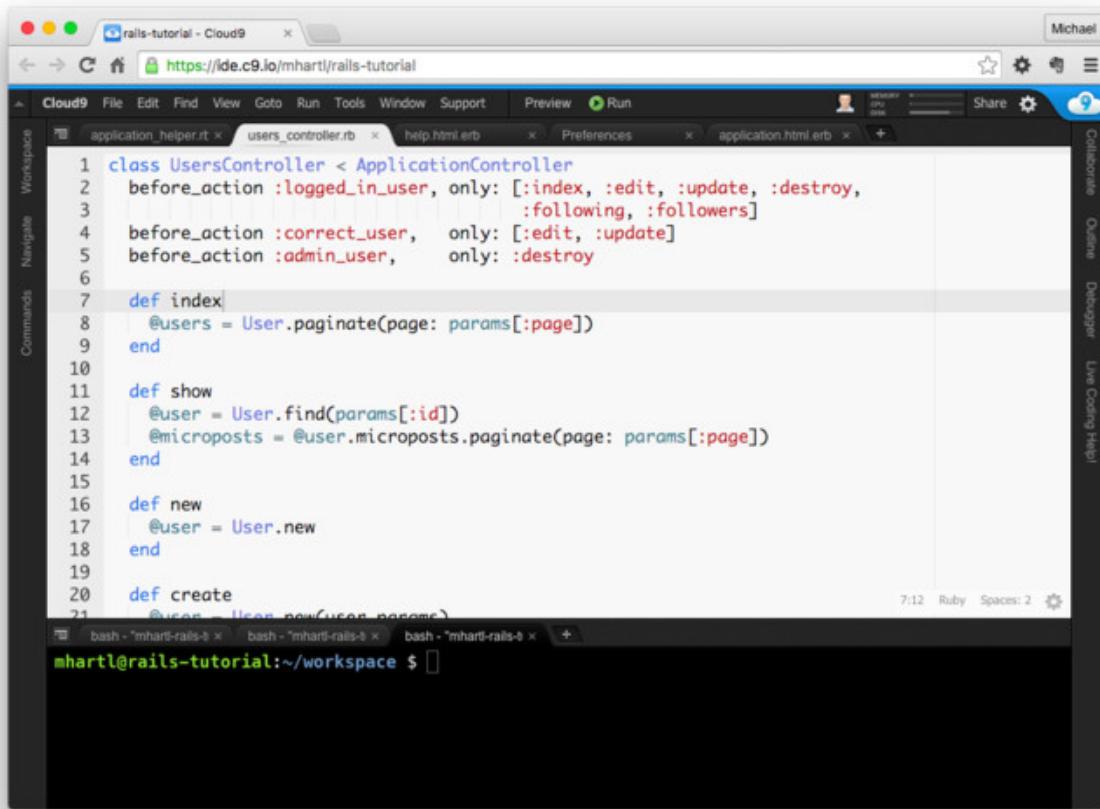


Figure 78: The Cloud9 editor with a light background.

Another feature common to good text editors is some sort of package system. 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 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 79](#). By clicking around the Sublime menu items, we can also find Sublime Text > Preferences > Package Control, as shown in [Figure 80](#) and [Figure 81](#).

