The Unix Workbench

Sean Kross

The Unix Workbench

Sean Kross

This book is for sale at http://leanpub.com/unix

This version was published on 2017-07-20

This is a Leanpub book. Leanpub empowers authors and publishers with the Lean Publishing process. Lean Publishing is the act of publishing an in-progress ebook using lightweight tools and many iterations to get reader feedback, pivot until you have the right book and build traction once you do.

Tweet This Book!

Please help Sean Kross by spreading the word about this book on Twitter!

The suggested hashtag for this book is #unixwb.

Find out what other people are saying about the book by clicking on this link to search for this hashtag on Twitter:

https://twitter.com/search?q=#unixwb

In memory of Toby Kross.

Contents

Acknowledgements

Thank you to Jeff Leek, Roger Peng, and Brian Caffo for your advice and support. Also thank you to Jon Calder and Elissa Redmiles for your edits and suggestions.

Introduction

This book is intended for folks who are new to programming and new to Unix-like operating systems like macOS and Linux distributions like Ubuntu. Most of the technologies discussed in this book will be accessed via a command line interface. Command line interfaces can seem alien at first, so this book attempts to draw parallels between using the command line and actions that you would normally take while using your mouse and keyboard. You'll also learn how to write little pieces of software in a programming language called Bash, which allows you to connect together the tools we'll discuss. My hope is that by the end of this book you be able to use different Unix tools as if they're interconnecting Lego bricks.

Unix forms a foundation that is often very helpful for accomplishing other goals you might have for you and your computer, whether that goal is running a business, writing a book, curing disease, or creating the next great app. The means to these goals are sometimes carried out by writing software. Software can't be mined out of the ground, nor can software seeds be planted in spring to harvest by autumn. Software isn't produced in factories on an assembly line. Software is a hand-made, often bespoke good. If a software developer is an artisan, then Unix is their workbench. Unix provides an essential and simple set of tools in a distraction-free environment. Even if you're not a software developer learning Unix can open you up to new methods of thinking and novel ways to scale your ideas. My goal for this book is to help you get started with Unix by writing the book I would have wanted

when I was first learning Unix. If you have any additions, corrections, or comments for this book please open an issue or send a pull request to: https://github.com/seankross/the-unix-workbench. If you're unsure what a pull request is don't worry, you'll find out in the Git and GitHub chapter of this book!

What is Unix?

Unix is an operating system and a set of tools. The tool we'll be using the most in this book is a shell, which is a computer program that provides a command line interface. You've probably seen a command line interface in the movies: an elite computer hacker sits in front of a black screen with green glowing text, furiously typing in commands and shouting something like "Spike them!" Using the command line interface lets you enter lines of code into a shell (also called a console) and that code instructs your computer to perform a specific task. Throughout this book I may use the terms command line, shell, and console interchangeably. You'll learn about using the command line in the Command Line Basics chapter.

The shell is a very direct and powerful way to manipulate a computer. You can produce wonderful creations that help thousands of people, or you can wreak havoc on yourself and on others. Like Benjamin Parker[1] once said: "With great power comes great responsibility."

There are several popular shell programs but in this book we'll be using a shell called Bash because it's the default shell program on Mac and Ubuntu.

[1] https://en.wikipedia.org/wiki/Uncle_Ben

Getting Unix

> An indispensible thing never has much value. - Russian proverb

Mac & Ubuntu Users

If you're using a Mac or you're using the Ubuntu operating system find a program called **Terminal** and open it. You can skip the next section about Windows.

Windows

If you're using the latest version of Windows 10 you should enable and install Bash on Ubuntu on Windows. You can find the installation guide from Microsoft here[2].

If you don't have the latest version of Windows 10 you should download VirtualBox[3] and then set up the latest version of Ubuntu with VirtualBox. Instructions for doing this tend to change slightly over time, so I suggest using Google to search for "how to install Ubuntu on Windows with VirtualBox" and then you can follow the instructions that you find.

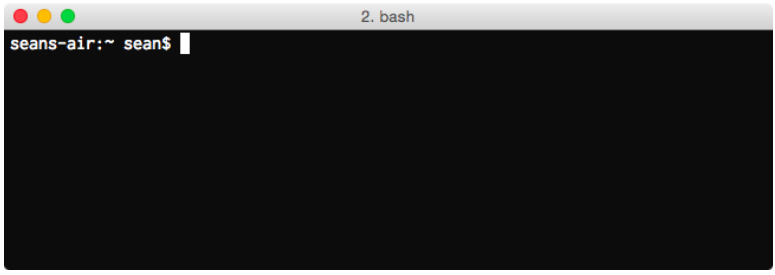
[2] https://msdn.microsoft.com/en-us/commandline/wsl/install_guide
[3] https://www.virtualbox.org/

Command Line Basics

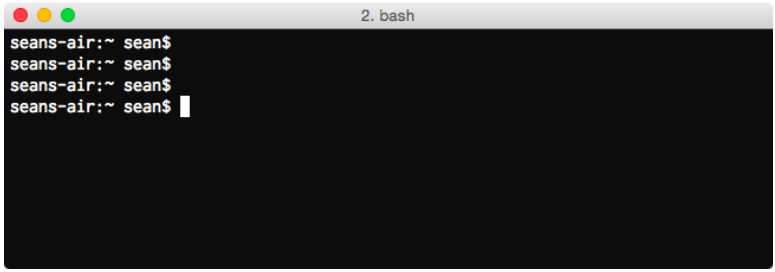
Of a small spark a great fire. - Gaelic proverb

Hello Terminal!

Once you have opened up Terminal then you should see a window that looks something like this:



What you're looking at is the bash shell! Your shell will surely look different than mine, but all bash shells have the same essential parts. As you can see in my shell it says `seans-air:~ sean$`. This string of characters is called the **prompt**. You type command line commands after the prompt. The prompt is just there to let you know that the shell is ready for you to type in a command. Press `Enter` on your keyboard a few times to see what happens with the prompt. Your shell should now look like this:



If you don't type anything after the prompt and you press enter then nothing happens and you get a new prompt under the old prompt. The white rectangle after the prompt is just a cursor that allows you to edit what you've typed into the shell. Your cursor might look like a rectangle, a line, or an underscore, but all cursors behave the same way. After typing something into the command line you can move the cursor back and forth with the left and right arrow keys, just like you would when typing an email.

Don't you think your shell looks messy with all of those old prompts? Don't worry, you're about to learn your first shell command which will clear up your shell! Type `clear` at the prompt and then hit enter. Voila! Your shell is back to how you started.

Every command line command is actually a little computer program, even commands as simple as `clear`. These commands all tend to have the following structure:

```
[command] [options] [arguments]
```

Some simple commands like `clear` don't require any options or arguments. Options are usually preceded by a hyphen (-) and they tweak the behavior of the command. Arguments can be names of files, raw data, or other options that the command

requires. A simple command that has an argument is `echo`. The `echo` command prints a phrase to the console. Enter `echo 'Hello World!'` into the command line to see what happens:

```
echo 'Hello World!'


## Hello World!
```

We'll be using the above syntax for the rest of the book, where on one line there will be a command that I've entered into the command line, and then below that command the console output of the command will appear (if there is any console output). You can use `echo` to print any phase surrounded by double quotes (`""`) to the console.

If you want to see the last command press the `Up` arrow key. You can press `Up` and `Down` in order to scroll through the history of commands that you've entered. If you want to re-execute a past command, you can scroll to that command then press `Enter`. Try getting back to the `echo "Hello World!"` command and execute it again.

Summary

- You type command line commands after the prompt.
- `clear` will clean up your terminal.
- `echo` prints text to your terminal.
- You can scroll through your command history with the `Up` and `Down` arrow keys.

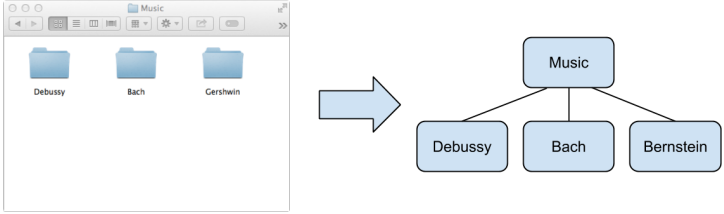
Exercises

1. Print your name to the terminal.
2. Clear your terminal after completing #1.

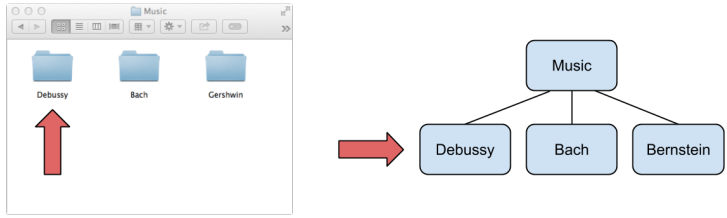
Navigating the Command Line

You've learned two command line commands (`clear` and `echo`) which is pretty good! Before you learn more commands we need to discuss how files and folders are organized on your computer.

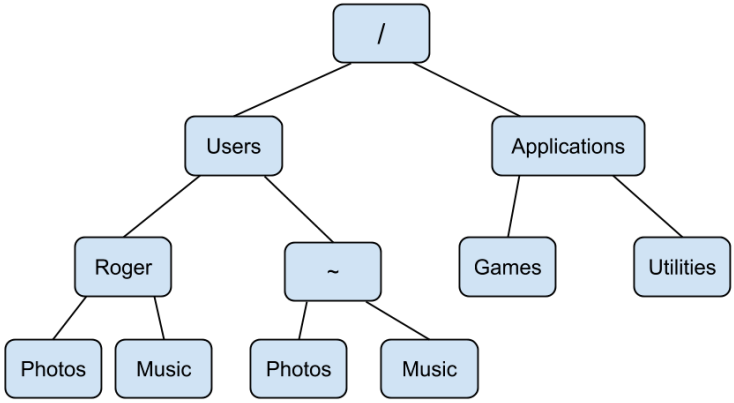
Computers are organized in a hierarchy of folders, where a folder can contain many folders and files. People who use Unix often refer to folders as directories and these terms are interchangeable. This directory hierarchy forms a tree, like the diagram below. You can use the command line to navigate these trees on your computer.



As you can see in the image below, my Debussy directory is contained in my Music directory. This is the simplest case of how directories are structured.



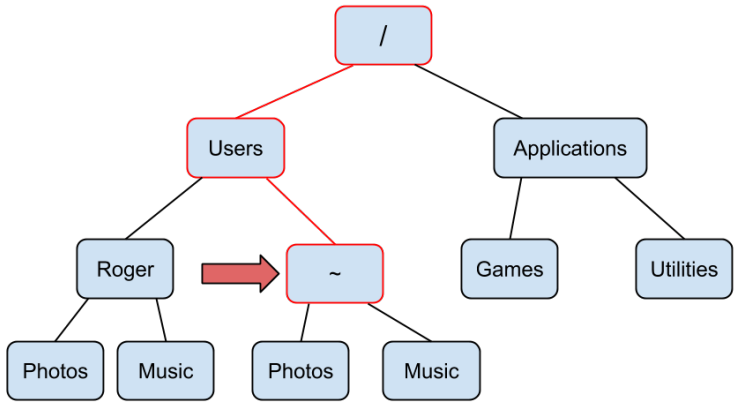
The directory structure on most computers is much more complicated, but the structure on your computer probably looks something like this:



There are a few special directories that you should be aware of on your computer. The directory at the top of this tree is called the root directory. The root directory contains all other directories, and is represented by a slash (/).

