LEARN ENOUGH

COMMAND LINE

TO BE DANGEROUS

TUTORIAL BY

MICHAEL HARTL

Learn Enough Command Line to Be Dangerous

A tutorial introduction to the Unix command line

Michael Hartl

Contents

About the author

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

Chapter 1

Basics

Learn Enough Command Line to Be Dangerous is an introduction to the command line for complete beginners, the first in a series of tutorials designed to teach the common foundations of "computer magic" (Box 1.1) to as broad an audience as possible. It is aimed both at those who work with software developers and those who aspire to become developers themselves. Unlike most introductions to the command line, which typically assume a relatively high level of technical sophistication, *Learn Enough Command Line to Be Dangerous* assumes no prerequisites other than general computer knowledge (how to launch an application, how to use a web browser, how to touch type, etc.). Among other things, this means that it doesn't assume you know how to use a text editor, or even what a text editor is. Indeed, this tutorial doesn't even assume you know what a *command line* is, so if you're confused by the title, you're still in the right place. Finally, even if you already know how to use the command line, following this tutorial (and doing the exercises) will help fill in any gaps in your knowledge, and might even teach you a few new things.

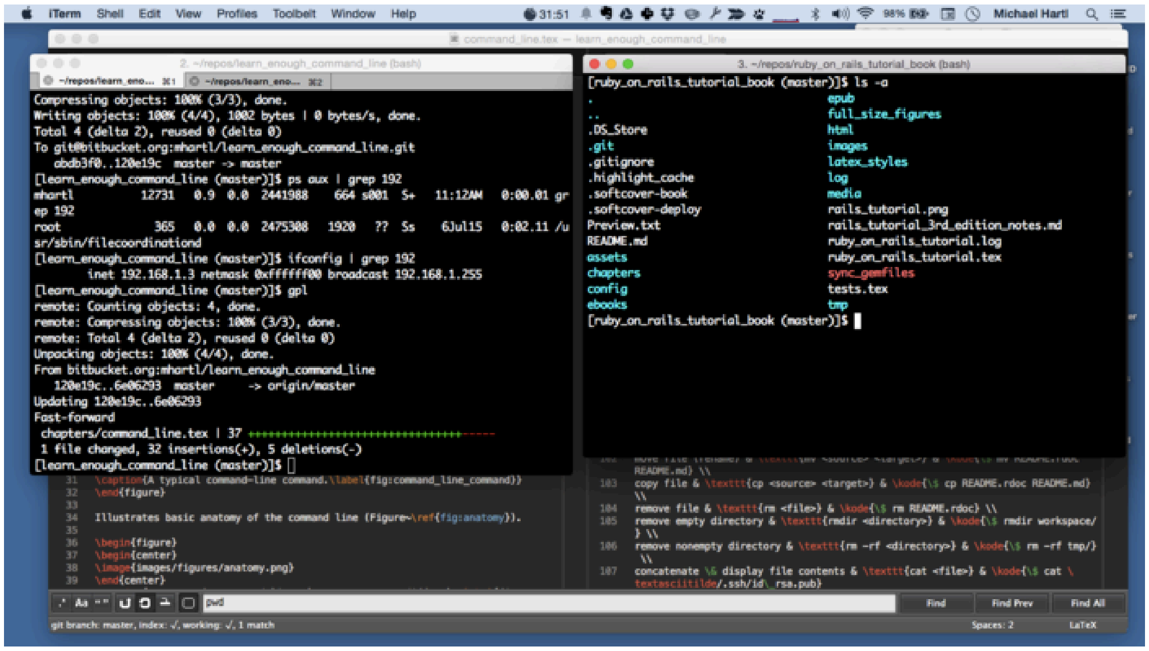
Box 1.1. The magic of computers

Computers may be as close as we get to *magic* in the real world: we type incantations into a machine, and—if the incantations are right—the machine does our bidding. To perform such magic, computer witches and wizards rely not only on words, but also on wands, potions, and an ancient tome or two. Taken together, these tricks of the trade are known as *software development*: computer programming, plus tools like *command lines*, *text editors*, and *version control*. Knowledge of these tools is perhaps the main dividing line between "technical" and "non-technical" people (or, to put it in magical terms, between magicians and Muggles). The present tutorial represents the first step needed to cross this technical/non-technical divide. The resulting *technical sophistication* (Box 1.5) will make us software magicians—able to cast computer spells, and get the machine to do our bidding.

I'm Michael Hartl, and I am perhaps best known as the creator of the Ruby on Rails Tutorial, a book and screencast series that together constitute one of the leading introductions to web development. (You may also know me, in my more mathematical mode, as the founder of Tau Day and author of *The Tau Manifesto*.) One of the most frequently asked questions about the Rails Tutorial is, "Is the Rails Tutorial good for complete beginners?" The answer is, "Not really." While it is possible for complete beginners to learn web development with the Ruby on Rails Tutorial (and an impressively large number have), it can be challenging and occasionally frustrating, and I don't generally recommend it. Instead, I recommend starting here.

Many programming tutorials either gloss over the command line or assume you already know how to use it. But understanding the basics of the command line is *absolutely essential* to becoming a skilled developer.[1] Indeed, if you look at the desktop of an experienced computer programmer, even on a system with a polished graphical user interface like macOS, you are likely to find a large number of "terminal windows", each containing a series of commands at a command line (Figure 1.1). Proficiency at the command line is also useful for anyone who needs to *work* with developers, such as product managers, project managers, and designers. Making this essential component of technical

[1]As discussed in Section 1.1, this statement applies to the *Unix tradition*, which is the principal computing tradition behind the Internet and the World Wide Web, and the tradition followed by this tutorial. In addition to being important for developers, knowing the command line is also essential for system administrators (sysadmins).



```
Compressing objects: 100% (3/3), done.
Writing objects: 100% (4/4), 1002 bytes | 0 bytes/s, done.
Total 4 (delta 2), reused 0 (delta 0)
To git@bitbucket.org:mhartl/learn_enough_command_line.git
   abdb3f0..120e19c  master -> master
[learn_enough_command_line (master)]$ ps aux | grep 192
mhartl          12731   0.9  0.0  2441988    664 s001  S+   11:12AM   0:00.01 gr
ep 192
root              365   0.0  0.0  2475308   1920   ??  Ss    6Jul15   0:02.11 /u
sr/sbin/filecoordinationd
[learn_enough_command_line (master)]$ ifconfig | grep 192
        inet 192.168.1.3 netmask 0xffffff00 broadcast 192.168.1.255
[learn_enough_command_line (master)]$ gpl
remote: Counting objects: 4, done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 2), reused 0 (delta 0)
Unpacking objects: 100% (4/4), done.
From bitbucket.org:mhartl/learn_enough_command_line
   120e19c..6e06293  master     -> origin/master
Updating 120e19c..6e06293
Fast-forward
 chapters/command_line.tex | 37 ++++++++++++++++++++++++++++++++-----
 1 file changed, 32 insertions(+), 5 deletions(-)
[learn_enough_command_line (master)]$ 
```

```
[ruby_on_rails_tutorial_book (master)]$ ls -a
.                                       epub
..                                      full_size_figures
.DS_Store                               html
.git                                    images
.gitignore                              latex_styles
.highlight_cache                        log
.softcover-book                         media
.softcover-deploy                       rails_tutorial.png
Preview.txt                             rails_tutorial_3rd_edition_notes.md
README.md                               ruby_on_rails_tutorial.log
assets                                  ruby_on_rails_tutorial.tex
chapters                                sync_gemfiles
config                                  tests.tex
ebooks                                  tmp
[ruby_on_rails_tutorial_book (master)]$ 
```

```
\caption{A typical command-line command.\label{fig:command_line_command}}
\end{figure}

Illustrates basic anatomy of the command line (Figure~\ref{fig:anatomy}).

\begin{figure}
\begin{center}
\image{images/figures/anatomy.png}
\end{center}
```

```
README.md} \\
copy file & \texttt{cp <source> <target>} & \kode{\$ cp README.rdoc README.md}
\\
remove file & \texttt{rm <file>} & \kode{\$ rm README.rdoc} \\
remove empty directory & \texttt{rmdir <directory>} & \kode{\$ rmdir workspace/
} \\
remove nonempty directory & \texttt{rm -rf <directory>} & \kode{\$ rm -rf tmp/}
 \\
concatenate \& display file contents & \texttt{cat <file>} & \kode{\$ cat \
textasciitilde/.ssh/id\_rsa.pub}
```

iTerm
Shell
Edit
View
Profiles
Toolbelt
Window
Help
Michael Hartl
command_line.tex — learn_enough_command_line
2. ~/repos/learn_enough_command_line (bash)
3. ~/repos/ruby_on_rails_tutorial_book (bash)
Find
Find Prev
Find All
Spaces: 2
LaTeX

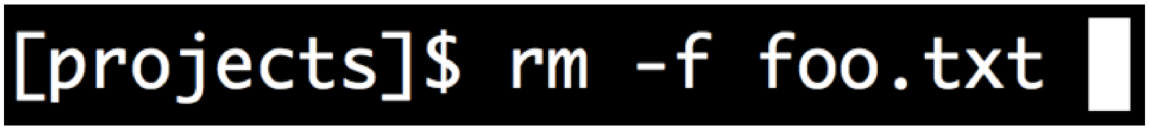
Figure 1.1: Terminal windows on the desktop of an experienced developer.

sophistication accessible to as broad an audience as possible is the goal of *Learn Enough Command Line to Be Dangerous*.

1.1 Introduction

As author Neal Stephenson famously put it, "In the Beginning… Was the Command Line." Although a graphical user interface (GUI) can dramatically simplify computer use, in many contexts the most powerful and flexible way to interact with a computer is through a *command-line interface* (CLI). In such an interface, the user types *commands* that tell the computer to perform desired tasks. These commands can then be combined in various ways to achieve a variety of outcomes. An example of a typical command-line command appears in Figure 1.2.

This tutorial covers the basics of the Unix command line, where *Unix* refers to a family of operating systems that includes Linux, Android, iOS (iPhone



```
[projects]$ rm -f foo.txt
```

Figure 1.2: A prototypical command-line command.

and Pad), and macOS.[2] Unix systems serve most of the software on the World Wide Web, run most mobile and tablet devices, and power many of the world's desktop computers as well. As a result of Unix's central role in modern computing, this tutorial covers the Unix way of developing software. The main exception to Unix's dominance is Microsoft Windows, which is not part of the Unix tradition, but those who mostly develop using native Windows development tools will still benefit from learning the Unix command line. Among other things, at some point such users are likely to need to issue commands on a Unix server (e.g., via the "secure shell" command `ssh`), at which point familiarity with Unix commands becomes essential. As a result, Windows users are encouraged to set up a Linux-compatible development environment by installing a *virtual machine* (Box 1.2) or following the Windows steps in *Learn Enough Dev Environment to Be Dangerous*. Another good option is to use a cloud IDE, which comes with a built-in command line; this option is also covered in *Learn Enough Dev Environment to Be Dangerous*.

Box 1.2. Running a virtual machine

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

Note: These steps can be rather tricky, and unfortunately we don't have the resources to offer individual technical support on virtual machines. If you run into trouble, we suggest giving the cloud IDE a try instead.

[2]In a fairly typical turn of events, the name *Unix* started as a pun on a rival system called *Multics*.

The steps to install a virtual machine appear as follows:

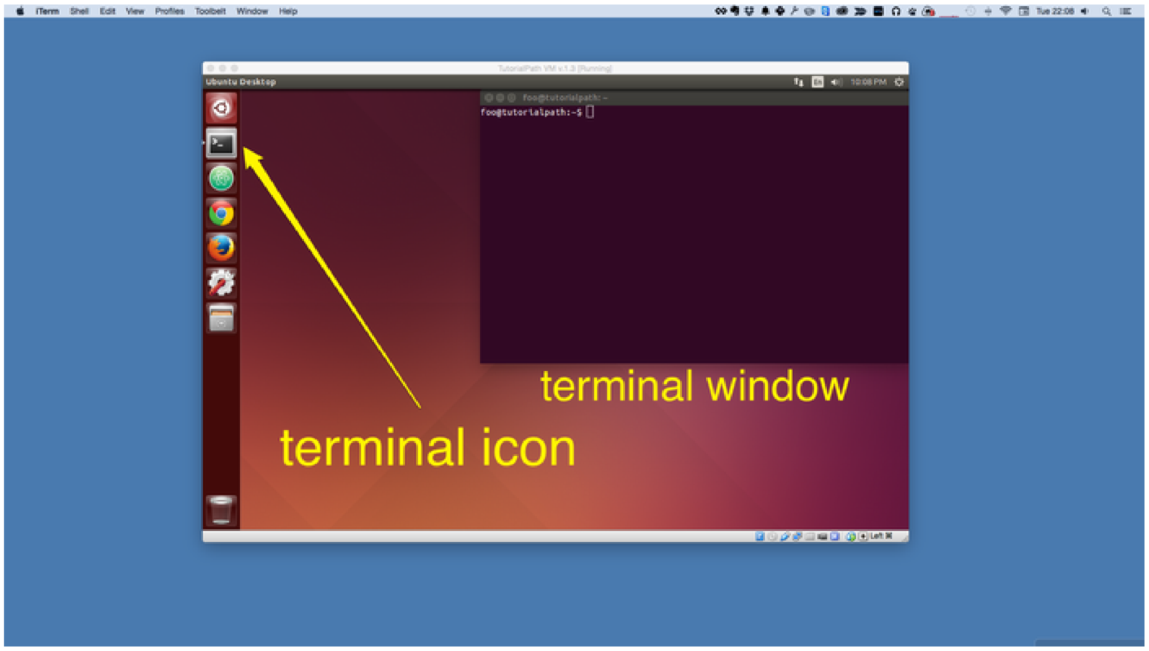
1. Install the right version of VirtualBox for your system (free).
2. Download the Learn Enough Virtual Machine (large file).
3. Once the download is complete, double-click the resulting "OVA" file and follow the instructions to install the Virtual Machine (VM).
4. Double-click the VM itself and log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

(Getting all these steps to work is a good exercise in *technical sophistication*, an idea we'll develop further starting in Section 1.4 (Box 1.5).) The result will be a Linux desktop environment (including a command-line terminal program) preconfigured for this tutorial, as shown in Figure 1.3.

In the longer run, I recommend switching to a Mac as soon as possible. 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.)

1.2 Running a terminal

To run a command-line command, we first need to start a *terminal*, which is the program that gives us a command line. The exact details depend on the particular operating system you're using.



```
foo@tutorialpath:~$
```

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

macOS

On macOS, you can open a terminal window using the macOS application *Spotlight*, which you can launch either by typing ⌘□ (Command-space) or by clicking on the magnifying glass in the upper right part of your screen. Once you've launched Spotlight, you can start a terminal program by typing "terminal" in the Spotlight Search bar. (If you are interested in using a more advanced and customizable terminal program, I recommend installing iTerm, but this step is optional.)

At this point, you might see the alert shown in Listing 1.1.

Listing 1.1: A macOS terminal alert.

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.