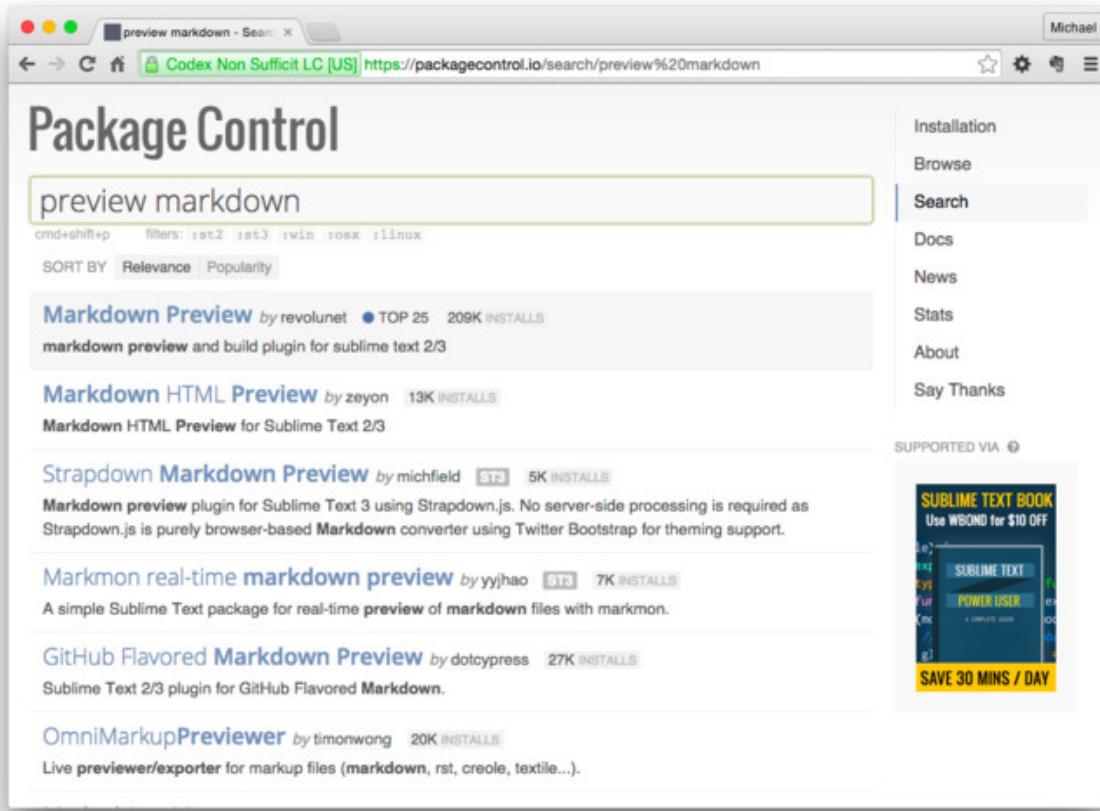


Figure 79: Searching for a Sublime Text package.

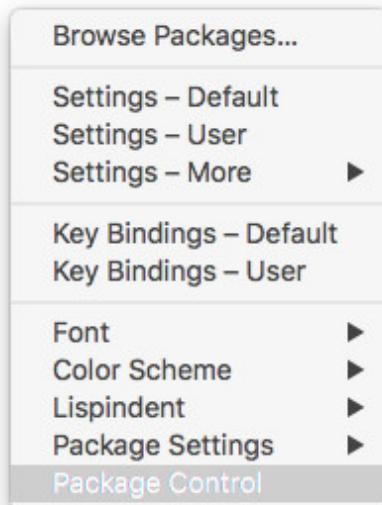


Figure 80: Sublime Text's Package Control menu item.