The home directory is another special directory that is represented by a tilde (~). Your home directory contains your personal files, like your photos, documents, and the contents of your desktop. When you first open up your shell you usually start off in your home directory. Imagine tracing all of the

directories from your root directory to the directory you're currently in. This sequence of directories is called a **path**. The diagram below illustrates the path from a hypothetical root directory to the home directory.



This path can be written as `/Users/sean`.

Open the command line if you closed it. Your shell starts in your home directory. Whatever directory your shell is in is called the **working directory**. Enter the `pwd` command into your shell to **p**rint the **w**orking **d**irectory.

```
pwd
```

```
## /Users/sean
```

You can change your working directory using the `cd` command. If you use the `cd` command without any arguments then your working directory is changed to your home directory.

Enter `cd` into the command line and then enter `pwd`.

```
cd
pwd


## /Users/sean
```

You were in your working directory to start, and by entering `cd` into the command line you did technically **c**hange **d**irectory, you just changed it to your home directory (the directory you were in to begin with). To use `cd` to change your working directory to a directory other than your home directory, you need to provide `cd` with the path to another directory as an argument. You can specify a path as either a path that is **relative** to your current directory, or you can specify the **absolute** path to a directory starting from the root of your computer. Let's say we simply want to change the working directory to one of the folders that is inside our home directory. First we need to be able to see which folders are in our working directory. You can list the files and folders in a directory using the `ls` command. Let's use the `ls` command in our home directory to list the files and folders contained within it.

```
ls


## Desktop
## Documents
## Photos
## Music
## todo.txt
```

It looks like I have four folders and one text file in my home directory. Now let's switch into the Music directory:

```
cd Music
```

As you can see the path to the current working directory has changed:

```
pwd
```

```
## /Users/sean/Music
```

I specified a **relative** path when I entered `cd Music`. The path to the Music directory is just `Music/` relative to my previous working directory. I can go back to `/Users/sean/` with the command `cd ..` which changes the working directory to the folder above the current working directory:

```
cd ..
pwd
```

```
## /Users/sean
```

Notice that `..` is also a relative path, since it specifies the directory above your current working directory. Similarly `.` is the path to your current working directory. Therefore since my current working directory is `/Users/sean` then `cd Music` is the same as `cd ./Music`.

I can `cd` to any folder as long as I know the **absolute** path to that folder. For example I can `cd` to `/Users/sean/Music` by entering the following into the shell:

```
cd ~/Music
pwd


## /Users/sean/Music
```

It doesn't matter what directory I'm in since I'm using an absolute path, I can jump straight to that directory (Remember that ~ is a shortcut for the path to your home folder). Of course you shouldn't expect yourself to have every absolute path on your computer memorized! You can use a terminal feature called **tab completion** in order to speed up typing paths and other commands. Enter the following into your shell, and then try pressing the `Tab` key (on some machines you need to press it twice):

```
cd ~/
```

(press `Tab`)

```
## Desktop
## Documents
## Photos
## Music
## todo.txt
```

Pressing tab shows you a list of all files and folders inside of the ~/ directory. Now I'm going to type ~/D into my terminal and you can see what happens when I press tab again:

```
cd ~/D
```

(press `Tab`)

```
## Desktop
## Documents
```

Since I added a "D" to the path, only folders with names that start with a "D" are listed. If I type `cd ~/De` into the console and then press `Tab` then the command will autocomplete to `cd ~/Desktop/`. If I press tab again, the console will list all of the files and folders on my desktop.

Make sure to pause and try this yourself in your own terminal! You won't have the same files or folders that I do, but you should try using `cd` and tab completion with directories and files that start with the same letters.

Summary

- You can identify a specific file or folder by its path.
- The root directory (`/`) contains all of the folders and files on your computer.
- Your home directory (`~`) is the directory where your terminal always starts.
- Use the `cd` command to **c**hange your working **d**irectory.
- The `pwd` command will **p**rint the **w**orking **d**irectory.
- The `ls` command will **ls**t files and folders in a directory.

Exercises

1. Set your working directory to the root directory.
2. Set your working directory to your home directory using three different commands.
3. Find a folder on your computer using your file and folder browser, and then set your working directory to that folder using the terminal.
4. List all of the files and folders in the directory you navigated to in #3.

Creation and Inspection

Now that you can fluidly use your terminal to bound between directories all over your computer I'll show you some actions you can perform on folders and files. One of the first actions you'll probably want to take when opening up a fresh terminal is to create a new folder or file. You can **m**a**k**e a **dir**ectory with the `mkdir` command, followed by the path to the new directory. First let's look at the contents of my home directory:

```
cd
ls
```

```
## Desktop
## Documents
## Photos
## Music
## todo.txt
```

I want to create a new directory to store some code files I'm going to write later, so I'll use `mkdir` to create a new directory called `Code`:

```
mkdir Code
ls
```

```
## Desktop
## Documents
## Photos
## Music
## todo.txt
## Code
```

It worked! Notice that the argument `Code` to `mkdir` is a relative path, however I could have specified an absolute path. In general you should expect Unix tools that take paths as arguments to accept both relative and absolute paths.

There are a few different ways to create a new file on the command line. The most simple way to create a blank file is to use the `touch` command, followed by the path to the file you want to create. In this example I'm going to create a new journal entry using touch:

```
touch journal-2017-01-24.txt
ls
```

```
## Desktop
## Documents
## Photos
## Music
## todo.txt
## Code
## journal-2017-01-24.txt
```

A new file has been created! I've been using `ls` to list the files and folders in the current directory, but using `ls` alone doesn't differentiate between which of the listed items are folders and which are files. Thankfully you can use the `-l` option with `ls` in order to get a long listing of files in a directory.

```
ls -l
```

```
## drwxr-xr-x  2 sean  staff  68 Jan 24 12:31 Code
## drwxr-xr-x  2 sean  staff  94 Jan 20 12:44 Des\
ktop
## drwxr-xr-x  2 sean  staff  24 Jan 20 12:44 Doc\
uments
## drwxr-xr-x  2 sean  staff  68 Jan 20 12:36 Mus\
ic
## drwxr-xr-x  2 sean  staff  68 Jan 20 12:35 Pho\
tos
## -rw-r--r--  1 sean  staff  90 Jan 24 11:33 jou\
rnal-2017-01-24.txt
## -rw-r--r--  1 sean  staff  70 Jan 24 10:58 tod\
o.txt
```

There is a row in the resulting table for each file or folder. If the entry in the first column is a `d`, then the row in the

table corresponds to a **d**irectory, otherwise the information in the row corresponds to a file. As you can see in my home directory there are five directories and two files. The string of characters following the `d` in the case of a directory or following the first - in the case of a file represent the permissions for that file or directory. We'll cover permissions in a later section. The columns of this table also show who created the file, the group that the creator of the file belongs to (we'll cover groups later when we cover permissions), the size of the file, the time and date when the file was last modified, and then finally the name of the file.

Now that we've created a file there are a few different ways that we can inspect and edit this file. First let's use the `wc` command to view the **w**ord **c**ount and other information about the file:

```
wc todo.txt
```

```
##       3      14      70 todo.txt
```

The `wc` command displays the number of lines in a file followed by the number of words and then the number of characters. Since this file looks pretty small (only three lines) let's try printing it to the console using the `cat` command.

```
cat todo.txt
```

```
## - email Jeff
## - write letter to Aunt Marie
## - get groceries for Shabbat
```

The `cat` command is often used to print text files to the terminal, despite the fact that it's really meant to con**cat**enate files. You can see this concatenation in action in the following example:

```
cat todo.txt todo.txt
```

```
## - email Jeff
## - write letter to Aunt Marie
## - get groceries for Shabbat
## - email Jeff
## - write letter to Aunt Marie
## - get groceries for Shabbat
```

The `cat` command will combine every text file that is provided as an argument.

Let's take a look at how we could view a larger file. There's a file inside the Documents directory:

```
ls Documents
```

```
## a-tale-of-two-cities.txt
```

Let's examine this file to see if it's reasonable to read it with `cat`:

```
wc Documents/a-tale-of-two-cities.txt
```

```
##      17    1005    5799 Documents/a-tale-of-tw\
o-cities.txt
```

Wow, over 1000 words! If we use `cat` on this file it's liable to take up our entire terminal. Instead of using `cat` for this large file we should use `less`, which is a program designed for viewing multi-page files. Let's try using `less`:

```
less Documents/a-tale-of-two-cities.txt
```

```
I. The Period

It was the best of times,
it was the worst of times,
it was the age of wisdom,
it was the age of foolishness,
it was the epoch of belief,
it was the epoch of incredulity,
it was the season of Light,
it was the season of Darkness,
it was the spring of hope,
it was the winter of despair,
we had everything before us, we had nothing befor\
e us, we were all going direct
Documents/a-tale-of-two-cities.txt
```

You can scroll up and down the file line-by-line using the up and down arrow keys, and if you want to scroll faster you can

use the `spacebar` to go to the next page and the `b` key to go to the previous page. In order to quit `less` and go back to the prompt press the `q` key.

As you can see the `less` program is a kind of Unix tool with behavior that we haven't seen before because it "takes over" your terminal. There are a few programs like this that we'll discuss throughout this book.

There are also two easy to remember programs for glimpsing the beginning or end of a text file: `head` and `tail`. Let's quickly use `head` and `tail` on `a-tale-of-two-cities.txt`:

```
head Documents/a-tale-of-two-cities.txt
```

```
## I. The Period
##
## It was the best of times,
## it was the worst of times,
## it was the age of wisdom,
## it was the age of foolishness,
## it was the epoch of belief,
## it was the epoch of incredulity,
## it was the season of Light,
## it was the season of Darkness,
```

As you can see `head` prints the first ten lines of the file to the terminal. You can specify the number of lines printed with the `-n` option followed by the number of lines you'd like to see:

```
head -n 4 Documents/a-tale-of-two-cities.txt
```

```
## I. The Period
##
## It was the best of times,
## it was the worst of times,
```

The `tail` program works exactly the same way:

```
tail Documents/a-tale-of-two-cities.txt
```

```
of an atrocious murderer, and to-morrow of a wret\
ched pilferer who had robbed a
farmer's boy of sixpence.
All these things, and a thousand like them, came \
to pass in and close upon the
dear old year one thousand seven hundred and seve\
nty-five. Environed by them,
while the Woodman and the Farmer worked unheeded,\
 those two of the large jaws,
and those other two of the plain and the fair fac\
es, trod with stir enough,
and carried their divine rights with a high hand.\
 Thus did the year one
thousand seven hundred and seventy-five conduct t\
heir Greatnesses, and myriads
of small creatures—the creatures of this chronicl\
e among the rest—along the
roads that lay before them.
```

We've now gone over a few tools for inspecting files, folders, and their contents including `ls`, `wc`, `cat`, `less`, `head`, and `tail`. Before the end of this section we should discuss a few more

techniques for creating and also editing files. One easy way to create a file is using **output redirection**. Output redirection stores text that would be normally printed to the command line in a text file. You can use output redirection by typing the greater-than sign (`>`) at the end of a command followed by the name of the new file that will contain the output from the proceeding command. Let's try an example using `echo`:

```
echo "I'm in the terminal."
```

```
## I'm in the terminal.
```

```
echo "I'm in the file." > echo-out.txt
```

Only the first command printed output to the terminal. Let's see if the second command worked:

```
ls
```

```
## Desktop
## Documents
## Photos
## Music
## todo.txt
## Code
## journal-2017-01-24.txt
## echo-out.txt
```

```
cat echo-out.txt
```

```
## I'm in the file.
```

Looks like it worked! You can also **append** text to the end of a file using two greater-than signs (`>>`). Let's try this feature out:

```
echo "I have been appended." >> echo-out.txt
cat echo-out.txt
```

```
## I'm in the file.
## I have been appended.
```

Now for a **word of warning**. Imagine that I want to append another line to the end of `echo-out.txt`, so typed `echo "A third line." > echo-out.txt` into the terminal when really I meant to type `echo "A third line." >> echo-out.txt` (notice I used `>` when I meant to use `>>`). Let's see what happens:

```
echo "A third line." > echo-out.txt
cat echo-out.txt
```

```
## A third line.
```

Unfortunately I have unintentionally overwritten what was already contained in `echo-out.txt`. There's no undo button in Unix so I'll have to live with this mistake. This is the first of several lessons demonstrating the damage that you should try to avoid inflicting with Unix. Make sure to take extra care when executing commands that can modify or delete a file, a typo in the command can be potentially devastating. Thankfully there are a few strategies for protecting yourself from mistakes, including managing permissions for files, and tracking versions of your files with Git, which we will discuss thoroughly in a later chapter.