[~]$
```

This alert is the result of a change made in macOS Catalina. You don't need to do anything about it right now; we'll address this issue the first time it makes any difference in this tutorial (Section 2.3). For more information, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

Linux

On Linux, you can click the terminal icon as shown in Figure 1.3. The result should be something like Figure 1.4, although the exact details on your system will likely differ.

Windows

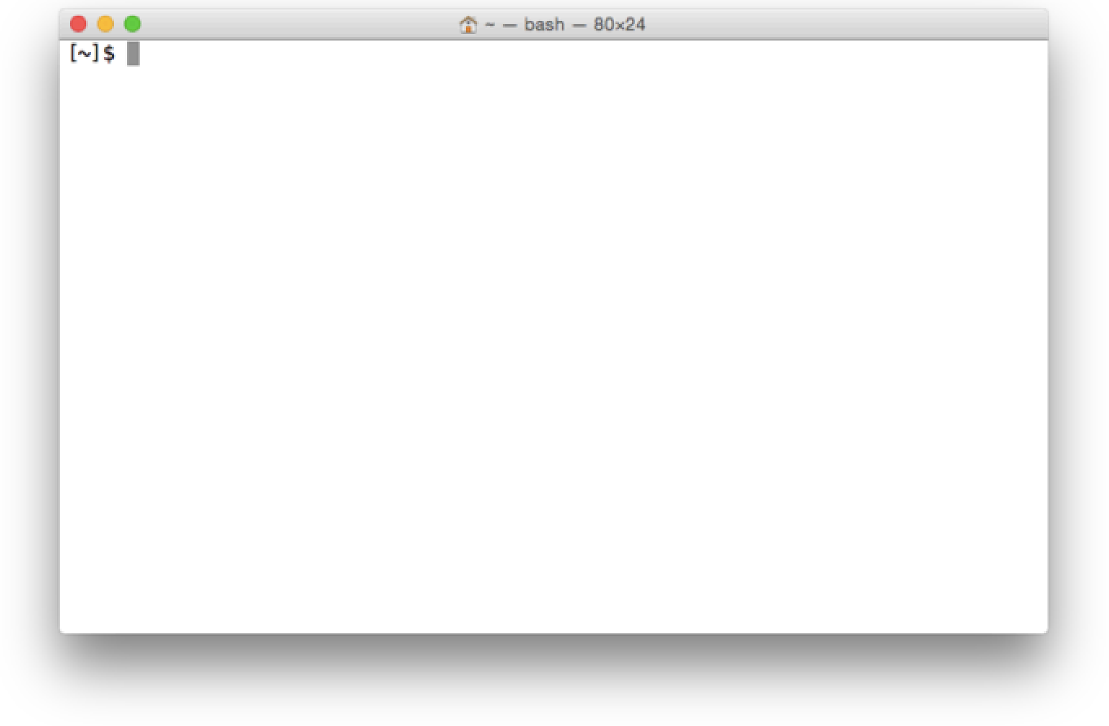
On Windows, the recommended option is to install Linux (which, incredibly, Microsoft has decided to support natively) as described in the Windows section of the free tutorial *Learn Enough Dev Environment to Be Dangerous*. Once Linux is installed, you should look for a terminal icon as described in Section 1.2. Apply your technical sophistication (Box 1.5) if you get stuck.

Terminal window

Regardless of which operating system you use, your terminal window should look something like Figure 1.4, though details may differ.

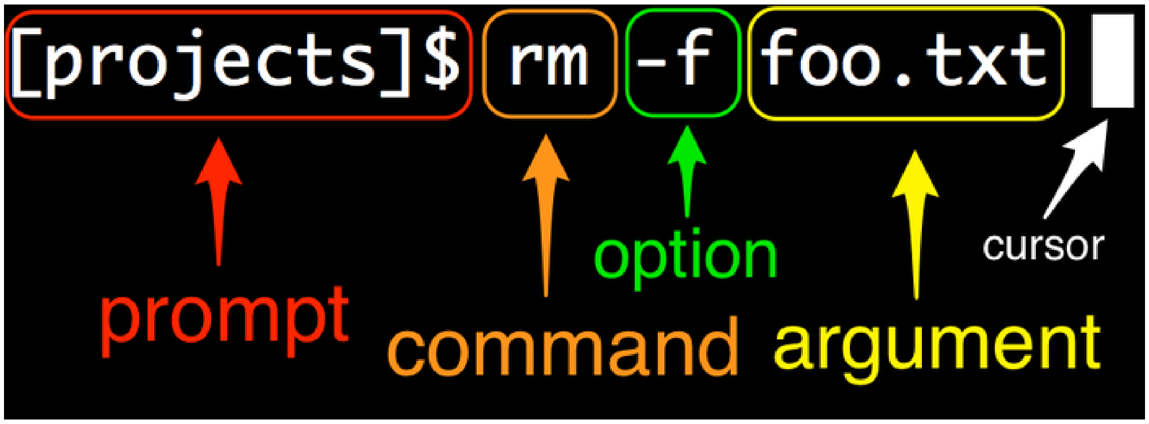
The example we saw in Figure 1.2 includes all of the typical elements of a command, as illustrated in Figure 1.5: the *prompt* (to "prompt" the user to do something) followed by a *command* (as in "give the computer a command"), an *option* (as in "choose a different option"),[3] and an *argument* (as in the "argument of a function" in mathematics). It's essential to understand that the prompt is supplied automatically by the terminal, and you do not need to type it. (Indeed, if you do type it, it will likely result in an error.) Moreover, the exact details of the prompt will differ, and are not important for the purposes of this tutorial (Box 1.3).

[3] An option is sometimes also called a *flag*.



```
[~]$
```

Figure 1.4: A terminal window.



```
[projects]$ rm -f foo.txt
```

Figure 1.5: Anatomy of a command line. (Your prompt may differ.)

Box 1.3. What is the prompt?

Every command line starts with some symbol or symbols designed to "prompt" you to action. The prompt usually ends with a dollar sign `$` or a `%`, and is preceded by information that depends on the details of your system. For example, on some systems the prompt might look like this:

```
Michael's MacBook Air:~ mhartl$
```

In Figure 1.4, the prompt looks like this instead:

```
[~]$
```

and in Figure 1.5 it looks like this:

```
[projects]$
```

Finally, the prompt I'm looking at right now looks like this:

```
[learn_enough_command_line (master)]$
```

For the purposes of this tutorial, the details of the prompt are not important, but we will discuss useful ways to customize the prompt starting in the next tutorial after this one (*Learn Enough Text Editor to Be Dangerous*).

1.2.1 Exercises

Learn Enough Command Line to Be Dangerous includes a large number of exercises. I strongly recommend getting in the habit of attempting them before moving on to the next section, as they reinforce the material we've just covered and will give you essential practice in using the many commands discussed. It's not generally the case that they are *required* to proceed, though, so if you get stuck it's sometimes a good idea to continue forward and then revisit the exer-

cise at a later time. Indeed, this is good advice for the main text as well—you'll be surprised how often a seemingly impossible idea or intractable problem will look easy the second time around.
Solutions to exercises are available for free at learnenough.com/solutions with any Learn Enough purchase. To see other people's answers and to record your own, join the Learn Enough Society at learnenough.com/society.

1. By referring to Figure 1.5, identify the prompt, command, options, arguments, and cursor in each line of Figure 1.6.

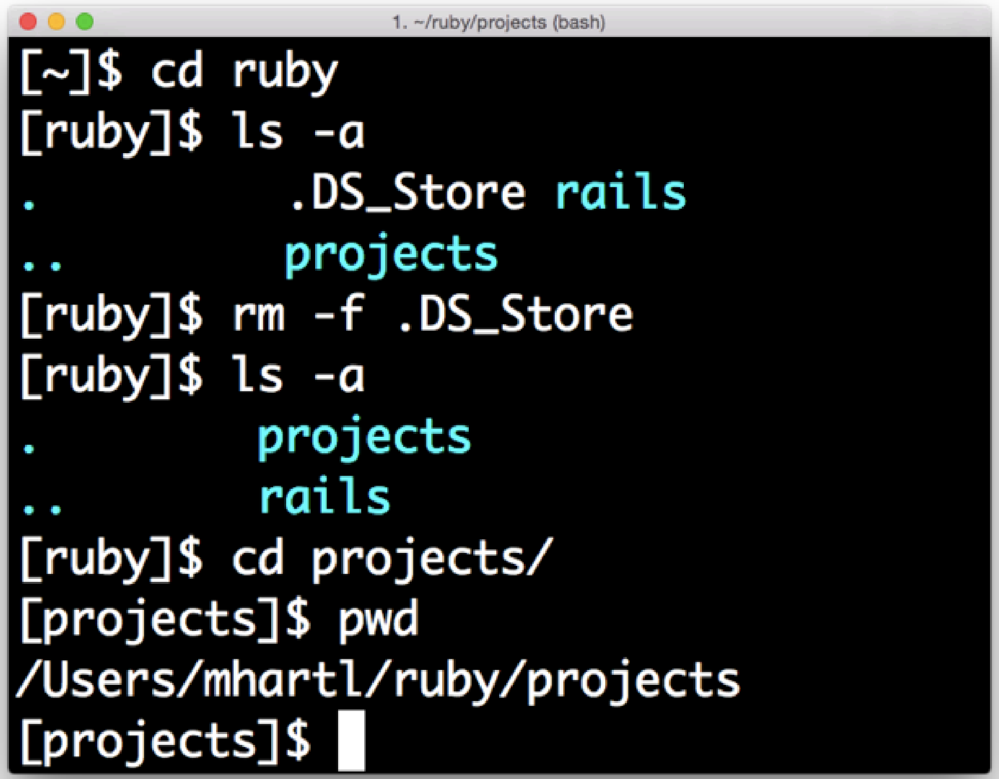
2. Most modern terminal programs have the ability to create multiple *tabs* (Figure 1.7), which are useful for organizing a set of related terminal windows.[4] By examining the menu items for your terminal program (Figure 1.8), figure out how to create a new tab. *Extra credit*: Learn the keyboard shortcut for creating a new tab. (Learning keyboard shortcuts for your system is an excellent habit to cultivate.)

1.3 Our first command

We are now prepared to run our first command, which prints the word "hello" to the screen. (The place where characters get printed is known as "standard out", which is usually just the screen, and rarely refers to a physical printer.) The command is **`echo`**, and the argument is the string of characters—or simply *string* for short—that we want to print. To run the **`echo`** command, type "echo hello" at the prompt, and then press the Return key (also called Enter):

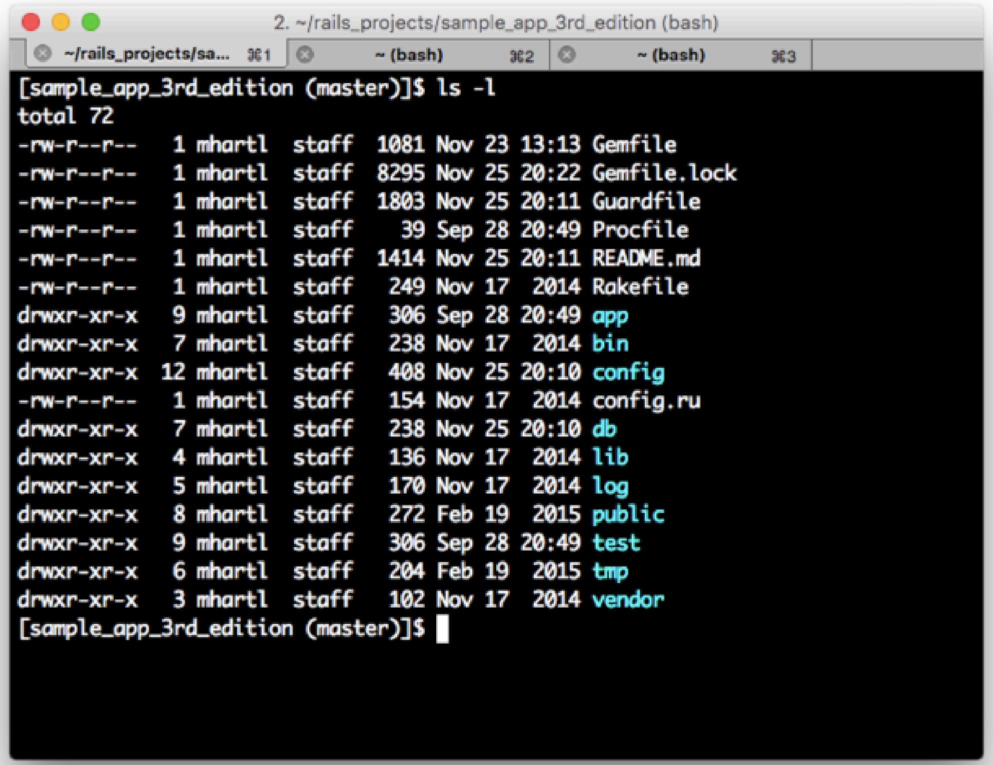
```
$ echo hello
hello
$
```

[4]For example, when developing web applications, in addition to my main command-line tab I often have separate tabs for running a local web server and an automated test suite.



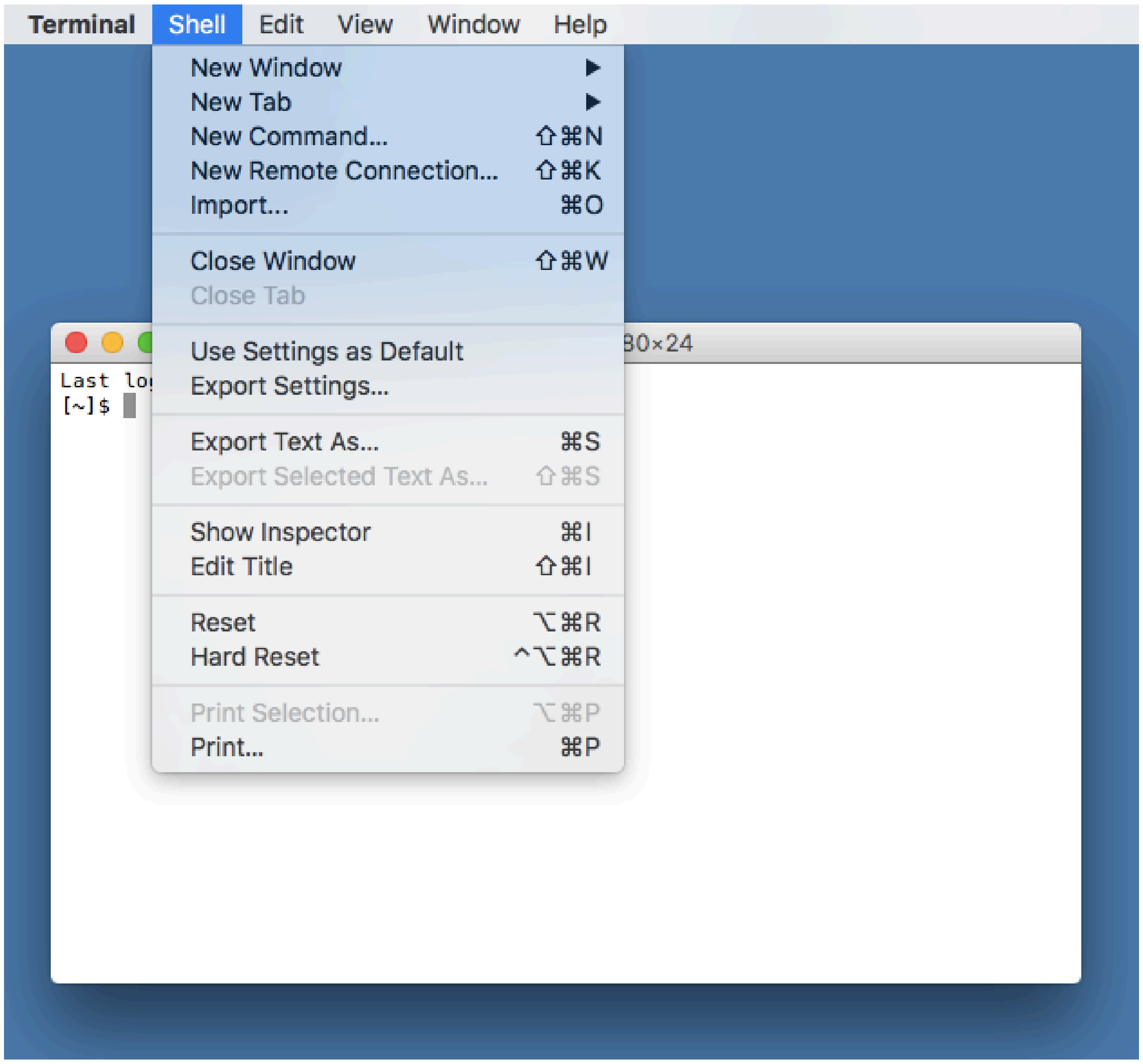
```
[~]$ cd ruby
[ruby]$ ls -a
.          .DS_Store rails
..         projects
[ruby]$ rm -f .DS_Store
[ruby]$ ls -a
.         projects
..        rails
[ruby]$ cd projects/
[projects]$ pwd
/Users/mhartl/ruby/projects
[projects]$
```

Figure 1.6: A series of typical commands.


2. ~/rails_projects/sample_app_3rd_edition (bash)
~/rails_projects/sa... ⌘1
~ (bash) ⌘2
~ (bash) ⌘3

```
[sample_app_3rd_edition (master)]$ ls -l
total 72
-rw-r--r--   1 mhartl  staff  1081 Nov 23 13:13 Gemfile
-rw-r--r--   1 mhartl  staff  8295 Nov 25 20:22 Gemfile.lock
-rw-r--r--   1 mhartl  staff  1803 Nov 25 20:11 Guardfile
-rw-r--r--   1 mhartl  staff    39 Sep 28 20:49 Procfile
-rw-r--r--   1 mhartl  staff  1414 Nov 25 20:11 README.md
-rw-r--r--   1 mhartl  staff   249 Nov 17  2014 Rakefile
drwxr-xr-x   9 mhartl  staff   306 Sep 28 20:49 app
drwxr-xr-x   7 mhartl  staff   238 Nov 17  2014 bin
drwxr-xr-x  12 mhartl  staff   408 Nov 25 20:10 config
-rw-r--r--   1 mhartl  staff   154 Nov 17  2014 config.ru
drwxr-xr-x   7 mhartl  staff   238 Nov 25 20:10 db
drwxr-xr-x   4 mhartl  staff   136 Nov 17  2014 lib
drwxr-xr-x   5 mhartl  staff   170 Nov 17  2014 log
drwxr-xr-x   8 mhartl  staff   272 Feb 19  2015 public
drwxr-xr-x   9 mhartl  staff   306 Sep 28 20:49 test
drwxr-xr-x   6 mhartl  staff   204 Feb 19  2015 tmp
drwxr-xr-x   3 mhartl  staff   102 Nov 17  2014 vendor
[sample_app_3rd_edition (master)]$
```

Figure 1.7: A terminal window with three tabs.