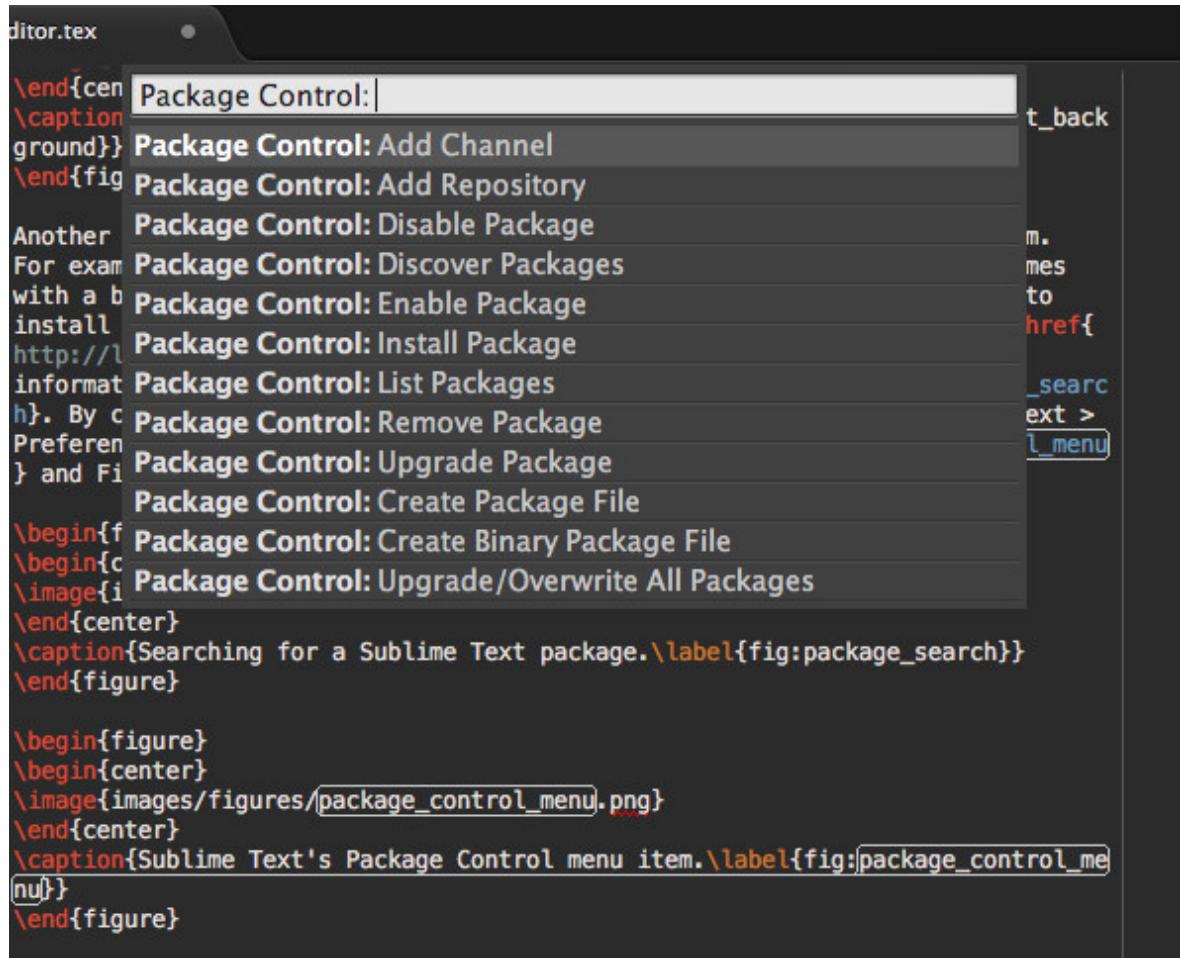


Figure 81: Sublime Text’s Package Control.

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 24](#)), [Google around](#) to figure out how to add custom commands to your editor. (The code in [Listing 24](#) is generated using the custom Sublime Text tab trigger `c list` (for “code listing”), which I have also [ported](#) to Atom.)