Finally we should discuss how to edit text files. There are several file editors that are available for your terminal including `vim`[4] and `emacs`[5]. Entire books have been written about how to use both of these text editors, and if you're interested in one of them you should look for resources online about how to use them. The one text editor we will discuss using is called `nano`. Just like `less`, `nano` uses your entire terminal window. Let's edit `todo.txt` using `nano`:

```
nano todo.txt
```

[4] https://en.wikipedia.org/wiki/Vim_(text_editor)
[5] https://en.wikipedia.org/wiki/Emacs

```
  GNU nano 2.0.6                File: todo.txt

- email Jeff
- write letter to Aunt Marie
- get groceries for Shabbat

^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev\
 Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next\
 Page  ^U UnCut Text ^T To Spell
```

Once you've started `nano` you can start editing the text file. The top line of the `nano` editor shows the file you're currently working on, and the bottom two lines show a few commands that you can use in `nano`. The carrot character (`^`) represents the `Control` key on your keyboard, so you can for example type `Control + O` in order to save the changes you've made to the text file, or `Control + X` in order to exit `nano` and go back to the prompt.

`nano` is a good editor for beginners because it works similarly to word processors you've used before. You can use the arrow keys in order to move your cursor around the file, and the rest of the keys on your keyboard work as expected. Let's add an item to my to-do list and then I'll save and exit `nano` by typing `Control + O` followed by `Control + X`.

```
  GNU nano 2.0.6                File: todo.txt

- email Jeff
- write letter to Aunt Marie
- get groceries for Shabbat
- write final section of "command line basics"


^G Get Help   ^O WriteOut   ^R Read File  ^Y Prev\
 Page  ^K Cut Text   ^C Cur Pos
^X Exit       ^J Justify    ^W Where Is   ^V Next\
 Page  ^U UnCut Text ^T To Spell
```

Now let's quickly check if those changes were saved correctly:

```
cat todo.txt
```

```
## - email Jeff
## - write letter to Aunt Marie
## - get groceries for Shabbat
## - write final section of "command line basics"
```

You can also create new text files with `nano`. Instead of using an existing path to a file as the argument to `nano`, use a path to a file that does not yet exist and then save your changes to that file.

Summary

- Use `mkdir` to create new directories.

- The `touch` command creates empty files.
- You can use `>` to redirect the output of a command into a file.
- `>>` will append command output to the end of a file.
- Print a text file to the command line using `cat`.
- Inspect properties of a text file with `wc`.
- Peak at the beginning and end of a text file with `head` and `tail`.
- Scroll through a large text file with `less`.
- `nano` is simple text editor.

Exercises

1. Create a new directory called `workbench` in your home directory.
2. Without changing directories create a file called `readme.txt` inside of `workbench`.
3. Append the numbers `1`, `2`, and `3` to `readme.txt` so that each number appears on it's own line.
4. Print `readme.txt` to the command line.
5. Use output redirection to create a new file in the `workbench` directory called `list.txt` which lists the files and folders in your home directory.
6. Find out how many characters are in `list.txt` without opening the file or printing it to the command line.

Migration and Destruction

In this section we'll discuss moving, renaming, copying, and deleting files and folders. First let's revisit the contents of our current working directory:

```
ls
```

```
Code
Documents
Photos
Desktop
Music
echo-out.txt
journal-2017-01-24.txt
todo.txt
```

It's gotten a little sloppy, so let's clean this directory up. First I want to make a new directory to store all of my journal entries in called `Journal`. We already know how to do that:

```
mkdir Journal
```

Now I want to move my journal entry `journal-2017-01-24.txt` into the `Journal` directory. We can **move** it using the `mv` command. `mv` takes two arguments: first the path to the file or folder that you wish to move followed by the destination folder. Let's try using `mv` now:

```
mv journal-2017-01-24.txt Journal
ls
```

```
Code
Documents
Journal
Photos
Desktop
Music
echo-out.txt
todo.txt
```

Looks like it worked! I just realized however that I want to move the `Journal` directory into the `Documents` folder. Thankfully we can do this with `mv` in the same way:

```
mv Journal Documents
ls
```

```
Code
Documents
Photos
Desktop
Music
echo-out.txt
todo.txt
```

Let's just make sure it ended up in the right place:

```
ls Documents
```

```
Journal
a-tale-of-two-cities.txt
```

Looks good! Another hidden use of the `mv` command is that you can use it to rename files and folders. The first argument is the path to the folder or file that you want to rename, and the second argument is a path with the new name for the file or folder. Let's rename `todo.txt` so it includes today's date:

```
mv todo.txt todo-2017-01-24.txt
ls
```

```
Code
Documents
Photos
Desktop
Music
echo-out.txt
todo-2017-01-24.txt
```

Looks like it worked nicely. Similar to the `mv` command, the `cp` command **cop**ies a file or folder from one location to another. As you can see `cp` is used exactly like `mv` when copying files, the file or folder you wish to copy is the first argument, followed by the path to the folder where you want the copy to be made:

```
cp echo-out.txt Desktop
ls
```

```
Code
Documents
Photos
Desktop
Music
echo-out.txt
todo-2017-01-24.txt


ls Desktop


echo-out.txt
```

Be aware that there is one difference between copying files and folders, when copying folders you need to specify the `-r` option, which is short for *recursive*. This ensures that the underlying directory structure of the directory you wish to copy remains intact. Let's try copying my `Documents` directory into the `Desktop` directory:

```
cp -r Documents Desktop
ls Desktop


Documents
echo-out.txt
```

Finally, let's discuss how to delete files and folders with the command line. **A word of extreme caution**: in general I don't recommend deleting files or folders on the command line because as we've discussed before there is ***no undo button*** on

the command line. If you delete a file that is critical to your computer functioning you may cause irreparable damage. I *highly* recommend moving files or folders to a designated trash folder and then deleting them the way you would normally delete files and folders outside of the command line (The path to the Trash Bin is `~/.Trash` on Mac and `~/.local/share/Trash` on Ubuntu). If you decide to delete a file or folder on your computer make absolutely sure that the command you've typed is correct before you press `Enter`. If you do delete a file or folder by accident stop using your computer immediately and consult with a computer professional or your IT department so they can try to recover the file.

Now that you've been warned, let's discuss `rm`, the Avada Kedavra[6] of command line programs. When **rem**oving files `rm` only requires the path to a file in order to delete it. Let's test its destructive power on `echo-out.txt`:

```
rm echo-out.txt
ls
```

```
Code
Documents
Photos
Desktop
Music
todo-2017-01-24.txt
```

> I felt a great disturbance in the Force, as if millions of voices suddenly cried out in terror, and were suddenly silenced. - *Obi-wan Kenobi*

[6] https://en.wikipedia.org/wiki/Magic_in_Harry_Potter#Unforgivable_Curses

The file `echo-out.txt` is gone forever. Remember when we copied the entire `Documents` directory into `Desktop`? Let's get rid of that directory now. Just like when we were using `cp` the `rm` command requires you to use the `-r` option when deleting entire directories. Let's test this battle station:

```
ls Desktop
```

```
Documents
echo-out.txt
```

```
rm -r Desktop/Documents
ls Desktop
```

```
echo-out.txt
```

Now that the awesome destructive power of `rm` is on your side, you've learned the basics of the command line! See you in the next chapter for a discussion of more advanced command line topics.

Summary

- `mv` can be used for moving or renaming files or folders.
- `cp` can copy files or folders.
- You should try to avoid using `rm` which permanently removes files or folders.

Exercises

1. Create a file called `message.txt` in your home directory and move it into another directory.
2. Copy the `message.txt` you just moved into your home directory.
3. Delete both copies of `message.txt`. Try to do this without using `rm`.

Working with Unix

> It is not the knowing that is difficult, but the doing. - Chinese proverb

Self-Help

Each of the commands that we've discussed so far are thoroughly documented, and you can view their documentation using the `man` command, where the first argument to `man` is the command you're curious about. Let's take a look at the documentation for `ls`:

```
man ls
```

```
LS(1)                     BSD General Commands Ma\
nual                    LS(1)

NAME
     ls -- list directory contents

SYNOPSIS
     ls [-ABCFGHLOPRSTUW@abcdefghiklmnopqrstuwx1]\
 [file ...]

DESCRIPTION
```

```
     For each operand that names a file of a type\
 other than directory, ls
     displays its name as well as any requested, \
associated information.  For
:
```

The controls for navigating `man` pages are the same as they are for `less`. I often use `man` pages for quickly searching for an option that I've forgotten. Let's say that I forgot how to get `ls` to print a long list. After typing `man ls` to open the page, type `/` in order to start a search. Then type the word or phrase that you're searching for, in this case type in `long list` and then press `Enter`. The page jumps to this entry:

```
     -l      (The lowercase letter ``ell''.)  Lis\
t in long format.  (See below.)
             If the output is to a terminal, a to\
tal sum for all the file sizes is
             output on a line before the long lis\
ting.
```

Press the `n` key in order to search for the next occurrence of the word, and if you want to go to the previous occurrence type `Shift + n`. This method of searching also works with `less`. When you're finished looking at a `man` page type `q` to get back to the prompt.

The `man` command works wonderfully when you know which command you want to look up, but what if you've forgotten the name of the command you're looking for? You can use `apropos` to search all of the available commands and their descriptions. For example let's pretend that I forgot the name of my favorite command line text editor. You could type

`apropos editor` into the command line which will print a list of results:

```
apropos editor
```

```
## ed(1), red(1)                - text editor
## nano(1)                      - Nano's ANOther edit\
or, an enhanced free Pico clone
## sed(1)                       - stream editor
## vim(1)                       - Vi IMproved, a prog\
rammers text editor
```

The second result is `nano` which was just on the tip of my tongue! Both `man` and `apropos` are useful when a search is only a few keystrokes away, but if you're looking for detailed examples and explanations you're better off using a search engine if you have access to a web browser.