```
Last lo
[~]$
```

Figure 1.8: Some menu items for the default macOS terminal.

(I recommend always typing the commands out yourself, which will let you learn more than if you rely on copying and pasting.) Here we see that `echo hello` prints "hello" and then returns another prompt. Note that, for brevity, I've omitted all characters in the prompt except the dollar sign `$`.

Just to make the pattern clear, let's try a second `echo` command:

```
$ echo "goodbye"
goodbye
$ echo 'goodbye'
goodbye
$
```

Note here that we've wrapped "goodbye" in quotation marks—and we also see that we can use either double quotes, as in `"goodbye"`, or single quotes, as in `'goodbye'`. Such quotes can be used to group strings visually, though in many contexts they are not required by `echo` (Listing 1.2).[5]

Listing 1.2: Printing "hello, goodbye" two different ways.

```
$ echo hello, goodbye
hello, goodbye
$ echo "hello, goodbye"
hello, goodbye
$
```

One thing that can happen when using quotes is accidentally not matching them, as follows:

```
$ echo "hello, goodbye
>
```

At this point, it seems we're stuck. There are specific ways out of this quandary (in fact, in this case you can just add a closing quote and hit return), but it's good to have a general strategy for getting out of trouble (Figure 1.9).[6] This strategy

Figure 1.9: This cat appears to be stuck and should probably hit `Ctrl-C`.

is called “Ctrl-C” (Box 1.4).

Box 1.4. Getting out of trouble

When using the command line, there are lots of things that can get you in trouble, by which I mean the terminal will just hang or otherwise end up in a state that makes entering further commands difficult or impossible. Here are some examples of such commands:

```
$ echo "hello
```

```
$ grep foobar
```

[5] There are subtle differences between the two cases, but they aren’t important at the level of this tutorial. Use your technical sophistication (Box 1.5) if you’re curious. *Hint*: Google searches are your friends.

[6] Image retrieved from https://www.flickr.com/photos/pheezy/5875298232 on 2015-07-19. Copyright © 2011 by Evan P. Cordes and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

```
$ yes
```

```
$ tail
```

```
$ cat
```

In every case, the solution is the same: hit `Ctrl-C` (pronounced "control-see"). Here `Ctrl` refers to the "control" key on your keyboard, and `C` refers to the key labeled "C". `Ctrl-C` thus means "While holding down the control key, press C." In particular, `C` does *not* refer to the capital letter C, so you should not press Shift in addition to Ctrl. (`Ctrl-C` sends a *control code* to the terminal and has nothing to do with producing a capital C when typing normal text.) The result of typing `Ctrl-C` is sometimes written as `^C`, like this:

```
$ tail
^C
```

The origins of `Ctrl-C` are somewhat obscure, but as a mnemonic I like to think of it as meaning "cancel". However you remember it, *do* remember it: when you get into trouble at the command line, your best bet is usually to hit `Ctrl-C`.

Note: When `Ctrl-C` fails, 90% of the time hitting `ESC` (escape) will do the trick.

1.3.1 Exercises

1. Write a command that prints out the string "hello, world". *Extra credit*: As in Listing 1.2, do it two different ways, both with and without using quotation marks.

2. Type the command **`echo 'hello`** (with a mismatched single quote), and then get out of trouble using the technique from Box 1.4.

1.4 Man pages

The program we're using to run a command line, which is technically known as a *shell*,[7] includes a powerful (though often cryptic) tool to learn more about available commands. This tool is itself a command-line command called **`man`** (short for "manual"). Its argument is the name of the command (such as **`echo`**) that we want to learn more about. The details are system-dependent, but on my system the result of running **`man echo`** appears as in Listing 1.3.

Listing 1.3: The result of running **`man echo`**.

```
$ man echo
ECHO(1)          BSD General Commands Manual          ECHO(1)

NAME
   echo -- write arguments to the standard output

SYNOPSIS
   echo [-n] [string ...]

DESCRIPTION
   The echo utility writes any specified operands, separated by single blank
   (` ') characters and followed by a newline (`\n') character, to the stan-
   dard output.

   The following option is available:

   -n  Do not print the trailing newline character. This may also be
      achieved by appending `\c' to the end of the string, as is done by
      iBCS2 compatible systems. Note that this option as well as the
      effect of `\c' are implementation-defined in IEEE Std 1003.1-2001
      (``POSIX.1'') as amended by Cor. 1-2002. Applications aiming for
      maximum portability are strongly encouraged to use printf(1) to
      suppress the newline character.
:
```

On the last line of Listing 1.3, note the presence of a colon **`:`**, which indicates that there is more information below. The details of this last line are system-dependent, but on any system you should be able to access subsequent

[7]Many introductions to the command line cover elements of the shell that require knowledge of a text editor—knowledge which (as noted in the introduction) this tutorial does *not* assume. As a result, we'll defer these important topics to the follow-on tutorials to this one (Section 4.6), starting with *Learn Enough Text Editor to Be Dangerous*.

Figure 1.10: Applying `man` to `man`.

information one line at a time by pressing the down arrow key, or one page at a time by pressing the spacebar. To exit the man page, press "q" (for "quit"). (This interface to the man pages is the same as for the `less` program, which we'll learn about in Section 3.3.)

Because `man` itself is a command, we can apply `man` to `man` (Figure 1.10),[8] as shown in Listing 1.4.

Listing 1.4: The result of running `man man`.

```
$ man man
man(1)                                           man(1)
```

[8]Image retrieved from https://commons.wikimedia.org/wiki/File:Jazz_at_Knicks_man_to_man_defense.jpg on 2015-07-22. Copyright © 2005 by Lordcolus and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

```
NAME
    man - format and display the on-line manual pages

SYNOPSIS
    man [-acdfFhkKtwW] [--path] [-m system] [-p string] [-C config_file]
    [-M pathlist] [-P pager] [-B browser] [-H htmlpager] [-S section_list]
    [section] name ...

DESCRIPTION
    man formats and displays the on-line manual pages. If you specify sec-
    tion, man only looks in that section of the manual. name is normally
    the name of the manual page, which is typically the name of a command,
    function, or file. However, if name contains a slash (/) then man
    interprets it as a file specification, so that you can do man ./foo.5
    or even man /cd/foo/bar.1.gz.

    See below for a description of where man looks for the manual page
    files.

OPTIONS
    -C config_file
:
```

We can see from Listing 1.4 that the synopsis of **`man`** looks something like this:

```
man [-acdfFhkKtwW] [--path] [-m system] [-p string] ...
```

This is what I meant above when I described man pages as "often cryptic". Indeed, in many cases I find the details of man pages to be almost impossible to understand, but being able to scan over the man page to get a high-level overview of a command is a valuable skill, one well worth acquiring. To get used to reading man pages, I recommend running **`man <command name>`** when encountering a new command. Even if the details aren't entirely clear, reading the man pages will help develop the valuable skill of *technical sophistication* (Box 1.5).

Box 1.5. Technical sophistication

In mathematics, many subjects can be developed by applying pure deduction to a small number of assumptions, or *axioms*; examples include algebra, geometry, number theory, and analysis. As a result, such subjects are completely self-contained, and thus have no formal prerequisites—in principle, even a small child could learn them. In practice, though, something else is required, and mathematicians often recommend the informal prerequisite of *mathematical maturity*, which consists of the experience and general sophistication needed to understand and write mathematical proofs.

In technology, a similar skill (or, more accurately, set of skills) exists in the form of *technical sophistication*. In addition to "hard skills" like familiarity with text editors and the Unix command line, technical sophistication includes "soft skills" like looking for promising menu items and knowing the kinds of search terms to drop into Google (as illustrated in "Tech Support Cheat Sheet" from xkcd), along with an *attitude* of doing what it takes to make the machine do our bidding (Box 1.1).

These soft skills, and this attitude, are hard to teach directly, so as you progress through this and subsequent Learn Enough tutorials you should always be on the lookout for opportunities to increase your technical sophistication (such as, for example, learning how to get the gist of a program by scanning its man page (Section 1.4)). Over time, the cumulative effect will be that, like the author of "Tech Support Cheat Sheet", you'll have the seemingly magical ability to do everything in every program.

By the way, "Tech Support Cheat Sheet" is missing three important techniques for solving common problems:

1. Have you restarted the application?
2. Have you rebooted the device?
3. Have you tried uninstalling and reinstalling the app?

These steps are generally presented in the order you should try them, and #2 alone probably solves 90% of unexplained computer errors.

1.4.1 Exercises

1. According to the man page, what are the official short and long descriptions of `echo` on your system?

2. As seen in Listing 1.2, by default the `echo` command prints its argument to the screen and then puts the new prompt on a new line. The way it does this is by appending a special character called a *newline* (a special character that literally puts the string on a new line, written in many contexts as "backslash n" `\n`). Because `echo` is often used in programs to print out a sequence of strings *not* separated by newlines, there is a special command-line option to prevent the newline from being inserted.

   By reading the man page for `echo`, determine the command needed to print out "hello" *without* the trailing newline, and verify using your terminal that it works as expected. *Hints*: To determine the placement of the command-line option, it may help to refer to Figure 1.5. By comparing your result with Listing 1.5 and Listing 1.6, you should be able to verify that you've used the option properly. (*Note*: This exercise may fail when using the default terminal program on some older versions of macOS. In this case, I recommend installing iTerm (which isn't a bad idea anyway).)

Listing 1.5: The result of running `echo` with a newline (without option).

```
hello
[~]$
```

Listing 1.6: The result of running `echo` without a newline (*with* option).

```
hello[~]$
```

1.5 Editing the line

Command lines include several features to make it easy to repeat previous commands, possibly in edited form. These and many other command-line features

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

Table 1.1: Miscellaneous keyboard symbols.

often involve special keys on the keyboard, so for reference Table 1.1 shows these symbols for the various keys on a typical Macintosh keyboard. Apply your technical sophistication (Box 1.5) if your keyboard differs.

One of the most useful ways to edit the line is to "up arrow" ↑, which simply retrieves the previous command. Pressing up arrow again moves further up the list of commands, while "down arrow" ↓ goes back toward the bottom.

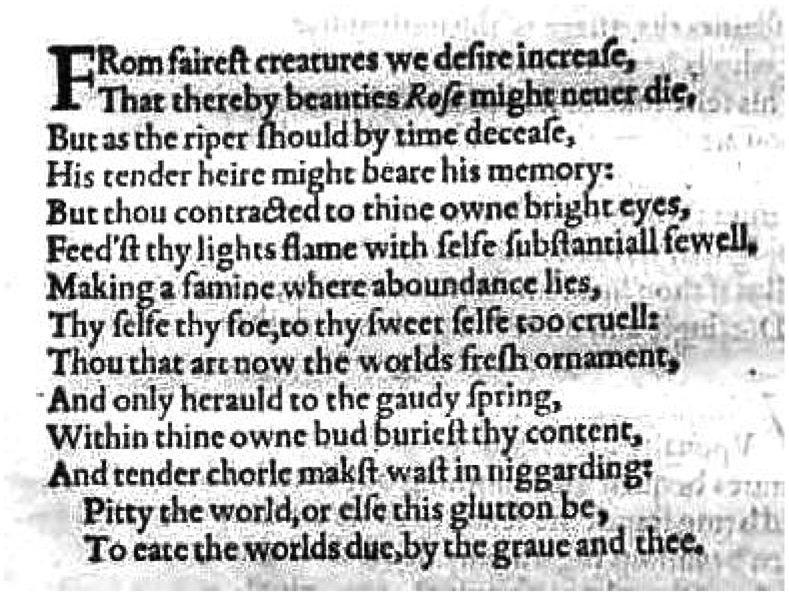
Other common ways to edit the line use the control key, which (as we saw in Box 1.4) is usually written as **`Ctrl`** or `^`. For example, when typing a new command, or dealing with a previous command, it is often convenient to be able to move quickly within the line. Suppose we typed

```
$ goodbye
```

only to realize that we wanted to put **`echo`** in front of it. We could use the left arrow key ← to get to the beginning of the line, but it's easier to type **`^A`**, which takes us there immediately. Similarly, **`^E`** moves to the end of the line.[9] Finally, **`^U`** clears to the beginning of the line and lets us start over.

The combination of **`^A`**, **`^E`**, and **`^U`** will work on most systems, but they don't do you much good if you're editing a longer line, such as this one containing the first line of Sonnet 1 by William Shakespeare (Listing 1.7).

[9]There are also commands for moving one *word* at a time (the keyboard sequences `ESC F` and `ESC B`), but I hardly ever use them myself, so it's clear they are not required to be *dangerous*.



FRom faireſt creatures we deſire increaſe,
That thereby beauties *Roſe* might neuer die,
But as the riper ſhould by time deceaſe,
His tender heire might beare his memory:
But thou contracted to thine owne bright eyes,
Feed'ſt thy lights flame with ſelfe ſubſtantiall fewell,
Making a famine where aboundance lies,
Thy ſelfe thy foe,to thy ſweet ſelfe too cruell:
Thou that art now the worlds freſh ornament,
And only herauld to the gaudy ſpring,
Within thine owne bud burieſt thy content,
And tender chorle makſt waſt in niggarding:
Pitty the world,or elſe this glutton be,
To eate the worlds due,by the graue and thee.

Figure 1.11: The original appearance of Shakespeare's first sonnet.

Listing 1.7: Printing the first line of Shakespeare's first sonnet.

```
$ echo "From fairest creatures we desire increase,"
```

Suppose we wanted to change "From" to "FRom" to more closely match the text from the original sonnet (Figure 1.11).[10] We could type `^A` followed by the right arrow key a few times, but on some systems it's possible to move directly to the desired spot by combining the keyboard and mouse via Option-click. That is, you can hold down the Option key on your keyboard (if it exists),[11] and then click with the mouse pointer on the place in the command where you want the cursor. This would let us move right to the "o" in "From", allowing us to delete the "r" and yielding Listing 1.8 directly.

[10]Note that the Original Pronunciation (OP) of Shakespearean English is different from modern pronunciation. Generally speaking, Shakespeare's sonnets include many word pairs that don't rhyme in modern English but do in OP. In the case of Figure 1.11, the word "memory" should be pronounced "MEM-or-aye", leading to a rhyme between lines 2 and 4 (ending in *die* and *memory*, respectively).

[11]Some keyboards lack an Option key, so obviously this trick won't work on such systems.

Listing 1.8: The result of editing a longer command-line command.