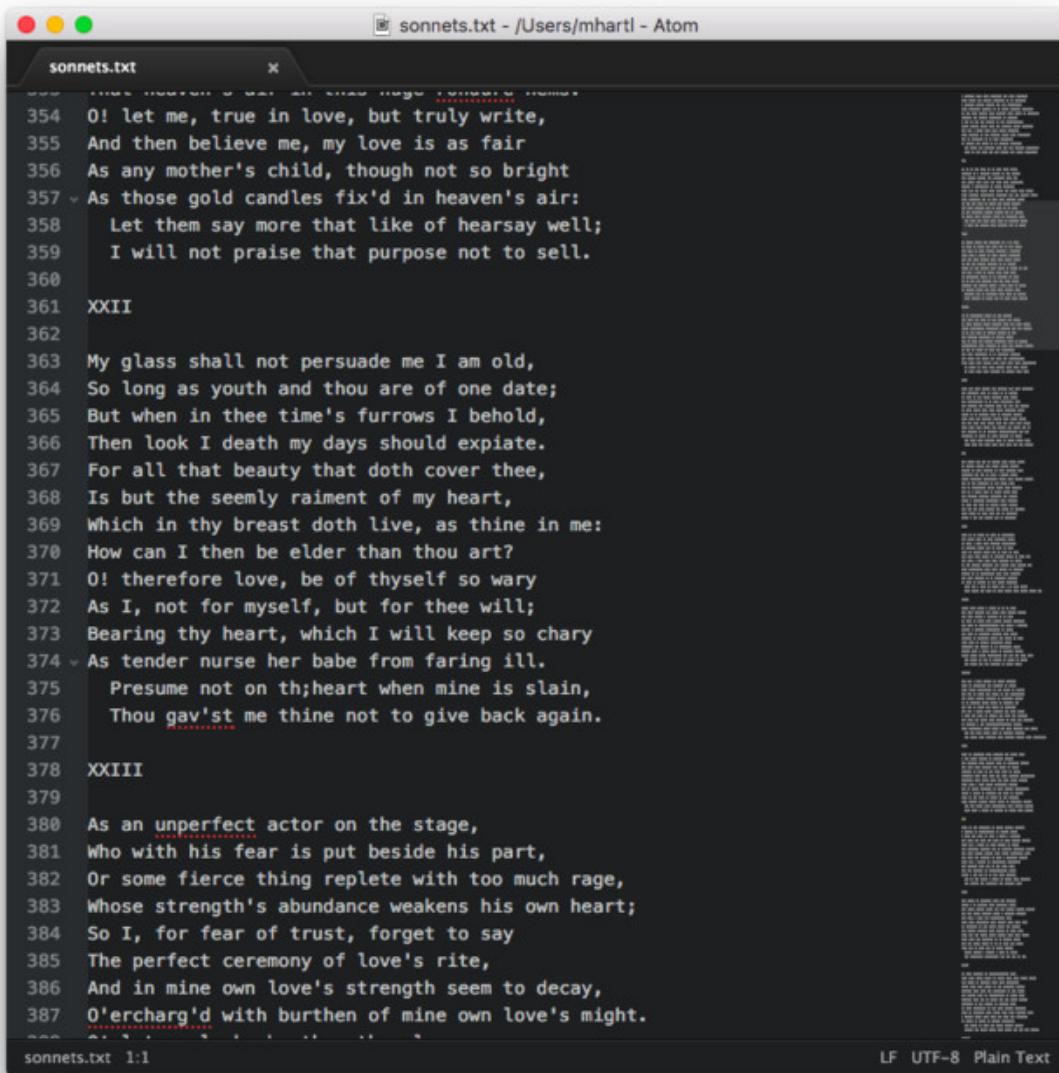
Listing 24: The boilerplate for a code listing in this document.

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

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

Exercises

1. Figure out how to change the syntax highlighting theme in your editor.
Use the file from [Listing 19](#) 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 82](#).

A screenshot of the Atom code editor window titled "sonnets.txt - /Users/mhartl - Atom". The file contains the first three sonnets of William Shakespeare's 154 Sonnets. The code editor features a dark theme with syntax highlighting. A vertical minimap is visible on the right side of the editor area, showing a thumbnail representation of the entire document's structure. The status bar at the bottom shows "sonnets.txt 1:1" on the left and "LF UTF-8 Plain Text" on the right.

```
354 O! let me, true in love, but truly write,
355 And then believe me, my love is as fair
356 As any mother's child, though not so bright
357 As those gold candles fix'd in heaven's air:
358 Let them say more that like of hearsay well;
359 I will not praise that purpose not to sell.
360
361 XXII
362
363 My glass shall not persuade me I am old,
364 So long as youth and thou are of one date;
365 But when in thee time's furrows I behold,
366 Then look I death my days should expiate.
367 For all that beauty that doth cover thee,
368 Is but the seemly raiment of my heart,
369 Which in thy breast doth live, as thine in me:
370 How can I then be elder than thou art?
371 O! therefore love, be of thyself so wary
372 As I, not for myself, but for thee will;
373 Bearing thy heart, which I will keep so chary
374 As tender nurse her babe from faring ill.
375 Presume not on th;heart when mine is slain,
376 Thou gav'st me thine not to give back again.
377
378 XXIII
379
380 As an unperfect actor on the stage,
381 Who with his fear is put beside his part,
382 Or some fierce thing replete with too much rage,
383 Whose strength's abundance weakens his own heart;
384 So I, for fear of trust, forget to say
385 The perfect ceremony of love's rite,
386 And in mine own love's strength seem to decay,
387 O'ercharg'd with burthen of mine own love's might.
```

Figure 82: The result of adding minimap to Atom.