Summary

- Use `man` to look up the documentation for a command.
- If you can't think of the name of a command use `apropos` to search for a word associated with that command.
- If you have access to a web browser, using a search engine might be better than `man` or `apropos`.

Exercises

1. Use `man` to look up the flag for human-readable output from `ls`.

2. Get help with `man` by typing `man man` into the console.
3. Wouldn't it be nice if there was a calendar command? Use `apropos` to look for such a command, then use `man` to read about how that command works.

Get Wild

Let's go into my `Photos` folder in my home directory and take a look around:

```
pwd
```

```
## /Users/sean
```

```
ls
```

```
## Code
## Documents
## Photos
## Desktop
## Music
## todo-2017-01-24.txt
```

```
cd Photos
ls
```

```
## 2016-06-20-datasci01.png
## 2016-06-20-datasci02.png
## 2016-06-20-datasci03.png
## 2016-06-21-lab01.jpg
## 2016-06-21-lab02.jpg
## 2017-01-02-hiking01.jpg
## 2017-01-02-hiking02.jpg
## 2017-02-10-hiking01.jpg
## 2017-02-10-hiking02.jpg
```

I've just been dumping pictures and figures into this folder without organizing them at all! Thankfully (in the words of Dr. Jenny Bryan) I have an unwavering commitment to the ISO 8601 date standard[7] so at least I know when these photos were taken. Instead of using `mv` to move around each individual photo I can select groups of photos using the * wildcard. A **wildcard** is a character that represents other characters, much like how joker in a deck of cards can represent other cards in the deck. Wildcards are a subset of metacharacters, a topic which we will discuss in detail later on in this chapter. The * ("star") wildcard represents *zero or more of any character*, and it can be used to match names of files and folders in the command line. For example if I wanted to list all of the files in my Photos directory which have a name that starts with "2017" I could do the following:

```
ls 2017*
```

[7] https://twitter.com/JennyBryan/status/816143967695687684

```
## 2017-01-02-hiking01.jpg
## 2017-01-02-hiking02.jpg
## 2017-02-10-hiking01.jpg
## 2017-02-10-hiking02.jpg
```

Only the files starting with "2017" are listed! The command `ls 2017*` literally means: list the files that start with "2017" followed by zero or more of any character. As you can imagine using wildcards is a powerful tool for working with groups of files that are similarly named.

Let's walk through a few other examples of using the star wildcard. We could only list the photos starting with "2016":

```
ls 2016*
```

```
## 2016-06-20-datasci01.png
## 2016-06-20-datasci02.png
## 2016-06-20-datasci03.png
## 2016-06-21-lab01.jpg
## 2016-06-21-lab02.jpg
```

We could list only the files with names ending in `.jpg`:

```
ls *.jpg
```

```
## 2016-06-21-lab01.jpg
## 2016-06-21-lab02.jpg
## 2017-01-02-hiking01.jpg
## 2017-01-02-hiking02.jpg
## 2017-02-10-hiking01.jpg
## 2017-02-10-hiking02.jpg
```

In the case above the file name can start with a sequence of zero or more of any character, but the file name must end in `.jpg`. Or we could also list only the first photos from each set of photos:

```
ls *01.*
```

```
## 2016-06-20-datasci01.png
## 2016-06-21-lab01.jpg
## 2017-01-02-hiking01.jpg
## 2017-02-10-hiking01.jpg
```

All of the files above have names that are composed of a sequence of characters, followed by the adjacent characters `01.`, followed by another sequence of characters. Notice that if I had entered `ls *01*` into the console every file would have been listed since `01` is a part of all of the file names in my Photos directory.

Let's organize these photos by year. First let's create one directory for each year of photos:

```
mkdir 2016
mkdir 2017
```

Now we can move the photos using wildcards:

```
mv 2017-* 2017/
ls
```

```
## 2016
## 2016-06-20-datasci01.png
## 2016-06-20-datasci02.png
## 2016-06-20-datasci03.png
## 2016-06-21-lab01.jpg
## 2016-06-21-lab02.jpg
## 2017
```

Notice that I've moved all files that start with "2017-" into the 2017 folder! Now let's do the same thing for files with names starting with "2016-":

```
mv 2016-* 2016/
ls
```

```
## 2016
## 2017
```

Finally my photos are somewhat organized! Let's list the files in each directory just to make sure all was moved as planned:

```
ls 2016/
```

```
## 2016-06-20-datasci01.png
## 2016-06-20-datasci02.png
## 2016-06-20-datasci03.png
## 2016-06-21-lab01.jpg
## 2016-06-21-lab02.jpg
```

```
ls 2017/
```

```
## 2017-01-02-hiking01.jpg
## 2017-01-02-hiking02.jpg
## 2017-02-10-hiking01.jpg
## 2017-02-10-hiking02.jpg
```

Looks good! There are a few more wildcards beyond the star wildcard which we'll discuss in the next section where searching file names gets a little more advanced.

Summary

- Wildcards can represent many kinds and numbers of characters.
- The star wildcard (*) represents zero or more of any character.
- You can use wildcards on the command line in order to work with multiple files and folders.

Exercises

1. Before I organized the photos by year, what command would have listed all of the photos of type `.png`?
2. Before I organized the photos by year, what command would have deleted all of my hiking photos?
3. What series of commands would you use in order to put my figures for a data science course and the pictures I took in the lab into their own folders?

Search

Regular Expressions

The ability to search through files and folders can greatly improve your productivity using Unix. First we'll cover searching through text files. I recently downloaded a list of the names of the states in the US. Let's take a look at this file:

```
cd ~/Documents
ls
```

```
## canada.txt
## states.txt
```

```
wc states.txt
```

```
## 50      60     472 states.txt
```

It makes sense that there are 50 lines, but it's interesting that there are 60 total words. Let's a take a peak at the beginning of the file:

```
head states.txt
```

```
## Alabama
## Alaska
## Arizona
## Arkansas
## California
## Colorado
## Connecticut
## Delaware
## Florida
## Georgia
```

This file looks basically how you would expect it to look! You may recall from Chapter 3 that the kind of shell that we're using is the bash shell. Bash treats different kinds of data differently, and we'll dive deeper into data types in Chapter 5. For now all you need to know is that text data are called **strings**. A string could be a word, a sentence, a book, or a file or folder name. One of the most effective ways to search through strings is to use **regular expressions**. Regular expressions are strings that define patterns in other strings. You can use regular expressions to search for a sub-string contained within a larger string, or to replace one part of a string with another string.

One of the most popular tools for searching through text files is `grep`. The simplest use of `grep` requires two arguments: a regular expression and a text file to search. Let's see a simple example of `grep` in action and then I'll explain how it works:

```
grep "x" states.txt
```

```
## New Mexico
## Texas
```

In the command above, the first argument to `grep` is the regular expression `"x"`. The `"x"` regular expression represents one instance of the letter "x". Every line of the `states.txt` file that contains at least one instance of the letter "x" is printed to the console. As you can see New Mexico and Texas are the only two state names that contain the letter "x". Let's try searching for the letter "q" in all of the state names using `grep`:

```
grep "q" states.txt
```

Nothing is printed to the console because the letter "q" isn't in any of the state names. We can search for more than individual characters though. For example the following command will search for the state names that contain the word "New":

```
grep "New" states.txt
```

```
## New Hampshire
## New Jersey
## New Mexico
## New York
```

In the previous case the regular expression we used was simply `"New"`, which represents an occurrence of the string "New". Regular expressions are not limited to just being individual characters or words, they can also represent parts of words. For example I could search all of the state names that contain the string "nia" with the following command:

```
grep "nia" states.txt
```

```
## California
## Pennsylvania
## Virginia
## West Virginia
```

All of the state names above happen to end with the string "nia".

Metacharacters

Regular expressions aren't just limited to searching with characters and strings, the real power of regular expressions come from using **metacharacters**. Remember that metacharacters are characters that can be used to represent other characters. To take full advantage of all of the metacharacters we should use `grep`'s cousin `egrep`, which just extends `grep`'s

capabilities. The first metacharacter we should discuss is the `"."` (period) metacharacter, which represents *any* character. If for example I wanted to search `states.txt` for the character "i", followed by any character, followed by the character "g" I could do so with the following command:

```
egrep "i.g" states.txt
```

```
## Virginia
## Washington
## West Virginia
## Wyoming
```

The regular expression "i.g" matches the sub-string "irg" in V*irg*inia, and West V*irg*inia, and it matches the sub-string "ing" in Wash*ing*ton and Wyom*ing*. The period metacharacter is a stand-in for the "r" in "irg" and the "n" in "ing" in the example above. The period metacharacter is extremely liberal, for example the command `egrep "." states.txt` would return every line of states.txt since the regular expression `"."` would match one occurrence of any character on every line (there's at least one character on every line).

Besides characters that can represent other characters, there are also metacharacters called **quantifiers** which allow you to specify the number of times a particular regular expression should appear in a string. One of the most basic quantifiers is `"+"` (plus) which represents one or more occurrences of the proceeding expression. For example the regular expression "s+as" means: one or more "s" followed by "as". Let's see if any of the state names match this expression:

```
egrep "s+as" states.txt
```

```
## Arkansas
## Kansas
```

Both Arkan*sas* and Kan*sas* match the regular expression `"s+as"`. Besides the plus metacharacter there's also the `"*"` (star) metacharacter which represents zero or more occurrences of the preceding expression. Let's see what happens if we change `"s+as"` to `"s*as"`:

```
egrep "s*as" states.txt
```

```
## Alaska
## Arkansas
## Kansas
## Massachusetts
## Nebraska
## Texas
## Washington
```

As you can see the star metacharacter is much more liberal with respect to matching since many more state names are matched by `"s*as"`. There are more specific quantifies you can use beyond "zero or more" or "one or more" occurrences of an expression. You can use curly brackets (`{ }`) to specify an exact number of occurrences of an expression. For example the regular expression `"s{2}"` specifies exactly two occurrences of the character "s". Let's try using this regular expression:

```
egrep "s{2}" states.txt
```

```
## Massachusetts
## Mississippi
## Missouri
## Tennessee
```

Take note that the regular expression `"s{2}"` is equivalent to the regular expression `"ss"`. We could also search for state names that have between two and three adjacent occurrences of the letter "s" with the regular expression `"s{2,3}"`:

```
egrep "s{2,3}" states.txt
```

```
## Massachusetts
## Mississippi
## Missouri
## Tennessee
```

Of course the results are the same because there aren't any states that have "s" repeated three times.

You can use a **capturing group** in order to search for multiple occurrences of a string. You can create capturing groups within regular expressions by using parentheses (`"( )"`). For example if I wanted to search states.txt for the string "iss" occurring twice in a state name I could use a capturing group and a quantifier like so:

```
egrep "(iss){2}" states.txt
```

```
## Mississippi
```

We could combine more quantifiers and capturing groups to dream up even more complicated regular expressions. For example, the following regular expression describes three occurrences of an "i" followed by two of any character:

```
egrep "(i.{2}){3}" states.txt
```

```
## Mississippi
```

The complex regular expression above still only matches "Mississippi".