```
$ echo "FRom fairest creatures we desire increase,"
```

I usually move around the command line with a combination of `^A`, `^E`, and right & left arrow keys, but for longer commands Option-click can be a big help. (I also frequently change my mind about the exact command I'm typing, in which case I usually find that hitting `^U` and starting over again is the fastest way to proceed.)

1.5.1 Exercises

1. Using the up arrow, print to the screen the strings "fee", "fie", "foe", and "fum" without retyping `echo` each time.

2. Starting with the line in Listing 1.7, use any combination of `^A`, `^E`, arrow keys, or Option-click to change the occurrences of the short s to the archaic long s "s" in order to match the appearance of the original (Figure 1.11). In other words, the argument to `echo` should read "FRom fairest creatures we desire increase,". *Hint*: It's unlikely that your keyboard can produce "s" natively, so either copy it from the text of this tutorial or Google for it and copy it from the Internet. (If you have trouble copying and pasting into your terminal, I suggest applying the ideas in Box 1.5 to figure out how to do it on your system.)

1.6 Cleaning up

When using the command line, sometimes it's convenient to be able to clean up by clearing the screen, which we can do with `clear`:

```
$ clear
```

A keyboard shortcut for this is `^L`.

Command	Description	Example
`echo <string>`	Print string to screen	`$ echo hello`
`man <command>`	Display manual page for command	`$ man echo`
`^C`	Get out of trouble	`$ tail ^C`
`^A`	Move to beginning of line	
`^E`	Move to end of line	
`^U`	Delete to beginning of line	
Option-click	Move cursor to location clicked	
Up & down arrow	Scroll through previous commands	
`clear` or `^L`	Clear screen	`$ clear`
`exit` or `^D`	Exit terminal	`$ exit`

Table 1.2: Important commands from Chapter 1.

Similarly, when we are done with a terminal window (or tab) and are ready to exit, we can use the **`exit`** command:

```
$ exit
```

A keyboard shortcut for this is **`^D`**.

1.6.1 Exercises

1. Clear the contents of the current tab.
2. Open a new tab, execute **`echo 'hello'`**, and then exit.

1.7 Summary

Important commands from this section are summarized in Table 1.2.

1.7.1 Exercises

1. Write a command to print the string **`Use "man echo"`**, *including* the quotes; i.e., take care not to print out **`Use man echo`** instead. *Hint*:

Use double quotes in the inner string, and wrap the whole thing in single quotes.

2. By running `man sleep`, figure out how to make the terminal “sleep” for 5 seconds, and execute the command to do so.
3. Execute the command to sleep for 5000 seconds, realize that’s well over an hour, and then use the instructions from Box 1.4 to get out of trouble.

Chapter 2

Manipulating files

Having covered how to run a basic command, we're now ready to learn how to manipulate files, one of the most important tasks at the command line. Because this tutorial assumes no technical prerequisites, we're not going to require any familiarity with programs designed to edit text. (Such programs, called *text editors*, are the subject of the next tutorial in the sequence, *Learn Enough Text Editor to Be Dangerous*.) This means that we'll need to create files by hand at the command line. But this is a feature, not a bug (Box 2.1), because learning to create files at the command line is a valuable skill in itself.

Box 2.1. Learning to speak "geek"

One important part of learning software development is becoming familiar with the hacker, nerd, and geek culture from which much of it springs. For example, the phrase "It's not a bug, it's a feature" is a common way of recasting a seeming flaw as a virtue. As Urban Dictionary puts it:

> **It's not a bug, it's a feature**
>
> Excuse made by software developers when they try to convince the user that a flaw in their program is actually what it's supposed to be doing.

The Jargon File, which includes an enormous and entertaining lexicon of hacker terms, expands on this theme in its entry on *feature*:

> "Undocumented feature" is a common, allegedly humorous euphemism for a *bug*. There's a related joke that is sometimes referred to as the "one-question geek test". You say to someone "I saw a Volkswagen Beetle today with a vanity license plate that read FEATURE". If he/she laughs, he/she is a *geek*.

The joke here is that, because "bug" is a common slang term for a Volkswagen Beetle, a Beetle with the vanity plate FEATURE is a real-life manifestation of "It's not a bug, it's a feature" (Figure 2.1).

Even if you're not a geek or nerd yourself, learning to "speak geek" will help you navigate both the technological landscape and the social world that surrounds it.

2.1 Redirecting and appending

Let's pick up (more or less) where we left off in Chapter 1, with an **echo** command to print out the first line of Shakespeare's first sonnet (Listing 1.7):

```
$ echo "From fairest creatures we desire increase,"
From fairest creatures we desire increase,
```

Our task now is to create a file containing this line. Even without the benefit of a text editor, it is possible to do this using the *redirect operator* **>**:

```
$ echo "From fairest creatures we desire increase," > sonnet_1.txt
```

Figure 2.1: It’s not a bug, it’s a feature.

(Recall that you can use up arrow to retrieve the previous command rather than typing it from scratch.) Here the right angle bracket **>** takes the string output from **echo** and redirects its contents to a file called **sonnet_1.txt**.

How can we tell if the redirect worked? We'll learn some more advanced command-line tools for inspecting files in Chapter 3, but for now we'll use the **cat** command, which simply dumps the contents of the file to the screen:

```
$ cat sonnet_1.txt
From fairest creatures we desire increase,
```

The name **cat** is short for "concatenate", which is a hint that it can be used to combine the contents of multiple files, but the usage above (to dump the contents of a single file to the screen) is extremely common. Think of **cat** as a "quick-and-dirty" way to view the contents of a particular file (Figure 2.2).[1]

In order to add the second line of the sonnet (in modernized spelling), we can use the *append operator* **>>** as follows:

```
$ echo "That thereby beauty's Rose might never die," >> sonnet_1.txt
```

This just adds the line to the end of the given file. As before, we can see the result using **cat**:

```
$ cat sonnet_1.txt
From fairest creatures we desire increase,
That thereby beauty's Rose might never die,
```

(To get to this command, I hope you just hit up arrow twice instead of retyping it. If so, you're definitely getting the hang of this.) The result above shows that the double right angle bracket **>>** appended the string from **echo** to the file **sonnet_1.txt** as expected.

Modernized treatments of the *Sonnets* sometimes emend *Rose* to *rose* (thereby obscuring the likely meaning), and we can make a second file following this convention using two more calls to **echo**:

[1]Image retrieved from https://www.flickr.com/photos/tiffanyday/4076289684 on 2015-07-22. Copyright 2009 by Tiffa Day and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 2.2: Viewing a file with **`cat`**.

```
$ echo "From fairest creatures we desire increase," > sonnet_1_lower_case.txt
$ echo "That thereby beauty's rose might never die," >> sonnet_1_lower_case.txt
```

In order to facilitate the comparison of files that are similar but not identical, Unix systems come with the helpful **`diff`** command:

```
$ diff sonnet_1.txt sonnet_1_lower_case.txt
< That thereby beauty's Rose might never die,
---
> That thereby beauty's rose might never die,
```

When discussing computer files, *diff* is frequently employed both as a noun ("What's the diff between those files?") and as a verb ("You should diff the files to see what changed."). As with many technical terms, this sometimes bleeds over into common usage, such as "Diff present ideas against those of various past cultures, and see what you get."[2]

2.1.1 Exercises

At the end of each of the exercises below, use the **`cat`** command to verify your answer.

1. Using **`echo`** and **`>`**, make files called **`line_1.txt`** and **`line_2.txt`** containing the first and second lines of Sonnet 1, respectively.

2. Replicate the original **`sonnet_1.txt`** (containing the first two lines of the sonnet) by first redirecting the contents of **`line_1.txt`** and then appending the contents of **`line_2.txt`**. Call the new file **`sonnet_1_copy.txt`**, and confirm using **`diff`** that it's identical to **`sonnet_1.txt`**.

[2]From the essay "What You Can't Say" by Paul Graham (2004). As Graham notes:

> The verb "diff" is computer jargon, but it's the only word with exactly the sense I want. It comes from the Unix `diff` utility, which yields a list of all the differences between two files. More generally it means an unselective and microscopically thorough comparison between two versions of something.

Hint: When there is no diff between two files, **`diff`** simply outputs nothing.

3. Use **`cat`** to combine the contents of **`line_1.txt`** and **`line_2.txt`** in reverse order using a *single* command, yielding the file **`sonnet_1_reversed.txt`**. *Hint*: The **`cat`** command can take multiple arguments.

2.2 Listing

Perhaps the mostly frequently typed command on the Unix command line is **`ls`**, short for "list" (Listing 2.1).

Listing 2.1: Listing files and directories with **`ls`**. (Output will vary.)

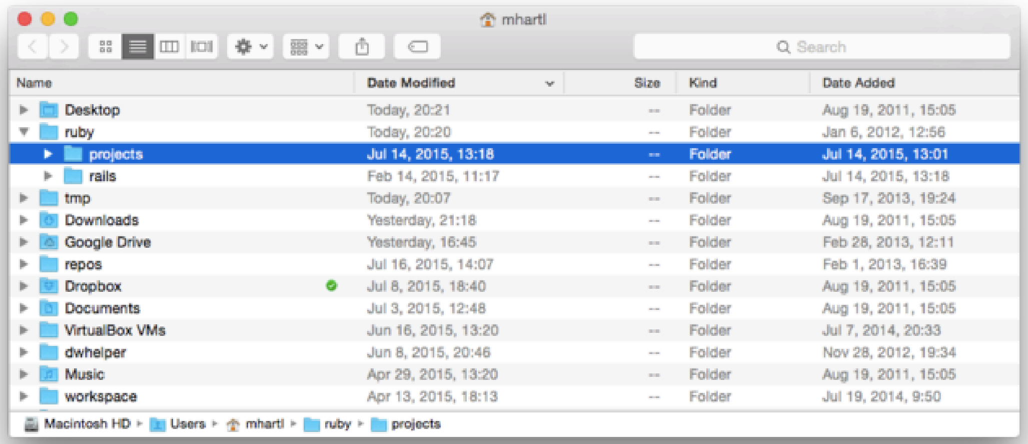
```
$ ls
Desktop
Downloads
sonnet_1.txt
sonnet_1_reversed.txt
```

The **`ls`** command simply lists all the files and directories in the current directory (except for those that are *hidden*, which we'll learn more about in a moment). In this sense, it's effectively a command-line version of the graphical browser used to show files and directories (also called "folders"), as seen in Figure 2.3. (We'll sharpen our understanding of directories and folders in Chapter 4.) As with a graphical file browser, the output in Listing 2.1 is just a sample, and results will differ based on the details of your system. (This goes for all the **`ls`** examples, so don't be concerned if there are minor differences in output.)

The **`ls`** command can be used to check if a file (or directory) exists, because trying to **`ls`** a nonexistent file results in an error message, as seen in Listing 2.2.

Listing 2.2: Running **`ls`** on a nonexistent file.

```
$ ls foo
ls: foo: No such file or directory
$ touch foo
$ ls foo
foo
```



Name	Date Modified	Size	Kind	Date Added
Desktop	Today, 20:21	--	Folder	Aug 19, 2011, 15:05
ruby	Today, 20:20	--	Folder	Jan 6, 2012, 12:56
projects	Jul 14, 2015, 13:18	--	Folder	Jul 14, 2015, 13:01
rails	Feb 14, 2015, 11:17	--	Folder	Jul 14, 2015, 13:18
tmp	Today, 20:07	--	Folder	Sep 17, 2013, 19:24
Downloads	Yesterday, 21:18	--	Folder	Aug 19, 2011, 15:05
Google Drive	Yesterday, 16:45	--	Folder	Feb 28, 2013, 12:11
repos	Jul 16, 2015, 14:07	--	Folder	Feb 1, 2013, 16:39
Dropbox	Jul 8, 2015, 18:40	--	Folder	Aug 19, 2011, 15:05
Documents	Jul 3, 2015, 12:48	--	Folder	Aug 19, 2011, 15:05
VirtualBox VMs	Jun 16, 2015, 13:20	--	Folder	Jul 7, 2014, 20:33
dwhelper	Jun 8, 2015, 20:46	--	Folder	Nov 28, 2012, 19:34
Music	Apr 29, 2015, 13:20	--	Folder	Aug 19, 2011, 15:05
workspace	Apr 13, 2015, 18:13	--	Folder	Jul 19, 2014, 9:50

Figure 2.3: The graphical equivalent of **`ls`**.

Listing 2.2 uses the **`touch`** command to create an empty file with the name **`foo`** (Box 2.2), so the second time we run **`ls`** the error message is gone. (The stated purpose of **`touch`** is to change the modification time on files or directories, but (ab)using **`touch`** to create empty files as in Listing 2.2 is a common Unix idiom.)

Box 2.2. Foo, bar, baz, etc.

When reading about computers, you will encounter certain strange words—words like *foo*, *bar*, and *baz*—with surprising frequency. Indeed, in addition to `ls foo` and `touch foo`, we have already seen four such references in this tutorial: in a typical command-line command (Figure 1.2), in a virtual machine password (`foobar!`, Box 1.2), again when getting out of trouble (`grep foobar`, Box 1.4), and yet again in a man page (Listing 1.4). The first three were my own uses, but the last I had nothing to do with:

```
...if name contains a slash (/) then man interprets
```

```
it as a file specification,  so that you can do man
./foo.5 or even man /cd/foo/bar.1.gz.
```

Here we see both *foo* and *bar* making an appearance in the man page for `man` itself—an unambiguous testament to their ubiquity in computing.

What is the origin of these odd terms? As usual, the Jargon File (via its entry on *foo*) enlightens us:

> **foo**: /foo/
>
> 1. interj. Term of disgust.
>
> 2. [very common] Used very generally as a sample name for absolutely anything, esp. programs and files (esp. scratch files).
>
> 3. First on the standard list of metasyntactic variables used in syntax examples. See also bar, baz, qux, quux, etc.
>
> When 'foo' is used in connection with 'bar' it has generally traced to the WWII-era Army slang acronym *FUBAR* [see original meaning], later modified to *foobar*. Early versions of the Jargon File interpreted this change as a post-war bowdlerization, but it now seems more likely that FUBAR was itself a derivative of 'foo' perhaps influenced by German *furchtbar* (terrible) — 'foobar' may actually have been the *original* form.

Following the link to metasyntactic variables, we then find the following:

> **metasyntactic variable**: n.
>
> A name used in examples and understood to stand for whatever thing is under discussion, or any random member of a class of things under discussion. The word foo is the canonical example. To avoid confusion, hackers never (well, hardly ever) use 'foo' or other words like it as permanent names for anything. In filenames, a common convention is that any filename beginning with a metasyntactic-variable name is a scratch file that may be deleted at any time.

> Metasyntactic variables are so called because (1) they are variables in the metalanguage used to talk about programs, etc.; (2) they are variables whose values are often variables (as in usages like “the value of f(foo,bar) is the sum of foo and bar”). However, it has been plausibly suggested that the real reason for the term “metasyntactic variable” is that it sounds good.

In other words, if you want to create a file, and the name doesn’t matter, the name is usually “foo”. Once you’ve used “foo”, the next file is called “bar”, the one after that “baz”. Continuations from there vary (“quux” is one common choice), but in many cases three is enough.

A common pattern when using the command line is changing directories using **`cd`** (covered in Chapter 4) and then immediately typing **`ls`** to view the contents of the directory. This lets us orient ourselves, and is a good first step toward whatever our next action might be.

One useful ability of **`ls`** is support for the *wildcard character* **`*`** (read “star”). For example, to list all files ending in “.txt”, we would type this:

```
$ ls *.txt
sonnet_1.txt
sonnet_1_reversed.txt
```

Here **`*.txt`** (read “star dot tee-ex-tee”) automatically expands to all the filenames that match the pattern “any string followed by .txt”.

There are three particularly important optional forms of **`ls`**, starting with the “long form”, using the option **`-l`** (read “dash-ell”):

```
$ ls -l *.txt
total 16
-rw-r--r-- 1 mhartl staff  87 Jul 20 18:05 sonnet_1.txt
-rw-r--r-- 1 mhartl staff 294 Jul 21 12:09 sonnet_1_reversed.txt
```

For now, you can safely ignore most of the information output by **`ls -l`**, but note that the long form lists a date and time indicating the last time the file was modified. The number before the date is the *size* of the file, in bytes.[3]

A second powerful **`ls`** variant is "list by **r**eversed **t**ime of modification (**l**ong format)", or **`ls -rtl`**, which lists the long form of each file or directory in order of how recently it was modified (*reversed* so that the most recently modified entries appear at the bottom of the screen for easy inspection). This is particularly useful when there are a lot of files in the directory but you really only care about seeing the ones that have been modified recently, such when confirming a file download. We'll see an example of this in Section 3.1, but you are free to try it now:

```
$ ls -rtl
<results system-dependent>
```

By the way, **`-rtl`** is the commonly used compact form, but you can also pass the options individually, like this:

```
$ ls -r -t -l
```

In addition, their order is irrelevant, so typing **`ls -trl`** gives the same result.

2.2.1 Hidden files

Finally, Unix has the concept of "hidden files (and directories)", which don't show up by default when listing files. Hidden files and directories are identified by starting with a dot **`.`**, and are commonly used for things like storing user preferences. For example, in *Learn Enough Git to Be Dangerous*, we'll create a file called **`.gitignore`** that tells a particular program (Git) to ignore files matching certain patterns. As a concrete example, to ignore all files ending in ".txt", we could do this:

[3] A *bit* is one piece of yes-or-no information (such as a 1 or a 0), and a *byte* is eight bits. Bytes are probably most familiar from "megabytes" and "gigabytes", which represent a million and a billion bytes, respectively. (The official story is a little more complicated, but level of detail here is certainly enough to be *dangerous*.)