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 5](#))...)
- 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 4](#).

| Command | Description |
|----------------|-----------------------|
| Select + ⌘/ | Toggle commenting out |
| Select + → | Indent |
| Select + ↑ → | Dedent |
| ^ G | Goto line number |
| ⌘W | Close a tab |

| | |
|-------------------------|---------------------------------|
| \$ echo \$PATH | Show the current path |
| \$ 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 4: Important commands from [Section 3.6](#).

4 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 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:

- [Learn Enough™ Vim to Be Dangerous](#)
- [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. If you're not sure where to go from here, you should take a look at [Learn Enough™ Git to Be Dangerous](#), which is currently in preparation. The full sequence appears as follows:

1. Developer Fundamentals

1. [Learn Enough™ Command Line to Be Dangerous](#)
2. [Learn Enough™ Text Editor to Be Dangerous](#) (you are here)
3. [Learn Enough™ Git to Be Dangerous](#)

2. Web Basics

1. [Learn Enough™ HTML to Be Dangerous](#)
 2. [Learn Enough™ CSS & Layout to Be Dangerous](#)
 3. [Learn Enough™ JavaScript to Be Dangerous](#)
3. **Intro Ruby Web Development**
1. [Learn Enough™ Ruby to Be Dangerous](#)
 2. [Learn Enough™ Sinatra to Be Dangerous](#)
 3. [Learn Enough™ Ruby on Rails to Be Dangerous](#)
4. **Professional Ruby Web Development**
- [The Ruby on Rails Tutorial](#)

Good luck!

Learn Enough™ Text Editor to Be Dangerous. Copyright © 2015 by Michael Hartl.

-
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. ↑
 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 —, but the latter is equivalent and renders better in isolation. For similar reasons, [Listing code:html](#) 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*. ↑
 5. Some developers use an *integrated development environment*, or IDE, for their day-to-day programming, but every IDE includes an integrated text editor, so the lessons in this tutorial still apply. ↑

6. Copyright © Randall Munroe and used unaltered under the terms of the [Creative Commons Attribution-NonCommercial 2.5 Generic](#) license. ↑
7. See [Learn Enough™ Git to Be Dangerous](#) for more details. ↑
8. Vim thus dovetails nicely with a command-line tutorial like [Learn Enough™ Command Line to Be Dangerous](#). ↑
9. 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. ↑
10. 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 3](#) caption refers to the Bash profile file in the home directory. ↑
11. 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*. ↑
12. These are not native keybindings (on OS X they would be Option-left arrow and Option-right arrow), which as noted in the introduction to [Section 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. ↑
13. Hitting `ESC` while in normal mode does no harm, so it’s a good idea to include this step in any case. ↑
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](#).) ↑
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. ↑
16. If you completed the exercises in [Section 2.7.1](#), you can use your own custom `get` alias in place of `curl -OL`. ↑
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 3](#)) to resolve any discrepancies. ↑
18. Other options include TextMate, NotePad++, jEdit, and BBEdit. ↑
19. I used Cloud9 for over a year without discovering the `c9` command. Thanks to alert reader Timothy Kiefer for using his *technical sophistication* ([Box 3](#)) to figure it out! ↑
20. 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. ↑

21. I actually have my Sublime Text editor configured to use the **esc** key for autcompletion 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 3](#)). ↑
22. Technically, comments are ignored by the [compiler](#) or [interpreter](#). Some languages have automated documentation systems that do process the comments. ↑
23. 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. In recent years, virtually all programmers and editors have switched to *emulated tabs*, which use a standard number of ordinary spaces (typically two or four) for each level of indentation. ↑
24. Some languages, notably Python, actually enforce some measure of proper indentation, but most language compilers and interpreters ignore it. ↑
25. In some editors the line is at something like 78 columns instead of 80 to allow a small margin for error. ↑

Table of Contents

Vim

Introduction to modern text editors

Advanced text editing

Conclusion