Character Sets

For the next couple of examples we're going to need some text data beyond the names of the states. Let's just create a short text file from the console:

```
touch small.txt
echo "abcdefghijklmnopqrstuvwxyz" >> small.txt
echo "ABCDEFGHIJKLMNOPQRSTUVWXYZ" >> small.txt
echo "0123456789" >> small.txt
echo "aa bb cc" >> small.txt
echo "rhythms" >> small.txt
echo "xyz" >> small.txt
echo "abc" >> small.txt
echo "tragedy + time = humor" >> small.txt
echo "http://www.jhsph.edu/" >> small.txt
echo "#%&-=***=-&%#" >> small.txt
```

In addition to quantifiers there are also regular expressions for describing sets of characters. The `\w` metacharacter corresponds to all "word" characters, the `\d` metacharacter corresponds to all "number" characters, and the `\s` metacharacter corresponds to all "space" characters. Let's take a look at using each of these metacharacters on small.txt:

```
egrep "\w" small.txt
```

```
## abcdefghijklmnopqrstuvwxyz
## ABCDEFGHIJKLMNOPQRSTUVWXYZ
## 0123456789
## aa bb cc
## rhythms
## xyz
## abc
## tragedy + time = humor
## http://www.jhsph.edu/
```

```
egrep "\d" small.txt
```

```
## 0123456789
```

```
egrep "\s" small.txt
```

```
## aa bb cc
## tragedy + time = humor
```

As you can see in the example above, the `\w` metacharacter matches all letters, numbers, and even the underscore character (_). We can see the compliment of this grep by adding the `-v` flag to the command:

```
egrep -v "\w" small.txt
```

```
## #%&-=***=-&%#
```

The `-v` flag (which stands for invert match) makes `grep` return all of the lines not matched by the regular expression. Note that the character sets for regular expressions also have their inverse sets: `\W` for non-words, `\D` for non-digits, and `\S` for non-spaces. Let's take a look at using `\W`:

```
egrep "\W" small.txt
```

```
## aa bb cc
## tragedy + time = humor
## http://www.jhsph.edu/
## #%&-=***=-&%#
```

The returned strings all contain non-word characters. Note the difference between the results of using the invert flag `-v` versus using an inverse set regular expression.

In addition to general character sets we can also create specific character sets using square brackets (`[ ]`) and then including the characters we wish to match in the square brackets. For example the regular expression for the set of vowels is `[aeiou]`. You can also create a regular expression for the compliment of a set by including a caret (`^`) in the beginning of a set. For example the regular expression `[^aeiou]` matches all characters that are not vowels. Let's test both on small.txt:

```
egrep "[aeiou]" small.txt
```

```
## abcdefghijklmnopqrstuvwxyz
## aa bb cc
## abc
## tragedy + time = humor
## http://www.jhsph.edu/
```

Notice that the word "rhythms" does not appear in the result (it's the longest word without any vowels that I could think of).

```
egrep "[^aeiou]" small.txt
```

```
## abcdefghijklmnopqrstuvwxyz
## ABCDEFGHIJKLMNOPQRSTUVWXYZ
## 0123456789
## aa bb cc
## rhythms
## xyz
## abc
## tragedy + time = humor
## http://www.jhsph.edu/
## #%&-=***=-&%#
```

Every line in the file is printed, because every line contains at least one non-vowel! If you want to specify a range of characters you can use a hyphen (-) inside of the square brackets. For example the regular expression `[e-q]` matches all of the lowercase letters between "e" and "q" in the alphabet inclusively. Case matters when you're specifying character sets, so if you wanted to only match uppercase characters you'd need to use `[E-Q]`. To ignore the case of your match you could combine the character sets with the `[e-qE-Q]` regex (short for regular expression), or you could use the `-i` flag with `grep` to **i**gnore the case. Note that the `-i` flag will work for any provided regular expression, not just character sets. Let's take a look at some examples using the regular expressions that we just described:

```
egrep "[e-q]" small.txt
```

```
## abcdefghijklmnopqrstuvwxyz
## rhythms
## tragedy + time = humor
## http://www.jhsph.edu/
```

```
egrep "[E-Q]" small.txt
```

```
## ABCDEFGHIJKLMNOPQRSTUVWXYZ
```

```
egrep "[e-qE-Q]" small.txt
```

```
## abcdefghijklmnopqrstuvwxyz
## ABCDEFGHIJKLMNOPQRSTUVWXYZ
## rhythms
## tragedy + time = humor
## http://www.jhsph.edu/
```

Escaping, Anchors, Odds, and Ends

One issue you may have thought about during our little exploration of regular expressions is how to search for certain punctuation marks in text considering that those same symbols are used as metacharacters! For example, how would you find a plus sign (+) in a line of text since the plus sign is **also** a metacharacter? The answer is simply using a backslash (\) before the plus sign in a regex, in order to "escape" the metacharacter functionality. Here are a few examples:

```
egrep "\+" small.txt
```

```
## tragedy + time = humor
```

```
egrep "\." small.txt
```

```
## http://www.jhsph.edu/
```

There are three more metacharacters that we should discuss, and two of them come as a pair: the caret (^), which represents the start of a line, and the dollar sign ($) which represents the end of line. These "anchor characters" only match the beginning and ends of lines when coupled with other regular expressions. For example, going back to looking at states.txt, I could search for all of the state names that begin with "M" with the following command:

```
egrep "^M" states.txt
```

```
## Maine
## Maryland
## Massachusetts
## Michigan
## Minnesota
## Mississippi
## Missouri
## Montana
```

Or we could search for all of the states that end in "s":

```
egrep "s$" states.txt
```

```
## Arkansas
## Illinois
## Kansas
## Massachusetts
## Texas
```

There's a mnemonic that I love for remembering which metacharacter to use for each anchor: "First you get the **power**, then you get the **money**." The caret character is used for exponentiation in many programming languages, so "power" (^) is used for the beginning of a line and "money" ($) is used for the end of a line.

Finally, let's talk about the "or" metacharacter (|), which is also called the "pipe" character. This metacharacter allows you to match either the regex on the right or on the left side of the pipe. Let's take a look at a small example:

```
egrep "North|South" states.txt
```

```
## North Carolina
## North Dakota
## South Carolina
## South Dakota
```

In the example above we're searching for lines of text that contain the words "North" or "South". You can also use multiple pipe characters to, for example, search for lines that contain the words for all of the cardinal directions:

```
egrep "North|South|East|West" states.txt
```

```
## North Carolina
## North Dakota
## South Carolina
## South Dakota
## West Virginia
```

Just two more notes on `grep`: you can display the line number that a match occurs on using the `-n` flag:

```
egrep -n "t$" states.txt
```

```
## 7:Connecticut
## 45:Vermont
```

And you can also `grep` multiple files at once by providing multiple file arguments:

```
egrep "New" states.txt canada.txt
```

```
## states.txt:New Hampshire
## states.txt:New Jersey
## states.txt:New Mexico
## states.txt:New York
## canada.txt:Newfoundland and Labrador
## canada.txt:New Brunswick
```

You now have the power to do some pretty complicated string searching using regular expressions! Imagine you wanted to search for all of the state names that both begin and end with a vowel. Now you can:

```
egrep "^[AEIOU]{1}.+[aeiou]{1}$" states.txt
```

```
## Alabama
## Alaska
## Arizona
## Idaho
## Indiana
## Iowa
## Ohio
## Oklahoma
```

I know there a many metacharacters to keep track of here so below I've included a table with several of the metacharacters we've discussed in this chapter:

Metacharacter	Meaning
.	Any Character
\w	A Word
\W	Not a Word
\d	A Digit
\D	Not a Digit
\s	Whitespace
\S	Not Whitespace
[def]	A Set of Characters
[^def]	Negation of Set
[e-q]	A Range of Characters
^	Beginning of String
$	End of String
\n	Newline
+	One or More of Previous
*	Zero or More of Previous
?	Zero or One of Previous
|	Either the Previous or the Following
{6}	Exactly 6 of Previous
{4, 6}	Between 4 and 6 or Previous
{4, }	More than 4 of Previous

If you want to experiment with writing regular expressions before you use them I highly recommend playing around with http://regexr.com/.

`find`

If you want to find the location of a file or the location of a group of files you can use the `find` command. This command has a specific structure where the first argument is the directory where you want to begin the search, and

all directories contained within that directory will also be searched. The first argument is then followed by a flag that describes the method you want to use to search. In this case we'll only be searching for a file by its name, so we'll use the `-name` flag. The `-name` flag itself then takes an argument, the name of the file that you're looking for. Let's go back to the home directory and look for some files from there:

```
cd
pwd


## /Users/sean
```

Let's start by looking for a file called states.txt:

```
find . -name "states.txt"


## ./Documents/states.txt
```

Right where we expected it to be! Now let's try searching for all `.jpg` files:

```
find . -name "*.jpg"
```

```
## ./Photos/2016-06-21-lab01.jpg
## ./Photos/2016-06-21-lab02.jpg
## ./Photos/2017/2017-01-02-hiking01.jpg
## ./Photos/2017/2017-01-02-hiking02.jpg
## ./Photos/2017/2017-02-10-hiking01.jpg
## ./Photos/2017/2017-02-10-hiking02.jpg
```

Good file hunting out there!

Summary

- `grep` and `egrep` can be used along with regular expressions to search for patterns of text in a file.
- Metacharacters are used in regular expressions to describe patterns of characters.
- `find` can be used to search for the names of files in a directory.

Exercises

1. Search `states.txt` and `canada.txt` for lines that contain the word "New".
2. Make five text files containing the names of states that don't contain one of each of the five vowels.
3. Download the GitHub repository for this book and find out how many `.html` files it contains.

Configure

History

Near the start of this book we discussed how you can browse the commands that you recently entered into the prompt using the `Up` and `Down` arrow keys. Bash keeps track of all of your recent commands, and you can browse your command history two different ways. The commands that we've used since opening our terminal can be accessed via the `history` command. Let's try it out:

```
history
```

```
## ...
## 48 egrep "^M" states.txt
## 49 egrep "s$" states.txt
## 50 egrep "North|South" states.txt
## 51 egrep "North|South|East|West" states.txt
## 52 egrep -n "t$" states.txt
## 53 egrep "New" states.txt canada.txt
## 54 egrep "^[AEIOU]{1}.+[aeiou]{1}$" states.txt
## 55 cd
## 56 pwd
## 57 find . -name "states.txt"
## 58 find . -name "*.jpg"
## 59 history
```

We've had our terminal open for a while so there are tons of commands in our history! Whenever we close a terminal our recent commands are written to the `~/.bash_history` file. Let's a take a look at the beginning of this file:

```
head -n 5 ~/.bash_history
```

```
## echo "Hello World!"
## pwd
## cd
## pwd
## ls
```

Looks like the very first commands we entered into the terminal! Searching your `~/.bash_history` file can be particularly useful if you're trying to recall a command you've used in the past. The `~/.bash_history` file is just a regular text file, so you can search it with `grep`. Here's a simple example:

```
grep "canada" ~/.bash_history
```

```
## egrep "New" states.txt canada.txt
```

Customizing Bash

Besides `~/.bash_history`, another text file in our home directory that we should be aware of is `~/.bash_profile`. The `~/.bash_profile` is a list of Unix commands that are run every time we open our terminal, usually with a different command on every line. One of the most common commands used in a `~/.bash_profile` is the `alias` command, which creates a shorter name for a command. Let's take a look at a `~/.bash_profile`:

```
alias docs='cd ~/Documents'
alias edbp='nano ~/.bash_profile'
```

The first `alias` creates a new command `docs`. Now entering `docs` into the command line is the equivalent of entering `cd ~/Documents` into the comamnd line. Open let's edit our `~/.bash_profile` with `nano`. If there's anything in your `~/.bash_profile` already then start adding lines at the end of the file. Add the line `alias docs='cd ~/Documents'`, then save the file and quit `nano`. In order to make the changes to our `~/.bash_profile` take effect we need to run `source ~/.bash_profile` in the console:

```
source ~/.bash_profile
```

Now let's try using `docs`:

```
docs
pwd


## /Users/sean/Documents
```

It works! Setting different `aliases` allows you to save time if there are long commands that use often. In the example `~/.bash_profile` above, the second line, `alias edbp='nano ~/.bash_profile'` creates the command `edbp` (**ed**it **b**ash **p**rofile) so that you can quickly add `aliases`. Try adding it to your `~/.bash_profile` and take your new command for a spin!