```
$ echo "*.txt" > .gitignore
$ cat .gitignore
*.txt
```

If we then run `ls`, the file won't show up, because it's hidden:

```
$ ls
sonnet_1.txt
sonnet_1_reversed.txt
```

To get `ls` to display hidden files and directories, we need to pass it the `-a` option (for "all"):

```
$ ls -a
.            .gitignore      sonnet_1_reversed.txt
..           sonnet_1.txt
```

Now `.gitignore` shows up, as expected. (We'll learn what `.` and `..` refer to in Section 4.3.)

2.2.2 Exercises

1. What's the command to list all the non-hidden files and directories that start with the letter "s"?
2. What is the command to list all the non-hidden files that contain the string "onnet", long-form by reverse modification time? *Hint*: Use the wildcard operator at both the beginning and the end.
3. What is the command to list *all* files (including hidden ones) by reverse modification time, in long form?

2.3 Renaming, copying, deleting

Next to listing files, probably the most common file operations involve renaming, copying, and deleting them. As with listing files, most modern operating systems provide a graphical user interface to such tasks, but in many contexts it is more convenient to perform them at the command line. *Note*: If you're using macOS, you should follow the instructions in Box 2.3 at this time.

Box 2.3. Switching macOS to Bash

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

The first step is to determine which shell your system is running, which you can do using the `echo` command (Section 1.3):

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

This prints out the `$SHELL` environment variable. If you see the result shown above, indicating that you're already using Bash, you're done and can proceed with the rest of the tutorial. (In rare cases, `$SHELL` may differ from the current shell, but the procedure below will still correctly change from one shell to another.) The alert shown in Listing 1.1 is safe to ignore. For more information, including how to switch to and use Z shell with this tutorial, see the Learn Enough blog post "Using Z Shell on Macs with the Learn Enough Tutorials".

The other possible result of `echo` is this:

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

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

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

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

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

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

At this point, you will probably start seeing the alert shown in Listing 1.1, which you should ignore.

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

The way to rename a file is with the **`mv`** command, short for "move":

```
$ echo "test text" > test
$ mv test test_file.txt
$ ls
test_file.txt
```

This renames the file called **`test`** to **`test_file.txt`**. The final step in the example runs **`ls`** to confirm that the file renaming was successful, but system-specific files other than the test file are omitted from the output shown. (The name "move" comes from the general use of **`mv`** to move a file to a different directory (Chapter 4), possibly renaming it en route. When the origin and target directories coincide, such a "move" reduces to a simple renaming.)

The way to copy a file is with **`cp`**, short for "copy":

```
$ cp test_file.txt second_test.txt
$ ls
second_test.txt
test_file.txt
```

Finally, the command for deleting a file is `rm`, for “remove”:

```
$ rm second_test.txt
remove second_test.txt? y
$ ls second_test.txt
ls: second_test.txt: No such file or directory
```

Note that, on many systems, by default you will be prompted to confirm the removal of the file. Any answer starting with the letter “y” or “Y” will cause the file to be deleted, and any other answers will prevent the deletion from occurring.

By the way, in the calls to `cp` and `rm` above, I would almost certainly not type out `test_file.txt` or `second_test.txt`. Instead, I would type something like `test`⇥ or `sec`⇥ (where ⇥ represents the tab key (Table 1.1)), thereby making use of *tab completion* (Box 2.4).

Box 2.4. Tab completion

Most modern command-line programs (shells) support *tab completion*, which involves automatically completing a word if there’s only one valid match on the system. For example, if the only file starting with the letters “tes” is `test_file`, we could create the command to remove it as follows:

```
$ rm tes⇥
```

where ⇥ is the tab key (Table 1.1). The shell would then complete the filename, yielding `rm test_file`. Especially with longer filenames (or directories), tab completion can save a huge amount of typing. It also lowers the *cognitive load*, since it means you don't have to remember the full name of the file—only its first few letters.

If the match is ambiguous, as would happen if we had files called `foobarquux` and `foobazquux`, the word will be completed only as far as possible, so

```
$ ls foo⇥
```

would be completed to

```
$ ls fooba
```

If we then hit tab *again*, we would see a list of matches:

```
$ ls fooba⇥
foobarquux foobazquux
```

We could then type more letters to resolve the ambiguity, so typing the `r` after `fooba` and hitting ⇥ would yield

```
$ ls foobar⇥
```

which would be completed to `foobarquux`. This situation is common enough that experienced command-line users will often just hit something like `f`⇥⇥ to get the shell to show all the possibilities:

```
$ ls f⇥⇥
figure_1.png foobarquux  foobazquux
```

Additional letters would then be typed as usual to resolve the ambiguity.

The default behavior of **`rm`** on an unconfigured Unix system is actually to remove the file without confirmation, but (because deletion is irreversible) many systems *alias* the **`rm`** command to use an option to turn on confirmation. (As you can verify by running **`man rm`**, this option is **`-i`**, so in fact **`rm`** is really **`rm -i`**.) There are many situations where confirmation is inconvenient, though, such as when you're deleting a list of files and don't want to have to confirm each one. This is especially common when using the wildcard **`*`** introduced in Section 2.2. For example, to remove all the files ending with ".txt" using a single command, *without* having to confirm each one, you can type this:

```
$ rm -f *.txt
```

Here **`-f`** (for "force") overrides the implicit **`-i`** option and removes all files immediately. (N.B. You are now in a position to understand the command in Figure 1.2.)

2.3.1 Unix terseness

One thing you might notice is that the commands in this section and in Section 2.2 are short: instead of **`list`**, **`move`**, **`copy`**, and **`remove`**, we have **`ls`**, **`mv`**, **`cp`**, and **`rm`**. Because the former command names are easier to understand and memorize, you may wonder why the actual commands aren't longer (Figure 2.4).

The answer is that Unix dates from a time when most computer users logged on to centralized servers over slow connections, and there could be a noticeable delay between the time users pressed a key and the time it appeared on the terminal. For frequently used commands like listing files, the difference between **`list`** and **`ls`** or **`remove`** and **`rm`** could be significant. As a result, the most commonly used Unix commands tend to be only two or three letters long. Because it makes them more difficult to memorize, this can be a minor inconvenience when learning them, but over a lifetime of command-line use the savings represented by, say, **`mv`** really add up.

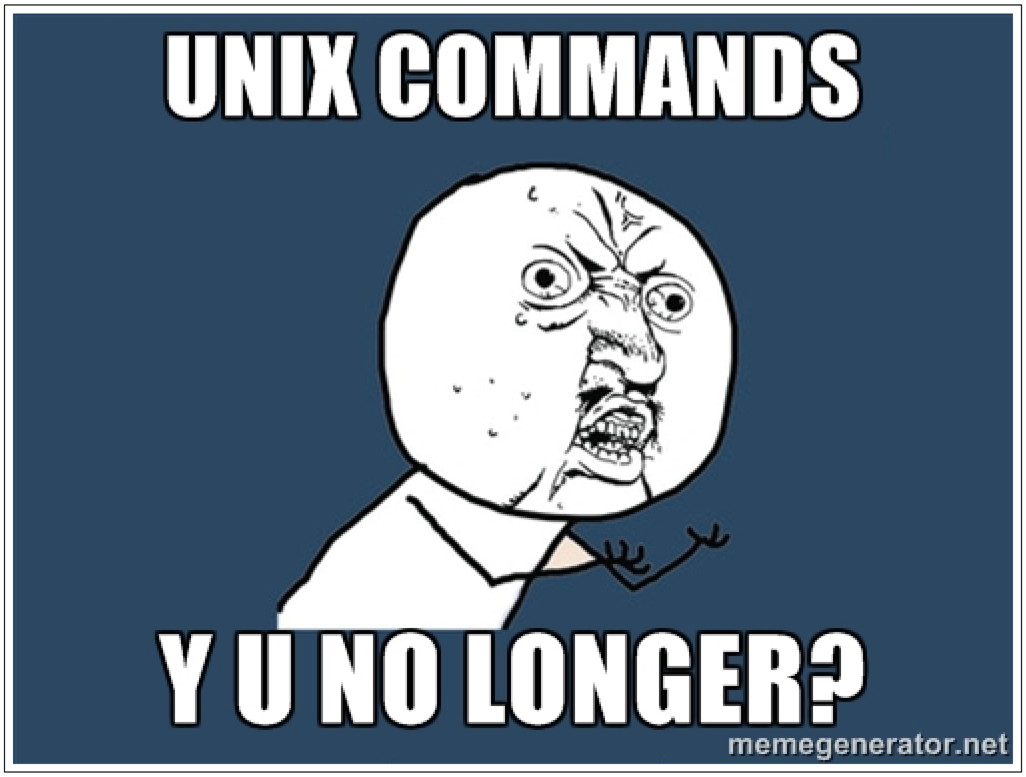


Figure 2.4: The terseness of Unix commands can be a source of confusion.

2.3.2 Exercises

1. Use the **`echo`** command and the redirect operator **`>`** to make a file called **`foo.txt`** containing the text "hello, world". Then, using the **`cp`** command, make a copy of **`foo.txt`** called **`bar.txt`**. Using the **`diff`** command, confirm that the contents of both files are the same.

2. By combining the **`cat`** command and the redirect operator **`>`**, create a copy of **`foo.txt`** called **`baz.txt`** *without* using the **`cp`** command.

3. Create a file called **`quux.txt`** containing the contents of **`foo.txt`** followed by the contents of **`bar.txt`**. *Hint*: As noted in Section 2.1.1, **`cat`** can take multiple arguments.

4. How do **`rm nonexistent`** and **`rm -f nonexistent`** differ for a nonexistent file?

2.4 Summary

Important commands from this section are summarized in Table 2.1.

2.4.1 Exercises

1. By copying and pasting the text from the HTML version of Figure 2.5, use **`echo`** to make a file called **`sonnet_1_complete.txt`** containing the full (original) text of Shakespeare's first sonnet. *Hint*: You may recall getting stuck when **`echo`** was followed by an unmatched double quote (Section 1.3 and Box 1.4), as in **`echo "`**, but in fact this construction allows you to print out a multi-line block of text. Just remember to put a closing quote at the end, and then redirect to a file with the appropriate name. Check that the contents are correct using **`cat`** (Figure 2.2).

2. Type the sequence of commands needed to create an empty file called **`foo`**, rename it to **`bar`**, and copy it to **`baz`**.

Command	**Description**	**Example**
`>`	Redirect output to filename	`$ echo foo > foo.txt`
`>>`	Append output to filename	`$ echo bar >> foo.txt`
`cat <file>`	Print contents of file to screen	`$ cat hello.txt`
`diff <f1> <f2>`	Diff files 1 & 2	`$ diff foo.txt bar.txt`
`ls`	List directory or file	`$ ls hello.txt`
`ls -l`	List long form	`$ ls -l hello.txt`
`ls -rtl`	Long by reverse modification time	`$ ls -rtl`
`ls -a`	List all (including hidden)	`$ ls -a`
`touch <file>`	Create an empty file	`$ touch foo`
`mv <old> <new>`	Rename (move) from old to new	`$ mv foo bar`
`cp <old> <new>`	Copy old to new	`$ cp foo bar`
`rm <file>`	Remove (delete) file	`$ rm foo`
`rm -f <file>`	Force-remove file	`$ rm -f bar`

Table 2.1: Important commands from Chapter 2.

FRom fairest creatures we desire increase,
That thereby beauties *Rose* might neuer die,
But as the riper should by time decease,
His tender heire might beare his memory:
But thou contracted to thine owne bright eyes,
Feed'st thy lights flame with selfe substantiall fewell,
Making a famine where aboundance lies,
Thy selfe thy foe,to thy sweet selfe too cruell:
Thou that art now the worlds fresh ornament,
And only herauld to the gaudy spring,
Within thine owne bud buriest thy content,
And tender chorle makst wast in niggarding:
Pitty the world,or else this glutton be,
To eate the worlds due,by the graue and thee.

Figure 2.5: A copy-and-pastable version of Shakespeare's first sonnet (*cf.* Figure 1.11).

3. What is the command to list only the files starting with the letter “b”? *Hint*: Use a wildcard.

4. Remove both **`bar`** and **`baz`** using a *single* call to **`rm`**. *Hint*: If those are the only two files in the current directory that start with the letter “b”, you can use the wildcard pattern from the previous exercise.

Chapter 3

Inspecting files

Having seen how to create and manipulate files, now it's time to learn how to examine their contents. This is especially important for files too long to fit on a single screen. In particular, we saw starting in Section 2.1 how to use the **`cat`** command to dump the file contents to the screen, but this doesn't work very well for longer files.

3.1 Downloading a file

To give us a place to start, rather than creating a long file by hand (which is cumbersome) we'll download a file from the Internet using the powerful **`curl`** utility (sometimes written as "cURL"), which allows us to interact with a URL[1] at the command line. Although it's not part of the core Unix command set, the **`curl`** command is widely available on Unix systems. To make sure it's available on your system, we can use the **`which`** command, which looks to see if the given program is available at the command line.[2] The way to use it is to type **`which`** followed by the name of the program—in this case, **`curl`**:

[1]URL is short for Uniform Resource Locator, and in practice usually just means "web address".

[2]Technically, **`which`** locates a file on the user's *path*, which is a list of directories where executable programs are located.

```
$ which curl
/usr/bin/curl
```

I've shown the output on my system (`/usr/bin/curl`, usually read as "user bin curl"), but the result on your system may differ. In particular, if the result is just a blank line, you will have to install `curl`, which you can do by Googling for "install curl" followed by the name of your operating system.

Once `curl` is installed, we can download a file called `sonnets.txt` containing a large corpus of text using the command in Listing 3.1.[3]

Listing 3.1: Using `curl` to download a longer file.

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

Be sure to copy the command exactly; in particular, note that the option `-OL` contains a capital letter "O" (`O`) and not a zero (`0`). (Figuring out what these options do is left as an exercise (Section 3.5.1).) Also, on some systems (for mysterious reasons) you might have to run the command twice to get it to work; by inspecting the results of `ls -rtl`, you should be able to tell if the initial call to `curl` created the file `sonnets.txt` as expected. (If you do have to repeat the `curl` command, you could press up arrow twice to retrieve it, but see Box 3.1 for alternatives.)

The result of running Listing 3.1 is `sonnets.txt`, a file containing all 154 of Shakespeare's sonnets. This file contains 2620 lines, far too many to fit on one screen. Learning how to inspect its contents is the goal of the rest of this section. (Among other things, we'll learn how to determine that it has 2620 lines without counting them all by hand.)

Box 3.1. Repeating previous commands

[3] If for any reason using `curl` fails, you can always visit the URL in a browser and then use the **File > Save As** feature to save it to your local disk.

Repeating previous commands is a frequent task when using the command line. So far in this tutorial, we've used the up-arrow key to retrieve (and possibly edit) previous commands, but this isn't the only possibility. An even quicker way to find and immediately run a previous command involves using the exclamation point `!`, which in the context of software development is usually pronounced "bang". To run the previous command exactly as written, we can use "bang bang":

```
$ echo "foo"
foo
$ !!
echo "foo"
foo
```

A closely related usage is "bang" followed by some number of characters, which runs the last command that started with those characters. For example, to run the last `curl` command, we could type this:

```
$ !curl
```

This would save us the trouble of typing out the options, the URL, etc. Depending on our history of commands, the even terser `!cu` or `!c` would work as well. This technique is especially useful when the desired command last happened many commands ago, which can make hitting up arrow cumbersome.

A second and incredibly powerful technique is `^R`, which lets you search interactively through your previous commands, and then optionally edit the result before executing. For example, we could try this to bring up the last `curl` command:

```
$ ^R
(reverse-i-search)`': curl
```

On most systems, hitting return would then put the last `curl` command after our prompt and allow us to edit it (if desired) before hitting return to execute it. When your workflow happens to involve repeatedly running a variety of similar commands, sometimes it can seem like "all commands start with `^R`."

3.1.1 Exercises

1. Use the command **`curl -I https://www.learnenough.com/`** to fetch the *HTTP header* for the Learn Enough website. What is the HTTP status code for the address? How does this differ from the status code for `learnenough.com` (without the **`https://`**)?

2. Using **`ls`**, confirm that **`sonnets.txt`** exists on your system. How big is it in bytes? *Hint*: Recall from Section 2.2 that the "long form" of **`ls`** displays a byte count.

3. The byte count in the previous exercise is high enough that it's more naturally thought of in *kilobytes* (often treated as 1000 bytes, but actually equal to $2^{10} = 1024$ bytes). By adding the **`-h`** ("human-readable") option to **`ls`**, list the long form of the sonnets file with a human-readable byte count.

4. Suppose you wanted to list the files and directories using **h**uman-readable byte counts, **a**ll, by **r**everse **t**ime-sorted **l**ong-form. What command would you use? Why might this command be a personal favorite of the author of this tutorial?[4]

3.2 Making heads and tails of it

Two complementary commands for inspecting files are **`head`** and **`tail`**, which respectively allow us to view the beginning (head) and end (tail) of the file. The **`head`** command shows the first 10 lines of the file (Listing 3.2).

Listing 3.2: Looking at the head of the sample text file.

```
$ head sonnets.txt
Shake-speare's Sonnets
```

[4]Having known about **`ls -a`** and **`ls -rtl`** for a while—which together yield the suggestive command **`ls -artl`**—one day I decided to add an "h"' (for obvious reasons). This is actually how I accidentally discovered the useful **`-h`** option some years ago.

```
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
```

Similarly, `tail` shows the last 10 lines of the file (Listing 3.3).

Listing 3.3: Looking at the tail of the sample text file.

```
$ tail sonnets.txt
The fairest votary took up that fire
Which many legions of true hearts had warm'd;
And so the general of hot desire
Was, sleeping, by a virgin hand disarm'd.
This brand she quenched in a cool well by,
Which from Love's fire took heat perpetual,
Growing a bath and healthful remedy,
For men diseas'd; but I, my mistress' thrall,
 Came there for cure and this by that I prove,
 Love's fire heats water, water cools not love.
```

These two commands are useful when (as is often the case) you know for sure you only need to inspect the beginning or end of a file.

3.2.1 Wordcount and pipes

By the way, I didn't recall offhand how many lines `head` and `tail` show by default. Since there are only 10 lines in the output, I could have counted them by hand, but in fact I was able to figure it out using the `wc` command (short for "wordcount"; recall Figure 2.4).

The most common use of `wc` is on full files. For example, we can run `sonnets.txt` through `wc`:

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

Here the three numbers indicate how many lines, words, and bytes there are in the file, so there are 2620 lines (thereby fulfilling the promise made at the end of Section 3.1), 17670 words, and 95635 bytes.

You are now in a position to be able to guess one method for determining how many lines are in **`head sonnets.txt`**. In particular, we can combine **`head`** with the redirect operator (Section 2.1) to make a file with the relevant contents, and then run **`wc`** on it, as shown in Listing 3.4.

Listing 3.4: Redirecting **`head`** and running **`wc`** on the result.