There are a few other details about the `~/.bash_profile` that are important when you're writing software which we'll discuss in the Bash Programming chapter.

Summary

- `history` displays what commands we've entered into the console since opening our current terminal.
- The `~/.bash_history` file lists commands we've used in the past.
- `alias` creates a command that can be used as a substitute for a longer command that we use often.
- The `~/.bash_profile` is a text file that is run every time we start a shell, and it's the best place to assign `aliases`.

Differentiate

It's important to be able to examine differences between files. First let's make two small simple text files in the Documents directory.

```
cd ~/Documents
head -n 4 states.txt > four.txt
head -n 6 states.txt > six.txt
```

If we want to look at which lines in these files are different we can use the `diff` command:

```
diff four.txt six.txt
```

```
## 4a5,6
## > California
## > Colorado
```

Only the differing lines are printed to the console. We could also compare differing lines in a side-by-side comparison using `sdiff`:

```
sdiff four.txt six.txt
```

```
## Alabama                Alabama
## Alaska                 Alaska
## Arizona                Arizona
## Arkansas               Arkansas
##                      > California
##                      > Colorado
```

In a common situation you might be sent a file, or you might download a file from the internet that comes with code known as a **checksum** or a **hash**. Hashing programs generate a unique code based on the contents of a file. People distribute hashes with files so that we can be sure that the file we think we've downloaded is the genuine file. One way we can prevent malicious individuals from sending us harmful files is to check to make sure the computed hash matches the provided hash. There are a few commonly used file hashes but we'll talk about two called MD5 and SHA-1.

Since hashes are generated based on file contents, then two identical files should have the same hash. Let's test this my making a copy of `states.txt`.

```
cp states.txt states_copy.txt
```

To compute the MD5 hash of a file we can use the `md5` command:

```
md5 states.txt
```

```
## MD5 (states.txt) = 8d7dd71ff51614e69339b03bd1c\
b86ac
```

```
md5 states_copy.txt
```

```
## MD5 (states_copy.txt) = 8d7dd71ff51614e69339b0\
3bd1cb86ac
```

As we expected they're the same! We can compute the SHA-1 hash using the `shasum` command:

```
shasum states.txt
```

```
## 588e9de7ffa97268b2448927df41760abd3369a9  stat\
es.txt
```

```
shasum states_copy.txt
```

```
## 588e9de7ffa97268b2448927df41760abd3369a9  stat\
es_copy.txt
```

Once again, both copies produce the same hash. Let's make a change to one of the files, just to illustrate the fact that the hash changes if file contents are different:

```
head -n 5 states_copy.txt > states_copy.txt
shasum states_copy.txt
```

```
## b1c1c805f123f31795c77f78dd15c9f7ac5732d4  stat\
es_copy.txt
```

Summary

- The `md5` and `shasum` commands use different algorithms to create codes (called hashes or checksums) that are unique to the contents of a file.
- These hashes can be used to ensure that a file is genuine.

Pipes

One of the most powerful features of the command line is skilled use of the **pipe** (`|`) which you can usually find above the backslash (`\`) on your keyboard. The pipe allows us to take the output of a command, which would normally be printed to the console, and use it as the input to another command. It's like fitting an actual pipe between the end of one program and connecting it to the top of another program! Let's take a look at a basic example. We know the `cat` command takes the contents of a text file and prints it to the console:

```
cd ~/Documents
cat canada.txt
```

```
## Nunavut
## Quebec
## Northwest Territories
## Ontario
## British Columbia
## Alberta
## Saskatchewan
## Manitoba
## Yukon
## Newfoundland and Labrador
## New Brunswick
## Nova Scotia
## Prince Edward Island
```

This output from `cat canada.txt` will go into our pipe, and we'll attach the dispensing end of the pipe to `head`, which we use to look at the first few lines of a file:

```
cat canada.txt | head -n 5
```

```
Nunavut
Quebec
Northwest Territories
Ontario
British Columbia
```

Notice that this is the same result we would get from `head -n 5 canada.txt`, we just used `cat` to illustrate how the pipe works. The general syntax of the pipe is `[program that produces output] | [program uses pipe output as input instead of a file]`.

A more common and useful example where we could use the pipe is answering the question: "How many US states end in a vowel?" We could use `grep` and regular expressions to list all of the state names that end with a vowel, then we could use `wc` to count all of the matching state names:

```
grep "[aeiou]$" states.txt | wc -l
```

```
## 32
```

The pipe can also be used multiple times in one command in order to take the output from one piped command and use it as the input to yet another program! For example we could use three pipes with `ls`, `grep`, and `less` so that we could scroll through the files in out current directory were created in February:

```
ls -al | grep "Feb" | less
```

```
-rw-r--r--   1 sean  staff   472 Feb 22 13:47 sta\
tes.txt
```

Remember you can use the `Q` key to quit `less` and return to the prompt.

Summary

- The pipe (`|`) takes the output of the program on its left side and directs the output to be the input for the program on its right side.

Exercises

1. Use pipes to figure out how many US states contain the word "New."
2. Examine your `~/.bash_history` to try to figure out how many unique commands you've ever used. (You may need to look up how to use the `uniq` and `sort` commands).

Make

Once upon a time there were no web browsers, file browsers, start menus, or search bars. When somebody booted up a computer all they got a was a shell prompt, and all of the work they did started from that prompt. Back then people still loved to share software, but there was always the problem of how software should be installed. The `make` program is the best attempt at solving this problem, and `make`'s elegance has carried it so far that it is still in wide use today. The guiding design goal of `make` is that in order to install some new piece of software one would:

1. Download all of the files required for installation into a directory.
2. `cd` into that directory.

3. Run `make`.

This is accomplished by specifying a file called `makefile`, which describes the relationships between different files and programs. In addition to installing programs, `make` is also useful for creating documents automatically. Let's build up a `makefile` that creates a `readme.txt` file which is automatically populated with some information about our current directory.

Let's start by creating a very basic `makefile` with `nano`:

```
cd ~/Documents/Journal
nano makefile
```

```
draft_journal_entry.txt:
  touch draft_journal_entry.txt
```

The simple `makefile` above shows illustrates a **rule** which has the following general format:

```
[target]: [dependencies...]
  [commands...]
```

In the simple example we created `draft_journal_entry.txt` is the **target**, a file which is created as the result of the **command(s)**. It's very important to note that any commands under a target **must be indented with a `Tab`**. If we don't use `Tabs` to indent the commands then `make` will fail. Let's save and close the `makefile`, then we can run the following in the console:

```
ls
```

```
## makefile
```

Let's use the `make` command with the target we want to be "made" as the only argument:

```
make draft_journal_entry.txt
```

```
## touch draft_journal_entry.txt
```

```
ls
```

```
## draft_journal_entry.txt
## makefile
```

The commands that are indented under our definition of the rule for the `draft_journal_entry.txt` target were executed, so now `draft_journal_entry.txt` exists! Let's try running the same `make` command again:

```
make draft_journal_entry.txt
```

```
## make: 'draft_journal_entry.txt' is up to date.
```

Since the target file already exists no action is taken, and instead we're informed that the rule for `draft_journal_entry.txt` is "up to date" (there's nothing to be done).

If we look at the general rule format we previously sketched out, we can see that we didn't specify any dependencies for this rule. A **dependency** is a file that the target depends on in order to be built. If a dependency has been updated since the last time `make` was run for a target then the target is not "up to date." This means that the commands for that target will be run the next time `make` is run for that target. This way, the changes to the dependency are incorperated into the target. The commands are only run when the dependencies or change, or when the target doesn't exist at all, in order to avoid running commands unnecessarily.

Let's update our `makefile` to include a `readme.txt` that is built automatically. First, let's add a table of contents for our journal:

```
echo "1. 2017-06-15-In-Boston" > toc.txt
```

Now let's update our `makefile` with `nano` to automatically generate a `readme.txt`:

```
nano makefile
```

```
draft_journal_entry.txt:
  touch draft_journal_entry.txt

readme.txt: toc.txt
  echo "This journal contains the following numbe\
r of entries:" > readme.txt
  wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt
```

Take note that the `-o` flag provided to `egrep` above extracts the regular expression match from the matching line, so that only the number of lines is appended to `readme.txt`. Now let's run `make` with `readme.txt` as the target:

```
make readme.txt
```

```
## echo "This journal contains the following numb\
er of entries:" > readme.txt
## wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt
```

Now let's take a look at `readme.txt`:

```
cat readme.txt
```

```
## This journal contains the following number of \
entries:
## 1
```

Looks like it worked! What do you think will happen if we run `make readme.txt` again?

```
make readme.txt
```

```
## make: 'readme.txt' is up to date.
```

You guessed it: nothing happened! Since the `readme.txt` file still exists and no changes were made to any of the dependencies for `readme.txt` (`toc.txt` is the only dependency) `make` doesn't run the commands for the `readme.txt` rule. Now let's modify `toc.txt` then we'll try running `make` again.

```
echo "2. 2017-06-16-IQSS-Talk" >> toc.txt
make readme.txt
```

```
## echo "This journal contains the following numb\
er of entries:" > readme.txt
## wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt
```

Looks like it ran! Let's check readme.txt to make sure.

```
cat readme.txt
```

```
## This journal contains the following number of \
entries:
## 2
```

It looks like `make` successfully updated `readme.txt`! With every change to `toc.txt`, running `make readme.txt` will *programmatically* update `readme.txt`.