```
$ head sonnets.txt > sonnets_head.txt
$ wc sonnets_head.txt
    10     46    294 sonnets_head.txt
```

We see from Listing 3.4 that there are 10 lines in **`head wc`** (along with 46 words and 294 bytes). The same method, of course, would work for **`tail`**.

On the other hand, you might get the feeling that it's a little unclean to make an intermediate file just to run **`wc`** on it, and indeed there's a way to avoid it using a technique called *pipes*. Listing 3.5 shows how to do it.

Listing 3.5: Piping the result of **`head`** through **`wc`**.

```
$ head sonnets.txt | wc
    10     46    294
```

The command in Listing 3.5 runs **`head sonnets.txt`** and then *pipes* the result through **`wc`** using the pipe symbol **`|`** (Shift-backslash on most QWERTY keyboards). The reason this works is that the **`wc`** command, in addition to taking a filename as an argument, can (like many Unix programs) take input from "standard in" (compare to "standard out" mentioned in Section 1.3), which in this case is the output of **`head sonnets.txt`** shown in Listing 3.2. The **`wc`** program takes this input and counts it the same way it counts a file, yielding the same line, word, and byte counts as Listing 3.4.

3.2.2 Exercises

1. By piping the results of `tail sonnets.txt` through `wc`, confirm that (like `head`) the `tail` command outputs 10 lines by default.

2. By running `man head`, learn how to look at the first `n` lines of the file. By experimenting with different values of `n`, find a `head` command to print out just enough lines to display the first sonnet in its entirety (Figure 1.11).

3. Pipe the results of the previous exercise through `tail` (with the appropriate options) to print out *only* the 14 lines composing Sonnet 1. *Hint*: The command will look something like `head -n <i> sonnets.txt | tail -n <j>`, where `<i>` and `<j>` represent the numerical arguments to the `-n` option.

4. One of the most useful applications of `tail` is running `tail -f` to view a file that's actively changing. This is especially common when monitoring files used to log the activity of, e.g., web servers, a practice known as "tailing the log file". To simulate the creation of a log file, run `ping learnenough.com > learnenough.log` in one terminal tab. (The `ping` command "pings" a server to see if it's working.) In a second tab, type the command to tail the log file. (At this point, both tabs will be stuck, so once you've gotten the gist of `tail -f` you should use the technique from Box 1.4 to get out of trouble.)

3.3 Less is more

Unix provides two utilities for the common task of wanting to look at more than just the head or tail of a file. The older of these programs is called `more`, but (I'd guess initially as a tongue-in-cheek joke) there's a more powerful variant called `less`.[5] The `less` program is interactive, so it's hard to capture in print, but here's roughly what it looks like:

[5]On some systems, apparently they're exactly the same program, so `less` really is `more` (or, more accurately, `more` is `less`).

```
$ less sonnets.txt
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
```

The point of **`less`** is that it lets you navigate through the file in several useful ways, such as moving one line up or down with the arrow keys, pressing space bar to move a page down, pressing **`^F`** to move forward a page (i.e., the same as spacebar) or **`^B`** to move back a page. To quit **`less`**, type **`q`** (for “quit”).

Perhaps the most powerful aspect of **`less`** is the forward slash key **`/`**, which lets you search through the file from beginning to end. For example, suppose we wanted to search through **`sonnets.txt`** for “rose” (Figure 3.1),[6] one of the most frequently used images in the *Sonnets*.[7] The way to do this in **`less`** is to type **`/rose`** (read “slash rose”), as shown in Listing 3.6.

[6]Image retrieved from https://commons.wikimedia.org/wiki/File:Tudor_Rose.svg on 2015-07-21. Copyright © 2011 by Sodacan and used unaltered under the Creative Commons Attribution-ShareAlike 2.0 Generic license.

[7]Although Shakespeare’s sonnets are undated, most of them were probably composed during the reign of Queen Elizabeth, whose royal house adopted a rose (Figure 3.1) as its heraldic emblem. Given this context, Shakespeare’s choice of floral imagery isn’t surprising, but in fact only a few commentators on the *Sonnets* have noticed the seemingly obvious reference.

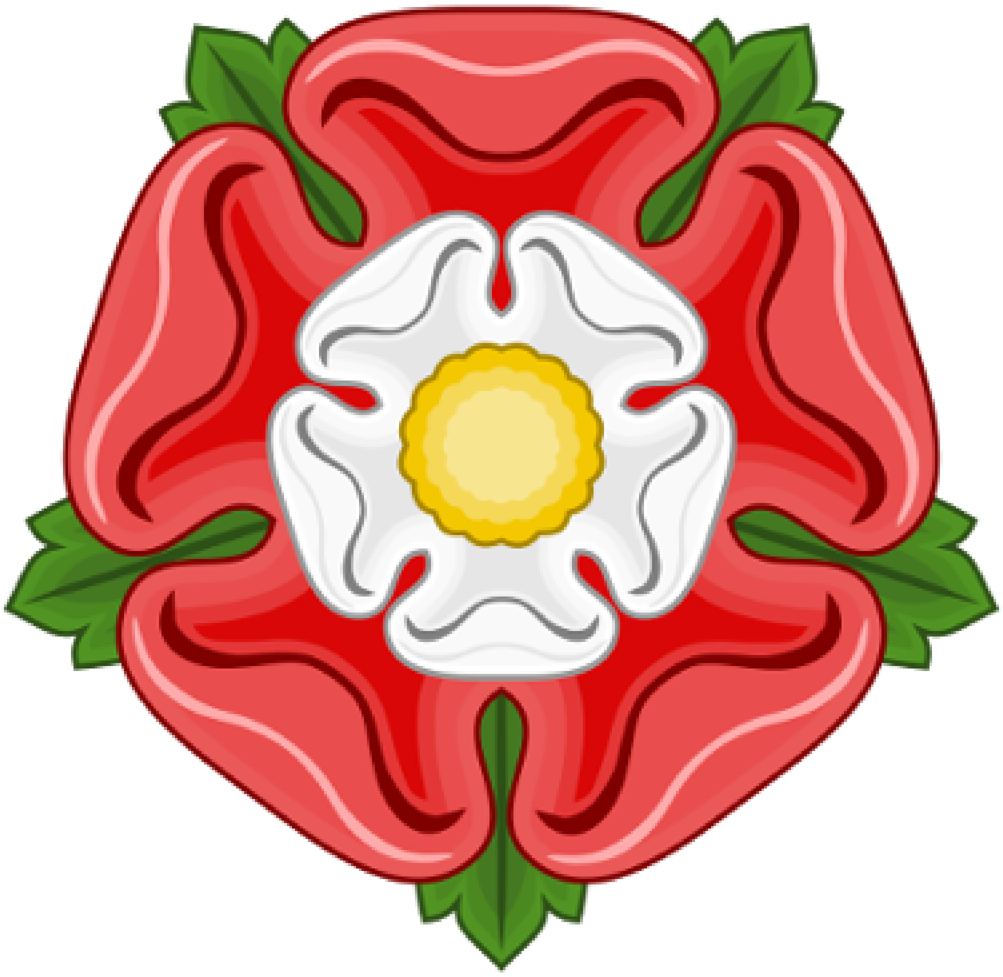


Figure 3.1: A famous rose from the time of Shakespeare.

Listing 3.6: Searching for the string “rose” using `less`.

```
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,
/rose
```

The result of pressing return after typing `/rose` in Listing 3.6 is to highlight the first occurrence of “rose” in the file. You can then press `n` to navigate to the next match, or `N` to navigate to the previous match.

The last two essential `less` less commands are `G` to move to the end of the file and `1G` (that’s `1` followed by `G`) to move back to the beginning. Table 3.1 summarizes what are in my view the most important key combinations (i.e., the ones I think you need to be *dangerous*), but if you’re curious you can find a longer list of commands at the Wikipedia page on `less`.

I encourage you to get in the habit of using `less` as your go-to utility for looking at the contents of a file. The skills you develop have other applications as well; for example, the man pages (Section 1.4) use the same interface as `less`, so by learning about `less` you’ll get better at navigating the man pages as well.

Command	Description	Example
`up & down arrow keys`	Move up or down one line	
`spacebar`	Move forward one page	
`^F`	Move forward one page	
`^B`	Move back one page	
`G`	Move to end of file	
`1G`	Move to beginning of file	
`/<string>`	Search file for string	**`/rose`**
`n`	Move to next search result	
`N`	Move to previous search result	
`q`	Quit **`less`**	

Table 3.1: The most important **`less`** commands.

3.3.1 Exercises

1. Run **`less`** on **`sonnets.txt`**. Go down three pages and then back up three pages. Go to the end of the file, then to the beginning, then quit.

2. Search for the string “All” (case-sensitive). Go forward a few occurrences, then back a few occurrences. Then go to the beginning of the file and count the occurrences by searching forward until you hit the end. Compare your count to the result of running **`grep All sonnets.txt | wc`**. (We’ll learn about **`grep`** in Section 3.4.)

3. Using **`less`** and **`/`** (“slash”), find the sonnet that begins with the line “Let me not”. Are there any other occurrences of this string in the *Sonnets*? *Hint*: Press **`n`** to find the next occurrence (if any). *Extra credit*: Listen to the sonnet in both modern and original pronunciation. Which version’s rhyme scheme is better?

4. Because **`man`** uses **`less`**, we are now in a position to search man pages interactively. By searching for the string “sort” in the man page for **`ls`**, discover the option to sort files by size. What is the command to display the long form of files sorted so the largest files appear at the bottom? *Hint*: Use **`ls -rtl`** as a model.

3.4 Grepping

One of the most powerful tools for inspecting file contents is **`grep`**, which probably stands for something, but it's not important what. (We'll actually mention it in a moment.) Indeed, *grep* is frequently used as a verb, as in "You should totally grep that file."

The most common use of **`grep`** is just to search for a substring in a file. For example, we saw in Section 3.3 how to use **`less`** to search for the string "rose" in Shakespeare's sonnets. Using **`grep`**, we can find the references directly, as shown in Listing 3.7.

Listing 3.7: Finding the occurrences of "rose" in Shakespeare's sonnets.

```
$ grep rose sonnets.txt
The rose looks fair, but fairer we it deem
As the perfumed tincture of the roses.
Die to themselves. Sweet roses do not so;
Roses of shadow, since his rose is true?
Which, like a canker in the fragrant rose,
Nor praise the deep vermilion in the rose;
The roses fearfully on thorns did stand,
 Save thou, my rose, in it thou art my all.
I have seen roses damask'd, red and white,
But no such roses see I in her cheeks;
```

With the command in Listing 3.7, it appears that we are in a position to count the number of lines containing references to the word "rose" by piping to **`wc`** (as in Section 3.3), as shown in Listing 3.8.

Listing 3.8: Piping the results of **`grep`** to **`wc`**.

```
$ grep rose sonnets.txt | wc
      10      82     419
```

Listing 3.8 tells us that 10 lines contain "rose" (or "roses", since "rose" is a substring of "roses"). But you may recall from Figure 1.11 that Shakespeare's first sonnet contains "Rose" with a *capital* "R". Referring to Listing 3.7, we see that this line has in fact been missed. This is because **`grep`** is case-sensitive by default, and "rose" doesn't match "Rose".

As you might suspect, **`grep`** has an option to perform case-insensitive matching as well. One way to figure it out is to search through the **`man`** page for **`grep`**:

- Type **`man grep`**
- Type **`/case`** and then return
- Read off the result (Figure 3.2)

(As noted briefly in Section 1.4, the man pages use the same interface as the **`less`** command we met in Section 3.3, so we can search through them using **`/`**.)

Applying the result of the above procedure yields Listing 3.9. Comparing the results of Listing 3.9 with Listing 3.8, we see that we now have 12 matching lines instead of only 10, so there must be a total of $12 - 10 = 2$ lines containing "Rose" (but not "rose") in the *Sonnets*.[8]

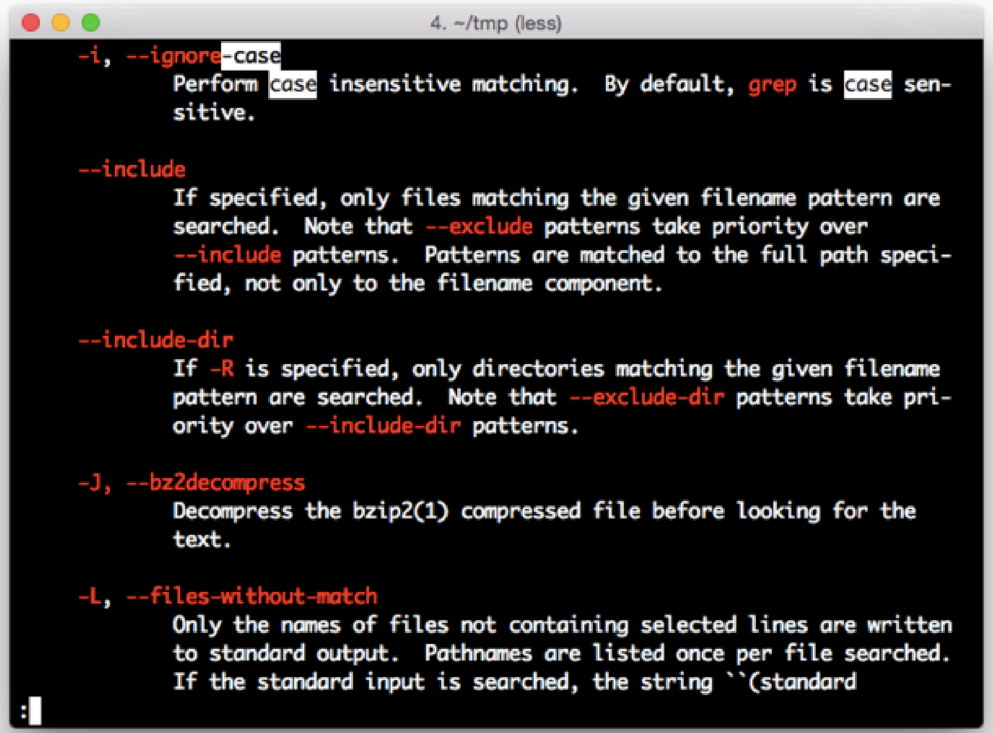
Listing 3.9: Doing a case-insensitive grep.

```
$ grep -i rose sonnets.txt | wc
   12    96    508
```

The **`grep`** utility gets its name from a pattern-matching system called *regular expressions* (also called *regexes* for short): *grep* stands for "**g**lobally search a **r**egular **e**xpression and **p**rint." A full treatment of regular expressions is well beyond the scope of this tutorial, but before moving on we'll sample just a small taste.

As one simple example, let's match every line in **`sonnets.txt`** that has a word beginning with the letters "ro", followed by any number of (lower-case) letters, and ending in "s". The way to represent "any letter" with a regular expression is **`[a-z]`**, and following a pattern with an asterisk **`*`** matches "zero or more" of that thing. Thus, **`ro[a-z]*s`** matches "ro" and "s" with zero or more letters in between. We can add spaces to the beginning and end to ensure that the match consists of entire words, like this:

[8] Actually, "ROSE", "RoSE", "rOSE", etc., all match as well, but "Rose" is the likeliest candidate. Confirming this hunch is left as an exercise (Section 3.4.1).



```
     -i, --ignore-case
             Perform case insensitive matching.  By default, grep is case sen-
             sitive.

     --include
             If specified, only files matching the given filename pattern are
             searched.  Note that --exclude patterns take priority over
             --include patterns.  Patterns are matched to the full path speci-
             fied, not only to the filename component.

     --include-dir
             If -R is specified, only directories matching the given filename
             pattern are searched.  Note that --exclude-dir patterns take pri-
             ority over --include-dir patterns.

     -J, --bz2decompress
             Decompress the bzip2(1) compressed file before looking for the
             text.

     -L, --files-without-match
             Only the names of files not containing selected lines are written
             to standard output.  Pathnames are listed once per file searched.
             If the standard input is searched, the string ``(standard
:
```

Figure 3.2: The result of searching **`man grep`** for “case”.