In order to simplify the `make` experience, we can create a rule at the top of our `makefile` called `all` where we can list all of the files that are built by the `makefile`. By adding the `all` target we can simply run `make` without any arguments in order to build all of the targets in the `makefile`. Let's open up `nano` and add this rule:

```
nano makefile
```

```
all: draft_journal_entry.txt readme.txt

draft_journal_entry.txt:
  touch draft_journal_entry.txt

readme.txt: toc.txt
  echo "This journal contains the following numbe\
r of entries:" > readme.txt
  wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt
```

While we have `nano` open let's add another special rule at the end of our `makefile` called `clean` which destroys the files created by our `makefile`:

```
all: draft_journal_entry.txt readme.txt

draft_journal_entry.txt:
  touch draft_journal_entry.txt

readme.txt: toc.txt
  echo "This journal contains the following numbe\
r of entries:" > readme.txt
  wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt

clean:
  rm draft_journal_entry.txt
  rm readme.txt
```

Let's save and close our `makefile` then let's test it out first let's clean up our repository:

```
make clean
ls


## rm draft_journal_entry.txt
## rm readme.txt
## makefile
## toc.txt


make
ls
```

```
## touch draft_journal_entry.txt
## echo "This journal contains the following numb\
er of entries:" > readme.txt
## wc -l toc.txt | egrep -o "[0-9]+" >> readme.txt
## draft_journal_entry.txt
## readme.txt
## makefile
## toc.txt
```

Looks like our `makefile` works! The `make` command is extremely powerful, and this section is meant to just be an introduction. For more in-depth reading about `make` I recommend Karl Broman[8]'s tutorial[9] or Chase Lambert[10]'s makefiletutorial.com[11].

Summary

- `make` is a tool for creating relationships between files and programs, so that files that depend on other files can be automatically rebuilt.
- `makefiles` are text files that contain a list of rules.
- Rules are made up of targets (files to be built), commands (a list of bash commands that build the target), and dependencies (files that the target depends on to be built).

[8] https://twitter.com/kwbroman
[9] http://kbroman.org/minimal_make/
[10] http://chaselambda.com
[11] http://makefiletutorial.com

Bash Programming

> Communities begin by building their kitchen. - French proverb

The last two chapters have discussed how to use the bash shell. Bash itself is a little programming language, and this chapter we're going to discuss how you can write your own computer programs in Bash. Programming in Bash is useful to know because of how seamlessly it integrates with all of the command line programs you've already learned. By the end of this chapter you should be able to write your own command line tools!

You can use `nano` to write all of the programs we're going to discuss in this chapter, however I recommend using the Atom[12] text editor because it's more user friendly.

Now let's create a new file called `math.sh` in the `~/Code/` directory and let's open that file with either `nano` or Atom.

```
cd ~/Code/
nano math.sh
```

You should now have a new, clean text file open. Any code block in the following chapters that starts with the code (or similar) below should indicate to you that we're working on a particular text file.

[12] https://atom.io/

```
#!/usr/bin/env bash
# File: math.sh
```

You do not need to add these lines to your file, though you should type exactly what I have typed below these lines. **Note**: please type all of the lines out for every program that we're going to write, do not copy and paste. Typing code is a little different from typing an email, and you should practice typing the code out yourself as much as possible. Both of these lines start with the pound symbol (`#`) and in the Bash programming language anything that is typed after a pound symbol is ignored (unless the pound symbol is between curly brackets (`{ }`), but that's only in very specific situations). The pound symbol allows you to make **comments** in your code which you can use to annotate code so that another human being who is reading your code can understand how your program is designed to function.

If you're using `nano` or another shell-based text editor you should perhaps open up two terminals, one where you can edit and save the programs you're working on, and one where you can run your programs. One advantage of using Atom is that you can keep Atom open in a separate window and then run your programs in your terminal window.

Math

The Bash programming language can do very basic arithmetic, which we'll demonstrate in this section. Now that you have `math.sh` open in your preferred text editor type the following into your text editor:

```
#!/usr/bin/env bash
# File: math.sh

expr 5 + 2
expr 5 - 2
expr 5 \* 2
expr 5 / 2
```

Save `math.sh` and then run this script in your shell:

```
bash math.sh
```

```
## 7
## 3
## 10
## 2
```

Let's break down what's going on in the Bash script you just created. Bash executes programs in order from the fist line in your file to the last line. The `expr` command can be used to **evaluate** Bash **expressions**. An expression is just a valid string of Bash code that, when run, produces a result. The arithmetic operators that you're already familiar with for addition (+), subtraction (-), and multiplication (*) work like you would expect them to. Notice that when doing multiplication you need to escape the star character, otherwise Bash thinks you're trying to create a regular expression! The division operator (/) does not work as you might expect it to since `5 / 2` = 2.5. Bash does **integer division**, which means that the result of dividing one number by another is always rounded down to the nearest integer. Let's take a look at a few examples on the command line:

```
expr 1 / 3
expr 10 / 3
expr 40 / 21
expr 40 / 20
```

```
## 0
## 3
## 1
## 2
```

The other numerical operator you should be aware of that you might not be familiar with is the modulus operator (%). The modulus operator returns the **remainder** after integer division. In integer division if A / B = C, and A % B = D, then B * C + D = A. Let's take a look at some examples on the command line:

```
expr 1 % 3
expr 10 % 3
expr 40 % 21
expr 40 % 20
```

```
## 1
## 1
## 19
## 0
```

Notice that when one number is completely divisible by another number then the result of the modulus is zero.

If you want to do more complex math, for example math with fractions and numbers with decimals then I highly suggest combining `echo` and the **b**ench **c**alculator program called `bc`. Open up a new file called `bigmath.sh` and type in the following:

```
#!/usr/bin/env bash
# File: bigmath.sh

echo "22 / 7" | bc -l
echo "4.2 * 9.15" | bc -l
echo "(6.5 / 0.5) + (6 * 2.2)" | bc -l
```

Save `bigmath.sh` and then run this script in your shell:

```
bash bigmath.sh
```

```
## 3.14285714285714285714
## 38.430
## 26.2
```

You can pipe any mathematical string to `bc` with the `-l` flag in order to use decimal numbers in your calculations.

Summary

- Bash programs are executed in order from the first line in a file until the last line.
- Anything written after a pound sign (`#`) is a comment and is not executed by Bash.
- You can do simple arithmetic with the `expr` command.
- Perform more complicated arithmetic by piping a string expression into `bc` using `echo`.

Exercises

1. Look at the `man` pages for `bc`.
2. Try doing some math in `bc` interactively.
3. Try writing some equations in a file and then provide that file as an argument to `bc`.

Variables

In Bash you can store data in variables. In chapter 4 we discussed environmental variables that are set by your operating system. You can also create your own variables. Make sure you follow these rules when you're naming variables:

- Every character should be lowercase.
- The variable name should start with a letter.
- The name should only contain alphanumeric characters and underscores (_).
- Words in the name should be separated by underscores.

If you follow those rules then you can avoid accidentally overwriting data stored in environmental variables.

You can assign data to a variable using the equals sign (=). The data you store in a variable can either be a string or a number. Let's create a variable now on the command line:

```
chapter_number=5
```

The variable name is on the left hand side of the equals sign, and the data which will be stored in that variable is on the right hand side of the equals sign. Notice that there are no spaces on either side of the equals sign, this is not allowed when assigning variables:

```
chapter_number = 5


## Error in running command bash
```

In order to print the data in a variable, also called the value of a variable, we can use `echo`. When you want to retrieve the value of a variable you must use the dollar sign (`$`) before the name of the variable. Let's try this out:

```
echo $chapter_number


## 5
```

You can modify the value of a variable using arithmetic operators by using the `let` command:

```
let chapter_number=$chapter_number+1
echo $chapter_number


## 6
```

You can also store strings in variables:

```
the_empire_state="New York"
echo $the_empire_state
```

```
## New York
```

Occasionally you might want to run a command like you would on the command line and store the result of that command in a variable. We can do this by wrapping the command in a dollar sign and parentheses (`$( )`) around a command. This syntax is called **command substitution**. The command is executed and then gets replaced by the string that resulted from running the command. For example if we wanted to store the number of lines in `math.sh`:

```
math_lines=$(cat math.sh | wc -l)
echo $math_lines
```

```
## 7
```

Variable names with a dollar sign can also be used inside other strings in order to insert the value of the variable into the string:

```
echo "I went to school in $the_empire_state."
```

```
## I went to school in New York.
```

When writing a Bash script, the script gives you a few variables for free. Let's create a new file called `vars.sh` with the following code:

```
#!/usr/bin/env bash
# File: vars.sh

echo "Script arguments: $@"
echo "First arg: $1. Second arg: $2."
echo "Number of arguments: $#"
```

Now let's try running the script a few times in a few different ways:

```
bash vars.sh
```

```
## Script arguments:
## First arg: . Second arg: .
## Number of arguments: 0
```

```
bash vars.sh red
```

```
## Script arguments: red
## First arg: red. Second arg: .
## Number of arguments: 1
```

```
bash vars.sh red blue
```

```
## Script arguments: red blue
## First arg: red. Second arg: blue.
## Number of arguments: 2
```

```
bash vars.sh red blue green
```

```
## Script arguments: red blue green
## First arg: red. Second arg: blue.
## Number of arguments: 3
```

Your script can accept arguments just like a command line program! The first argument to your script is stored in `$1`, the second argument is stored in `$2`, etc, etc. An array of all of the arguments passed to your script is stored in `$@`, and we'll discuss how to handle arrays later on in this chapter. The total number of arguments passed to your script is stored in `$#`. Now that you know how to pass arguments to your scripts you can start writing your own command line tools!

Summary

- Variables can be assigned with the equal sign (=) operator.
- Strings or numbers can be assigned to variables.
- The value of a variable can be accessed with the dollar sign ($) before the variable name.
- You can use the dollar sign and parentheses syntax (command substitution) to execute a command and save the output in a variable.
- You can access command line arguments within your own scripts using the dollar sign followed by the number of the argument.

Exercises

1. Write a Bash program where you assign two numbers to different variables, and then the program prints the sum of those variables.
2. Write another Bash program where you assign two strings to different variables, and then the program prints both of those strings. Write a version where the strings are printed on the same line, and a version where the strings are printed on different lines.
3. Write a Bash program that prints the number of arguments provided to that program multiplied by the first argument provided to the program.

User Input

If you're making Bash programs for you or for others to use one way you can get user input is to specify arguments for users to provide to your program, as we discussed in the previous section. You could also ask users to type in a string on the command line by temporarily stopping the execution of your program using the `read` command. Let's a write a small script where you can see how the read command works:

```
#!/usr/bin/env bash
# File: letsread.sh

echo "Type in a string and then press Enter:"
read response
echo "You entered: $response"
```

Now let's run this script:

```
bash letsread.sh
```

```
## Type in a string and then press Enter:
##
```

Let's type `Hello!` into the console, then press enter:

```
## Type in a string and then press Enter:
## Hello!
## You entered: Hello!
```

The `read` command prompts the user to type in a string, and the string that the user provides is stored in the variable that is given to the `read` command in the script.

Summary

- `read` stores a string that the user provides in a variable.

Exercises

1. Write a script that asks the user for an adjective, a noun, and a verb, and then use those words in a sentence (like Mad Libs[13]).