```
$ grep ' ro[a-z]*s ' sonnets.txt
  To that sweet thief which sourly robs from me.
Die to themselves. Sweet roses do not so;
When rocks impregnable are not so stout,
He robs thee of, and pays it thee again.
The roses fearfully on thorns did stand,
I have seen roses damask'd, red and white,
But no such roses see I in her cheeks;
```

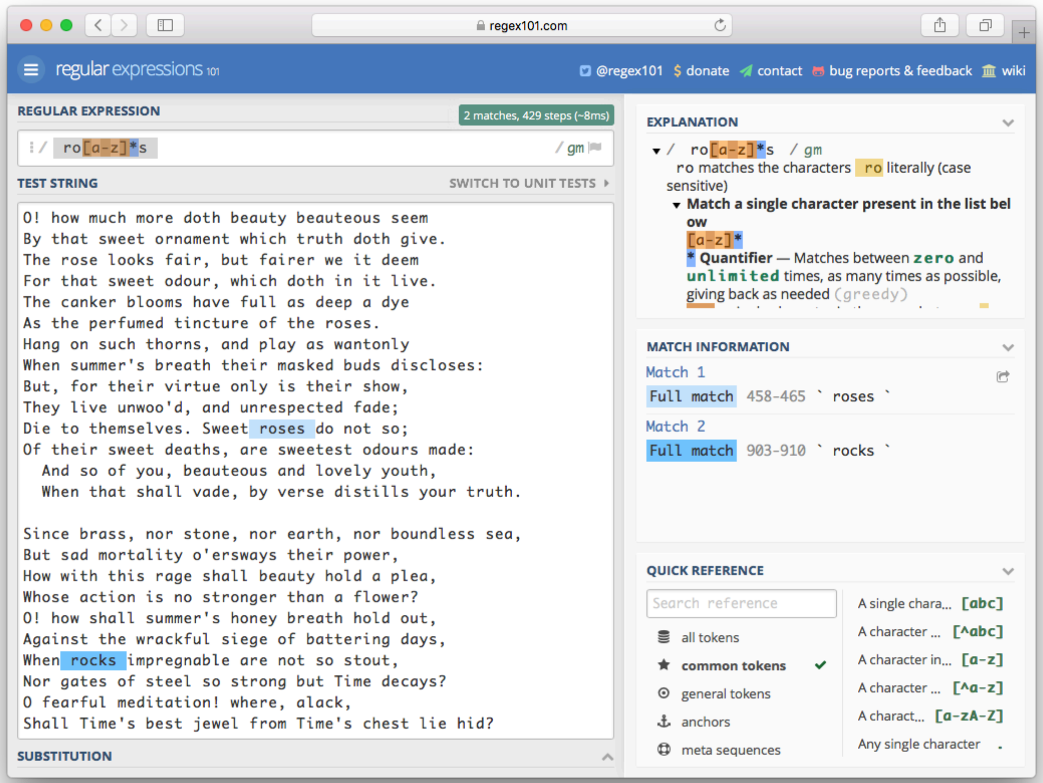
We can see that the regular expression matches strings like “robs” and “rocks” in addition to “roses”.

In general, one of the best tools for learning how to use regexes is an *online regex builder*, such as regex101, which lets you build up regexes interactively (Figure 3.3). Unfortunately, **grep** often doesn’t support the precise format used by regex builders (including hard-to-guess requirements for “escaping out” special characters), and precision in regular expressions is everything. As a result, despite its name origins, the truth is I rarely use the regular expression capabilities of **grep**. By the time the situation calls for regexes, I’m far likelier to reach for a text editor (*Learn Enough Text Editor to Be Dangerous*) or a full-strength programming language (*Learn Enough JavaScript to Be Dangerous*, *Learn Enough Ruby to Be Dangerous*).

Nevertheless, the aspects of **grep** discussed in this section are nearly enough to be *dangerous*, covering a huge number of common cases (including the important application of *grepping processes* (Box 3.2)). We’ll see one final **grep** variant in Chapter 4 as part of our discussion of Unix directories.

Box 3.2. Grepping processes

One of the many uses of `grep` is filtering the Unix *process list* for running programs that match a particular string. (On Unix-like systems such as Linux and macOS, user and system tasks each take place within a well-defined container called a *process*.) This is especially useful when there’s a rogue process on your system that needs to be killed. (A good way to find such processes is by running the `top` command, which shows the processes consuming the most resources.)



```
/ ro[a-z]*s / gm
```

```
O! how much more doth beauty beauteous seem
By that sweet ornament which truth doth give.
The rose looks fair, but fairer we it deem
For that sweet odour, which doth in it live.
The canker blooms have full as deep a dye
As the perfumed tincture of the roses.
Hang on such thorns, and play as wantonly
When summer's breath their masked buds discloses:
But, for their virtue only is their show,
They live unwoo'd, and unrespected fade;
Die to themselves. Sweet roses do not so;
Of their sweet deaths, are sweetest odours made:
  And so of you, beauteous and lovely youth,
  When that shall vade, by verse distills your truth.

Since brass, nor stone, nor earth, nor boundless sea,
But sad mortality o'ersways their power,
How with this rage shall beauty hold a plea,
Whose action is no stronger than a flower?
O! how shall summer's honey breath hold out,
Against the wrackful siege of battering days,
When rocks impregnable are not so stout,
Nor gates of steel so strong but Time decays?
O fearful meditation! where, alack,
Shall Time's best jewel from Time's chest lie hid?
```

/ ro[a-z]*s / gm
ro matches the characters ro literally (case sensitive)
Match a single character present in the list below
[a-z]*
*** Quantifier** — Matches between **zero** and **unlimited** times, as many times as possible, giving back as needed (greedy)

```
Match 1
Full match 458-465 ` roses `
Match 2
Full match 903-910 ` rocks `
```

A single chara...	[abc]
A character ...	[^abc]
A character in...	[a-z]
A character ...	[^a-z]
A charact...	[a-zA-Z]
Any single character	.

Figure 3.3: An online regex builder.

For example, at one point in the *Ruby on Rails Tutorial* book, it's important to eliminate a program called `spring` from the process list. To do this, first the processes need to be found, and the way to see all the processes on your system is to use the `ps` command with the `aux` options:

```
$ ps aux
```

Per Figure 2.4, `ps` is short for "process status". And for confusing and obscure reasons, options to `ps` aren't written with a dash (so it's `ps aux` instead of `ps -aux`). (How on earth are you supposed to know this? That's what this tutorial is for.)

To filter the processes by program name, you pipe the results of `ps` through `grep`:

```
$ ps aux | grep spring
ubuntu 12241 0.3 0.5 589960 178416 ? Ssl Sep20 1:46
spring app | sample_app | started 7 hours ago
```

The result shown gives some details about the process, but the most important thing is the first number, which is the *process id*, or pid (often pronounced to rhyme with "kid"). To eliminate an unwanted process, we use the `kill` command to issue the Unix terminate code (which happens to be 15) to the pid:

```
$ kill -15 12241
```

This is the technique I recommend for killing individual processes, such as a rogue web server (with the pid found via `ps aux | grep server`), but sometimes it's convenient to kill all the processes matching a particular process name, such as when you want to kill all the `spring` processes gunking up your system. In this case, you can kill all the processes with name `spring` using the `pkill` command as follows:

```
$ pkill -15 -f spring
```

Any time something isn't behaving as expected, or a process appears to be frozen, it's a good idea to run `top` or `ps aux` to see what's going on, pipe `ps aux` through `grep` to select the suspected processes, and then run `kill -15 <pid>` or `pkill -15 -f <name>` to clear things up.

3.4.1 Exercises

1. By searching **`man grep`** for "line number", construct a command to find the line numbers in **`sonnets.txt`** where the string "rose" appears.

2. You should find that the last occurrences of "rose" is (via "roses") on line 2203. Figure out how to go directly to this line when running **`less sonnets.txt`**. *Hint*: Recall from Table 3.1 that **`1G`** goes to the top of the file, i.e., line 1. Similarly, **`17G`** goes to line 17. Etc.

3. By piping the output of **`grep`** to **`head`**, print out the first (and *only* the first) line in **`sonnets.txt`** containing "rose". *Hint*: Use the result of the second exercise in Section 3.2.2.

4. In Listing 3.9, we saw two additional lines that case-insensitively matched "rose". Execute a command confirming that both of the lines contain the string "Rose" (and not, e.g., "rOSe"). *Hint*: Use a case-sensitive **`grep`** for "Rose".

5. You should find in the previous exercise that there are *three* lines matching "Rose" instead of the two you might have expected from Listing 3.9. This is because there is one line that contains both "Rose" *and* "rose", and thus shows up in both **`grep rose`** and **`grep -i rose`**. Write a command confirming that the number of lines matching "Rose" but *not* matching "rose" is equal to the expected 2. *Hint*: Pipe the result of **`grep`** to **`grep -v`**, and then pipe that result to **`wc`**. (What does **`-v`** do? Read the man page for **`grep`** (Box 1.5).)

Command	Description	Example
`curl`	Interact with URLs	`$ curl -O https://example.`
`which`	Locate a program on the path	`$ which curl`
`head <file>`	Display first part of file	`$ head foo`
`tail <file>`	Display last part of file	`$ tail bar`
`wc <file>`	Count lines, words, bytes	`$ wc foo`
`cmd1 | cmd2`	Pipe `cmd1` to `cmd2`	`$ head foo | wc`
`ping <url>`	Ping a server URL	`$ ping google.com`
`less <file>`	View file contents interactively	`$ less foo`
`grep <string> <file>`	Find string in file	`$ grep foo bar.txt`
`grep -i <string> <file>`	Find case-insensitively	`$ grep -i foo bar.txt`
`ps`	Show processes	`$ ps aux`
`top`	Show processes (sorted)	`$ top`
`kill -<level> <pid>`	Kill a process	`$ kill -15 24601`
`pkill -<level> -f <name>`	Kill matching processes	`$ pkill -15 -f spring`

Table 3.2: Important commands from Chapter 3.

3.5 Summary

Important commands from this section are summarized in Table 3.2.

3.5.1 Exercises

1. The **`history`** command prints the history of commands in a particular terminal shell (subject to some limit, which is typically large). Pipe **`history`** to **`less`** to examine your command history. What was your 17th command?

2. By piping the output of **`history`** to **`wc`**, count how many commands you've executed so far.

3. One use of **`history`** is to grep your commands to find useful ones you've used before, with each command preceded by the corresponding number in the command history. By piping the output of **`history`** to **`grep`**, determine the number for the last occurrence of **`curl`**.

4. In Box 3.1, we learned about `!!` ("bang bang") to execute the previous command. Similarly, `!n` executes command number `n`, so that, e.g., `!17` executes the 17th command in the command history. Use the result from the previous exercise to re-run the last occurrence of `curl`.

5. What do the `O` and `L` options in Listing 3.1 mean? *Hint*: Pipe the output of `curl -h` to `less` and search first for the string `-O` and then for the string `-L`.

Chapter 4

Directories

Having examined many of the Unix utilities for dealing with files, the time has come to learn about *directories*, sometimes known by the synonym *folders* (Figure 4.1). As we'll see, many of the ideas developed in the context of files also apply to directories, but there are many differences as well.

4.1 Directory structure

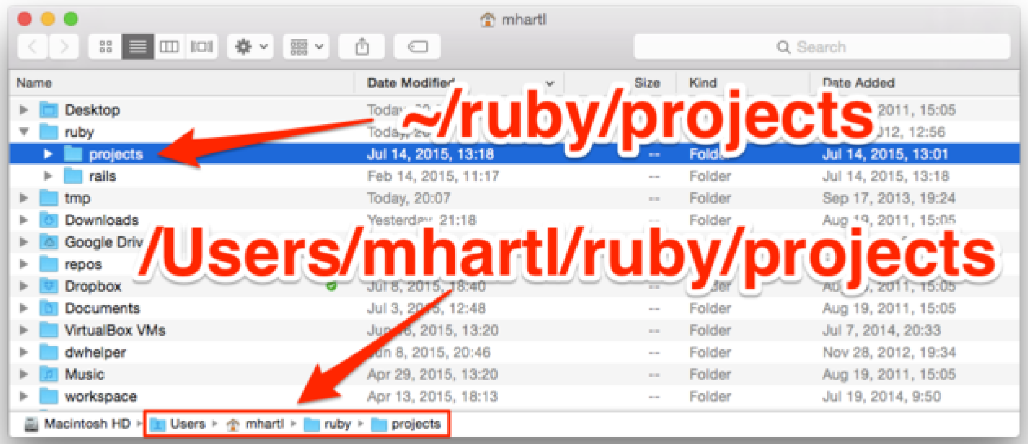
The structure of Unix-style directories is typically indicated using a list of directory names separated by forward slashes, which we can combine with the **`ls`** command (Section 2.2) like this:

```
$ ls /Users/mhartl/ruby
```

or like this:

```
$ ls /usr/local/bin
```

As seen in Figure 4.1, these representations correspond to directories in a hierarchical filesystem, with (say) **`mhartl`** a subdirectory of **`Users`** and **`ruby`** a subdirectory of **`mhartl`**.



Name	Date Modified	Size	Kind	Date Added
Desktop	[illegible]	[illegible]	[illegible]	[illegible] 2011, 15:05
ruby	[illegible]	[illegible]	[illegible]	[illegible] 12, 12:56
projects	Jul 14, 2015, 13:18	--	[illegible]	Jul 14, 2015, 13:01
rails	Feb 14, 2015, 11:17	--	Folder	Jul 14, 2015, 13:18
tmp	Today, 20:07	--	Folder	Sep 17, 2013, 19:24
Downloads	[illegible] 21:18	[illegible]	Folder	[illegible] 2011, 15:05
Google Driv	[illegible]	[illegible]	[illegible]	[illegible]
repos	[illegible]	[illegible]	[illegible]	[illegible]
Dropbox	Jul 8, 2015, 18:40	--	[illegible]	[illegible] 2011, 15:05
Documents	Jul 3, [illegible], 12:48	--	Folder	Aug 19, 2011, 15:05
VirtualBox VMs	[illegible], 2015, 13:20	--	Folder	Jul 7, 2014, 20:33
dwhelper	[illegible] 8, 2015, 20:46	--	Folder	Nov 28, 2012, 19:34
Music	Apr 29, 2015, 13:20	--	Folder	Aug 19, 2011, 15:05
workspace	Apr 13, 2015, 18:13	--	Folder	Jul 19, 2014, 9:50

Macintosh HD ▸ Users ▸ mhartl ▸ ruby ▸ projects

Figure 4.1: The correspondence between folders & directories.

Conventions vary when speaking about directories: a *user directory* like **`/Users/mhartl`** would probably be read as "slash users slash mhartl" or "slash users mhartl", whereas omitting the initial slash in spoken language is common for *system directories* such as **`/usr/local/bin`**, which would probably be pronounced "user local bin".[1] Because *all* Unix directories are ultimately subdirectories of the *root directory* **`/`** (pronounced "slash"), the leading and separator slashes are implied. *Note*: Referring to forward slashes incorrectly as "backslashes" is a source of intense suffering, and should be strictly avoided.

The most important directory for a particular user is the *home directory*, which on my macOS system is **`/Users/mhartl`**, corresponding to my username (**`mhartl`**). The home directory can be specified as an *absolute path*, as in **`/Users/mhartl`**, or using the shorthand for the home directory is the tilde character **`~`** (which is typed using Shift-backtick, located just to the left of the number 1 on most keyboards). As a result, the two paths shown in Figure 4.1 are identical: **`/Users/mhartl/ruby/projects`** is the same as

[1]For more on Unix system directories, see "What is /usr/local/bin?" at the Unix StackExchange. Thanks to reader Joost van der Linden for the suggestion.

`~/ruby/projects`. (Amusingly, the reason the tilde character is used for the home directory is simply because the "Home" key was the same as the key for producing "~" on some early keyboards.)

In addition to user directories, every Unix system has *system directories* for the programs essential for the computer's normal operation. Modifying system files or directories requires special powers granted only to the *superuser*, known as **`root`**. (This use of "root" is unrelated to the "root directory" mentioned above.) The superuser is so powerful that it's considered bad form to log in as **`root`**; instead, tasks executed as **`root`** should typically use the **`sudo`** command (Box 4.1).

Box 4.1. "`sudo` make me a sandwich."

`sudo` gives ordinary users the power to execute commands as the superuser. For example, let's try `touch`ing a file in the system directory `/opt` as follows:

```
$ touch /opt/foo
touch: /opt/foo: Permission denied
```

Because normal users don't have permission to modify `/opt`, the command fails, but it succeeds with `sudo`:

```
$ sudo touch /opt/foo
Password:
```

As shown, after entering `sudo` we are prompted to enter our user password; if entered correctly, and if the user has been configured to have `sudo` privileges (which is the default on most desktop Unix systems), then the command will succeed. As shown in the xkcd comic strip "Sandwich", this pattern of being denied at first, only to succeed using `sudo`, is a common pattern when using the command line.

To check that the file really was created, we can `ls` it:

```
$ ls -l /opt/foo
-rw-r--r-- 1 root wheel 0 Jul 23 19:13 /opt/foo
```

Note that (1) a normal user can `ls` a file in a system directory (without `sudo`) and (2) the name `root` appears in the listing, indicating that the superuser owns the file. (The meaning of the second term, `wheel`, is a little obscure, but you can learn about it on a site called, appropriately enough, superuser.)

To remove the file we just created, we again need superuser status:

```
$ rm -f /opt/foo
rm: /opt/foo: Permission denied
$ sudo !!
$ !ls
ls: /opt/foo: No such file or directory
```

Here the first `rm` fails, so we've run `sudo !!`, which runs `sudo` and then the previous command , and we've followed that up with `!ls`, which runs the previous `ls` command (Box 3.1).

It's also worth noting the English pronunciation of something like `sudo !!`, which is important when communicating via spoken language. As noted in Box 3.1, `!!` is pronounced "bang bang". `sudo`, meanwhile, is pronounced either "SOO-doo" or "SOO-doh". Both pronunciations are common, though I prefer the former because the `do` in `sudo` is in fact just the English word "do". Thus, my preferred pronunciation for `sudo !!` is "SOO-doo bang bang".

By the way, the `su` in `sudo` originally stood for "super-user", but over time its use expanded, and now is usually thought of as "substitute user". `sudo` is therefore a contraction of "substitute user do", with the substitute user being the superuser by default. Because the superuser can do anything, the command "sudo make me a sandwich" in "Sandwich" succeeds when a mere "make me a sandwich" does not.

4.1.1 Exercises

1. Write in words how you might speak the directory **`~/foo/bar`**.

2. In **`/Users/bill/sonnets`**, what is the home directory? What is the

username? Which directory is deepest in the hierarchy?

3. For a user with username **bill**, how do **/Users/bill/sonnets** and **~/sonnets** differ (if at all)?

4.2 Making directories

So far in this tutorial, we've created (and removed) a large number of text files. The time has finally come to make a directory to contain them. Although most modern operating systems include a graphical interface for this task, the Unix way to do it is with **mkdir** (short for "make directory", per Figure 2.4):

```
$ mkdir text_files
```

Having made the directory, we can move the text files there using a wildcard:

```
$ mv *.txt text_files/
```

We can confirm the move worked by listing the directory:

```
$ ls text_files/
sonnet_1.txt     sonnet_1_reversed.txt sonnets.txt
```

(Depending on how closely you've followed this tutorial, your results may vary.)

By default, running **ls** on a directory shows its *contents*, but we can show just the directory using the **-d** option:

```
$ ls -d text_files/
text_files/
```

This usage is especially common with the **-l** option (Section 2.2):

```
$ ls -ld text_files/
drwxr-xr-x 7 mhartl staff 238 Jul 24 18:07 text_files
```

Finally, we can change directories using **`cd`**:

```
$ cd text_files/
```

Note that **`cd`** typically supports tab completion, so (as described in Box 2.4) we can actually type **`cd tex`**⇥.

After running **`cd`**, we can confirm that we're in the correct directory using the "print working directory" command, **`pwd`**, together with another call to **`ls`**:

```
$ pwd
/Users/mhartl/text_files
$ ls
sonnet_1.txt     sonnet_1_reversed.txt sonnets.txt
```

These last steps of typing **`pwd`** to double-check the directory, and especially running **`ls`** to inspect the directory contents, are a matter of habit for many a grizzled command-line veteran. (Your result for **`pwd`** will, of course, be different, unless you happen to be using the username "mhartl" on macOS.)

4.2.1 Exercises

1. What is the option for making intermediate directories as required, so that you can create, e.g., **`~/foo`** and **`~/foo/bar`** with a single command? *Hint*: Refer to the man page for **`mkdir`**.

2. Use the option from the previous exercise to make the directory **`foo`** and, within it, the directory **`bar`** (i.e., **`~/foo/bar`**) with a single command.

3. By piping the output of **`ls`** to **`grep`**, list everything in the home directory that contains the letter "o".

4.3 Navigating directories

We saw in Section 4.2 how to use **`cd`** to change to a directory with a given name. This is one of the most common ways of navigating, but there are a couple of special forms worth knowing. The first is changing to the directory one level up in the hierarchy using **`cd ..`** (read "see-dee dot-dot"):

```
$ pwd
/Users/mhartl/text_files
$ cd ..
$ pwd
/Users/mhartl
```

In this case, because **`/Users/mhartl`** is my *home directory*, we could have accomplished the same thing using **`cd`** by itself:

```
$ cd text_files/
$ pwd
/Users/mhartl/text_files
$ cd
$ pwd
/Users/mhartl
```

The reason this works is that **`cd`** by itself changes to the user's home directory, wherever that is. This means that

```
$ cd
```

and

```
$ cd ~
```

are equivalent.

When changing directories, it's frequently useful to be able to specify the home directory somehow. For example, suppose we make a second directory and cd into it:

```
$ pwd
/Users/mhartl
$ mkdir second_directory
$ cd second_directory/
```

Now if we want to change to the `text_files` directory, we can `cd` to `text_files` via the home directory `~`:

```
$ pwd
/Users/mhartl/second_directory
$ cd ~/text_files
$ pwd
/Users/mhartl/text_files
```

Incidentally, we're now in a position to understand the prompts shown in Figure 1.6: I have my prompt configured to show the current directory, which might be something like `[~]`, `[ruby]`, or `[projects]`. (We'll discuss how to customize the prompt in *Learn Enough Text Editor to Be Dangerous*. Especially eager readers can exercise their technical sophistication (Box 1.5) by Googling for how to do it.)

Closely related to `..` for "one directory up" is `.` (read "dot") which means "the current directory". The most common use of `.` is when moving or copying files to the current directory:

```
$ pwd
/Users/mhartl/text_files
$ cd ~/second_directory
$ ls
$ cp ~/text_files/sonnets.txt .
$ ls
sonnets.txt
```

Note in that the first call to `ls` returns nothing, because `second_directory` is initially empty.

Another common use of `.` is in conjunction with the `find` command, which like `grep` is incredibly powerful, but in my own use it looks like this 99% of the time:

```
$ cd
$ find . -name '*.txt'
./text_files/sonnet_1.txt
./text_files/sonnet_1_reversed.txt
./text_files/sonnets.txt
```

In words, what this does is find files whose names match the pattern `*.txt`, starting in the current directory `.` and then in its subdirectories.[2] The `find` utility is incredibly useful for finding a misplaced file at the command line.

Perhaps my favorite use of `.` is in "open dot", which will work only on macOS:

```
$ cd ~/ruby/projects
$ open .
```

The remarkable `open` command opens its argument using whatever the default program is for the given file or directory. (A similar command, `xdg-open`, works on some Linux systems.) For example, `open foo.pdf` would open the PDF file with the default viewer (which is Preview on most Macs). In the case of a directory such as `.`, that default program is the Finder, so `open .` produces a result like that shown in Figure 4.1.

A final navigational command, and one of my personal favorites, is `cd -`, which cds to the *previous* directory, wherever it was:

```
$ pwd
/Users/mhartl/second_directory
$ cd ~/text_files
$ pwd
/Users/mhartl/text_files
$ cd -
/Users/mhartl/second_directory
```

I find that `cd -` is especially useful when combining commands, as described in Box 4.2.

[2]My directory has a huge number of text files, 'cause that's just how I roll, so the command I ran was actually `find . -name '*.txt' | grep text_files`, which filters out anything that doesn't match the directory being used in this tutorial.

Box 4.2. Combining commands

It's often convenient to combine commands at the command line, such as when installing software using the Unix programs `configure` and `make`, which often appear in the following sequence:

```
$ ./configure ; make ; make install
```

This line runs the `configure` program from the current directory `.`, and then runs both `make` and `make install`. (You are not expected to understand what these programs do, and indeed they won't work on your system unless you happen to be in the directory of a program designed to be installed this way.) Because they are separated by the semicolon character `;`, three commands are run in sequence.

An even better way to combine commands is with the double-ampersand `&&`:

```
$ ./configure && make && make install
```

The difference is that commands separated by `&&` run only if the previous command *succeeded*. In contrast, with `;` all the commands will be run no matter what, which will cause an error in the likely case that subsequent commands depend on the results of the ones that precede them.

I especially like to use `&&` in combination with `cd -`, which lets me do things like this:

```
$ build_article && cd ~/tau && deploy && cd -
```

Again, you are not expected to understand these commands, but the general idea is that we can (say) build an article in one directory, `cd` to a different directory, deploy (perhaps a website) to production, and then `cd` back (`cd -`) to the original directory, where we can continue our work. Then, if need be, we can just use up arrow (or one of the techniques from Box 3.1) to retrieve the whole thing and do it all again.

4.3.1 Exercises

1. How do the effects of **`cd`** and **`cd ~`** differ (or do they)?
2. Change to **`text_files`**, then change to **`second_directory`** using the "one directory up" double dot operator **`..`**.
3. From wherever you are, create an empty file called **`nil`** in **`text_files`** using whatever method you wish.
4. Remove **`nil`** from the previous exercise using a different path from the one you used before. (In other words, if you used the path **`~/text_files`** before, use something like **`../text_files`** or **`/Users/<username>/text_files`**.)

4.4 Renaming, copying, and deleting directories

The commands for renaming, copying, and deleting directories are similar to those for files (Section 2.3), but there are some subtle differences worth noting. The command with the least difference is **`mv`**, which works just as it does for files:

```
$ mkdir foo
$ mv foo/ bar/
$ cd foo/
-bash: cd: foo: No such file or directory
$ cd bar/
```

Here the error message indicates that the **`mv`** worked: there is no file or directory called **`foo`**.

```
$ cd foo/
-bash: cd: foo: No such file or directory
```

(The word **`bash`** refers to the name of the particular shell program being run, which in this case is the "Bourne-again shell".) The only minor subtlety is that the trailing slashes (which are typically added automatically by tab completion (Box 2.4)) are optional:

```
$ cd
$ mv bar foo
$ cd foo/
```

This issue with trailing slashes never makes a difference with **`mv`**, but with **`cp`** it can be a source of much confusion. In particular, when copying directories, the behavior you usually want is to copy the directory contents *including* the directory, which on many systems requires leaving off the trailing slash. When copying files, you also need to include the **`-r`** option (for "recursive"). For example, to copy the contents of the **`text_files`** directory to a new directory called **`foobar`**, we use the command shown in Listing 4.1.

Listing 4.1: Copying a directory.

```
$ cd
$ mkdir foobar
$ cd foobar/
$ cp -r ../text_files .
$ ls
text_files
```

Note that we've used **`..`** to make a *relative path*, going up one directory and then into **`text_files`**. Also note the *lack* of a trailing slash in **`cp -r ../text_files .`**; if we included it, we'd get Listing 4.2 instead.

Listing 4.2: Copying with a trailing slash.

```
$ cp -r ../text_files/ .
$ ls
sonnet_1.txt     sonnet_1_reversed.txt sonnets.txt   text_files
```

In other words, Listing 4.2 copies the individual files, but not the directory itself. As a result, I recommend always omitting the trailing slash, as in Listing 4.1; if you want to copy only the files, be explicit using the star operator, as in:

```
$ cp ../text_files/* .
```

Unlike renaming (moving) and copying directories, which use the same **`mv`** and **`cp`** commands used for files, in the case of removing directories there's a dedicated command called **`rmdir`**. In my experience, though, it rarely works, as seen here:

```
$ cd
$ rmdir second_directory
rmdir: second_directory/: Directory not empty
```

The error message here is what happens 99% of the time when I try to remove directories, because **`rmdir`** requires the directory to be empty. You can of course empty it by hand (using **`rm`** repeatedly), but this is frequently inconvenient, and I almost always use the more powerful (but *much* more dangerous) "remove recursive force" command **`rm -rf`**, which removes a directory, its files, and any subdirectories without confirmation (Listing 4.3).

Listing 4.3: Using **`rm -rf`** to remove a directory.

```
$ rm -rf second_directory/
$ ls second_directory
ls: second_directory: No such file or directory
```

As the error message from **`ls`** in Listing 4.3 indicates ("No such file or directory"), our use of **`rm -rf`** succeeded in removing the directory.

The powerful command **`rm -rf`** is too convenient to ignore, but remember: "With great power comes great responsibility" (Figure 4.2).[3]

[3]Image retrieved from https://www.flickr.com/photos/whatleydude/13950241545 on 2015-07-22. Copyright © 2014 by James Whatley and used unaltered under the terms of the Creative Commons Attribution 2.0 Generic license.

Figure 4.2: This superhero understands how to use the power of **`rm -rf`** responsibly.

4.4.1 Grep redux

Now that we know a little about directories, we are in a position to add a useful **grep** variation to our toolkit from Section 3.4. As with **cp** and **rm**, **grep** takes a "recursive" option, **-r**, which in this case greps through a directory's files and the files in its subdirectories. This is incredibly useful when you're looking for a string in a file somewhere in a hierarchy of directories, but you're not sure where the file is. Here's the setup, which puts the word "sesquipedalian" in a file called **long_word.txt**:

```
$ cd text_files/
$ mkdir foo
$ cd foo/
$ echo sesquipedalian > long_word.txt
$ cd
```

The final **cd** puts us back in the home directory. Suppose we now want to find the file containing "sesquipedalian". The way *not* to do it is this:

```
$ grep sesquipedalian text_files    # This doesn't work.
grep: text_files: Is a directory
```

Here **grep**'s error message indicates that the command didn't work, but adding **-r** does the trick:

```
$ grep -r sesquipedalian text_files
text_files/foo/long_word.txt:sesquipedalian
```

Because we don't usually care about case when searching files, I recommend making a habit of adding the **-i** option when grepping recursively, as follows:

```
$ grep -ri sesquipedalian text_files
text_files/foo/long_word.txt:sesquipedalian
```

Armed with **grep -ri**, we are now equipped to find strings of our choice in arbitrarily deep hierarchies of directories.

Command	**Description**	**Example**
`mkdir <name>`	Make directory with name	`$ mkdir foo`
`pwd`	Print working directory	`$ pwd`
`cd <dir>`	Change to <dir>	`$ cd foo/`
`cd ~/<dir>`	cd relative to home	`$ cd ~/foo/`
`cd`	Change to home directory	`$ cd`
`cd -`	Change to previous directory	`$ cd && pwd && cd -`
`.`	The current directory	`$ cp ~/foo.txt .`
`..`	One directory up	`$ cd ..`
`find`	Find files & directories	`$ find . -name foo*.*`
`cp -r <old> <new>`	Copy recursively	`$ cp -r ~/foo .`
`rmdir <dir>`	Remove (empty) dir	`$ rmdir foo/`
`rm -rf <dir>`	Remove dir & contents	`$ rm -rf foo/`
`grep -ri <string> <dir>`	Grep recursively (case-insensitive)	`$ grep -ri foo bar/`

Table 4.1: Important commands from Chapter 4.

4.4.2 Exercises

1. Make a directory **`foo`** with a subdirectory **`bar`**, then rename the subdirectory to **`baz`**.

2. Copy all the files in **`text_files`**, *with* directory, into **`foo`**.

3. Copy all the files in **`text_files`**, *without* directory, into **`bar`**.

4. Remove **`foo`** and everything in it using a single command.

4.5 Summary

Important commands from this section are summarized in Table 4.1.

4.5.1 Exercises

1. Starting in your home directory, execute a single command-line command to make a directory **`foo`**, change into it, create a file **`bar`** with con-

tent “baz”, print out **`bar`**’s contents, and then **`cd`** back to the directory you came from. *Hint*: Combine the commands as described in Box 4.2.

2. What happens when you run the previous command again? How many of the commands executed? Why?

3. Explain why the command **`rm -rf /`** is unbelievably dangerous, and why you should never type it into a terminal window, not even as a joke.

4. How can the previous command be made even more dangerous? *Hint*: Refer to Box 4.1. (This command is so dangerous you shouldn’t even *think* it, much less type it.)

4.6 Conclusion

Congratulations! You’ve officially learned enough command line to be *dangerous*. Of course, this is only one step on a longer journey, both toward command-line excellence (Box 4.3) and software development wizardry. As you proceed on this journey, you will probably discover that learning computer magic can be exciting and empowering, but it can also be *hard*. Indeed, you may already have discovered this fact, either on your own or while doing this tutorial. To those brave magicians-in-training who wish to proceed, I offer the following sequence:

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

2. **Web Basics**
   (a) *Learn Enough HTML to Be Dangerous*
   (b) *Learn Enough CSS & Layout to Be Dangerous*

   (c) *Learn Enough JavaScript to Be Dangerous*

3. **Application Development**
   (a) *Learn Enough Ruby to Be Dangerous*
   (b) *The Ruby on Rails Tutorial*
   (c) *Learn Enough Action Cable to Be Dangerous* (optional)

Box 4.3. Additional resources

I recommend following the *Learn Enough* sequence described in the main text, as it represents the shortest path to technical proficiency and software development skills, but at some point you'll probably want to expand your command-line skills as well. When that time comes, I recommend consulting this list, which consists mainly of resources recommended by readers of the present tutorial:

- *Conquering the Command Line* book and screencasts by Mark Bates
- edX course on Linux
- Codecademy course on the command line
- *Learning the Shell*

If you enjoyed this tutorial, please leave a review at Amazon. Thanks!