[13] https://en.wikipedia.org/wiki/Mad_Libs

Logic and If/Else

Conditional Execution

When writing computer programs it is often useful for your program to be able to make decisions based on inputs like arguments, files, and environmental variables. Bash provides mechanisms for creating **logical expressions** which resemble mathematical equations. These logical expressions can be evaluated until they are either true or false. In fact, `true` and `false` are both simple Bash commands! Let's try them both out now:

```
true
false
```

At first it doesn't look like they do much. In order to see how they work, we're going to need to look under the hood of Unix a little bit. Whenever you execute a program on the command line, in general one of two things will happen: either the command is executed successfully, or there's an error. In terms of errors there are many ways that a program can go wrong, and Unix can take different actions depending on what kind of error occurs. For example if I enter the name of a command that does not exist into the terminal, then I'll see an error:

```
this_command_does_not_exist
```

```
## Error in running command bash
```

Since that command does not exist, it creates a specific kind of error which is indicated by the program's **exit status**. The exit status of a program is an integer which indicates whether the program was executed successfully or if an error occurred. The exit status of the last program run is stored in the question mark variable (`$?`). We can take a look at the exit status of the last program with `echo`:

```
echo $?
```

```
## 127
```

This particular exit status made an indication to the shell that it should print an error message to the console. What's the exit status of a program that runs successfully? Let's take a look:

```
echo I will succeed.
echo $?
```

```
## I will succeed.
## 0
```

So the exit status of a successful program is `0`. Now let's take a look at the exit statuses of `true` and `false`:

```
true
echo $?
false
echo $?

## 0
## 1
```

As you can see `true` has an exit status of 0 and `false` has an exit status of 1. Since these programs don't do much else, you could define `true` as a program that always has an exit status of 0 and `false` as a program that always has an exit status of 1.

Knowing the exit status of these programs is important when discussing the **logical operators**: the AND operator (`&&`) and the OR operator (`||`). The AND and OR operators can be used for conditional execution of programs on the command line. Conditional execution occurs when the execution of one program depends on the exit status of another program. For example in the case of the AND operator, the program on the right hand side of `&&` will only be executed if the program on the left hand side of `&&` has an exit status of 0. Let's take a look at some small examples:

```
true && echo "Program 1 was executed."
false && echo "Program 2 was executed."

## Program 1 was executed.
```

Since `false` has an exit status of 1, the program `echo "Program 2 was executed."` is not executed, so nothing is printed to the console for that command. Several AND operators can be chained together like so:

```
false && true && echo Hello
echo 1 && false && echo 3
echo Athos && echo Porthos && echo Aramis
```

```
## 1
## Athos
## Porthos
## Aramis
```

In a series of programs joined together by AND operators, any programs to the right of a program that has a non-zero exit status is not executed.

The OR operator (`||`) follows a similar set of principles. Commands on the right hand side of `||` are only executed if the command on the left hand side *fails* and therefore has an exit status other than 0. Let's take a look at how this works:

```
true || echo "Program 1 was executed."
false || echo "Program 2 was executed."
```

```
## Program 2 was executed.
```

Only `echo "Program 2 was executed."` runs because `false` has a non-zero exit status. You can combine multiple OR operators so that only the first program with an exit status of 0 is executed:

```
false || echo 1 || echo 2
echo 3 || false || echo 4
echo Athos || echo Porthos || echo Aramis
```

```
## 1
## 3
## Athos
```

You can combine AND and OR operators in commands, which are evaluated from left to right:

```
echo Athos || echo Porthos && echo Aramis
echo Gaspar && echo Balthasar || echo Melchior
```

```
## Athos
## Aramis
## Gaspar
## Balthasar
```

By combining AND and OR operators you can precisely control the conditions for when certain commands should be executed.

Conditional Expressions

Enabling your Bash script to make decisions is extremely useful. Conditional execution allows you to control the circumstances where certain programs are executed based on whether those programs succeed or fail, but you can also

construct **conditional expressions** which are logical statements that are either equivalent to `true` or `false`. Conditional expressions either compare two values, or they ask a question about one value. Conditional expressions are always between double brackets (`[[ ]]`), and they either use **logical flags** or **logical operators**. For example, there are several logical flags you could use for comparing two integers. If we wanted to see if one integer was greater than another we could use `-gt`, the **g**reater **t**han flag. Enter this simple conditional expression into the command line:

```
[[ 4 -gt 3 ]]
```

The logical expression above is asking: Is 4 greater than 3? No result is printed to the console so let's check the exit status of that expression.

```
echo $?
```

```
## 0
```

It looks like the exit status of this program is 0, the same exit status as `true`. This conditional expression is saying that `[[ 4 -gt 3 ]]` is equivalent to `true`, which of course we know is logically consistent, 4 is in fact greater than 3! Let's see what happens if we flip the expression around so we're asking if 3 is greater than 4:

```
[[ 3 -gt 4 ]]
```

Again, nothing is printed to the console so we'll look at the exit status:

```
echo $?


## 1
```

Ah-ha! Obviously `3` is not greater than `4`, so this false logical expression resulted in an exit status of `1`, which is the same exit status as `false`! Because they have the same exit status `[[ 3 -gt 4 ]]` and `false` are essentially equivalent. To quickly test the logical value of a conditional expression, we can use the AND and OR operators so that an expression will print "t" if it's true and "f" if its false:

```
[[ 4 -gt 3 ]] && echo t || echo f
[[ 3 -gt 4 ]] && echo t || echo f


## t
## f
```

This is a little trick you can use to quickly look at the resulting value of a logical expression.

These **binary** logical expressions compare two values, but there are also **unary** logical expressions that only look at one value. For example, you can test whether or not a file exists using the `-e` logical flag. Let's take a look at this flag in action:

```
cd ~/Code
[[ -e math.sh ]] && echo t || echo f
```

```
## t
```

As you can see the file `math.sh` exists! Most of the time when you're writing bash scripts you won't be comparing two raw values or trying to find something out about one raw value, instead you'll want to create a logical statement about a value contained in a variable. Variables behave just like raw values in logical expressions. Let's take a look at a few examples:

```
number=7
[[ $number -gt 3 ]] && echo t || echo f
[[ $number -gt 10 ]] && echo t || echo f
[[ -e $number ]] && echo t || echo f


## t
## f
## f
```

As you can see 7 is greater than 3 though it is not greater than 10, and there is not file in this directory called 7. There are several other varieties of logical flags, and you can find a table of several of these flags below.

Logical Flag	Meaning	Usage
-gt	**G**reater **T**han	`[[ $planets -gt 8 ]]`
-ge	**G**reater Than or **E**qual To	`[[ $votes -ge 270 ]]`
-eq	**Eq**ual	`[[ $fingers -eq 10 ]]`
-ne	**N**ot **E**qual	`[[ $pages -ne 0 ]]`
-le	**L**ess Than or **E**qual To	`[[ $candles -le 9 ]]`
-lt	**L**ess **T**han	`[[ $wives -lt 2 ]]`
-e	A File **E**xists	`[[ -e $taxes_2016 ]]`
-d	A **D**irectory Exists	`[[ -d $photos ]]`
-z	Length of String is **Z**ero	`[[ -z $name ]]`
-n	Length of String is **N**on-Zero	`[[ -n $name ]]`

Try using each of these flags on the command line before moving on to the next section.

In addition to logical flags there are also logical operators. One of the most useful logical operators is the regex match operator `=~`. The regex match operator compares a string to a regular expression and if the string is a match for the regex then the expression is equivalent to `true`, otherwise it's equivalent to `false`. Let's test this operator a couple different ways:

```
[[ rhythms =~ [aeiou] ]] && echo t || echo f
my_name=sean
[[ $my_name =~ ^s.+n$ ]] && echo t || echo f
```

```
## f
## t
```

There's also the NOT operator `!`, which inverts the value of any conditional expression. The NOT operator turns true expressions into false expressions and vice-versa. Let's take a look at a few examples using the NOT operator:

```
[[ 7 -gt 2 ]] && echo t || echo f
[[ ! 7 -gt 2 ]] && echo t || echo f
[[ 6 -ne 3 ]] && echo t || echo f
[[ ! 6 -ne 3 ]] && echo t || echo f
```

```
## t
## f
## t
## f
```

Here's a table of some of the useful logical operators in case you need to reference how they're used later:

Logical Operator	Meaning	Usage
=~	Matches Regular Expression	`[[ $consonants =~ [aeiou] ]]`
=	String Equal To	`[[ $password = "pegasus" ]]`
!=	String Not Equal To	`[[ $fruit != "banana" ]]`
!	Not	`[[ ! "apple" =~ ^b ]]`

If and Else

Conditional expressions are powerful because you can use them to control how a Bash program that you're writing is executed. One of the fundamental constructs in Bash programming is the **IF statement**. Code written inside of an IF statement is only executed *if* a certain condition is true, otherwise the code is skipped. Let's write a small program with an IF statement:

```
#!/usr/bin/env bash
# File: simpleif.sh

echo "Start program"

if [[ $1 -eq 4 ]]
then
  echo "You entered $1"
fi

echo "End program"
```

First this program will print "Start program", then the IF statement will check if the conditional expression `[[ $1 -eq 4 ]]` is true. It will only be true if you provide `4` as the first argument to the script. If the conditional expression if true then it will execute the code in between `then` and `fi`, otherwise it will skip over that code. Finally the program will print "End program."

Let's try running this Bash program a few different ways. First we'll run this program with no arguments:

```
bash simpleif.sh
```

```
## Start program
## End program
```

Since we didn't provide any arguments to `simpleif.sh` the code within the IF statement was skipped! Now let's try providing an argument to this script:

```
bash simpleif.sh 77
```

```
## Start program
## End program
```

We provided the argument 77, however 77 is not equal to 4, therefore the code within the IF statement was once again skipped. Finally let's provide 4 as an argument:

```
bash simpleif.sh 4
```

```
## Start program
## You entered 4
## End program
```

It worked! Since the first argument to this script was 4, and 4 is equal to 4, the code within the IF statement was executed. You can pair IF statements with ELSE statements. An ELSE statement only runs if the conditional expression being evaluated by the IF statement is false. Let's create a simple program that uses an ELSE statement:

```
#!/usr/bin/env bash
# File: simpleifelse.sh

echo "Start program"

if [[ $1 -eq 4 ]]
then
  echo "Thanks for entering $1"
else
  echo "You entered: $1, not what I was looking f\
or."
fi

echo "End program"
```

Now let's try running this program a few different ways:

```
bash simpleifelse.sh 4
```

```
## Start program
## Thanks for entering 4
## End program
```

The conditional expression `[[ $1 -eq 4 ]]` was true so code inside of the IF statement was run and the code in the ELSE statement was not run. What do you think will happened when we make the conditional expression false?

```
bash simpleifelse.sh 3
```

```
## Start program
## You entered: 3, not what I was looking for.
## End program
```

The conditional expression `[[ $1 -eq 4 ]]` was false so code inside of the ELSE statement was run and the code in the IF statement was not run.

Between IF and ELSE statements you can also have ELIF statements. These statements act like IF statements except they're only evaluated if preceding IF and ELIF statements have all evaluated false conditional expressions. Let's create a brief program using ELIF:

```
#!/usr/bin/env bash
# File: simpleelif.sh

if [[ $1 -eq 4 ]]
then
  echo "$1 is my favorite number"
elif [[ $1 -gt 3 ]]
then
  echo "$1 is a great number"
else
  echo "You entered: $1, not what I was looking f\
or."
fi
```

First let's run the program with `4` as the first argument:

```
bash simpleelif.sh 4
```

```
## 4 is my favorite number
```

The condition in the IF statement was true, so only the first `echo` command was executed. Now let's run the program with `5` as the first argument:

```
bash simpleelif.sh 5
```

```
## 5 is a great number
```

If first condition is false since `5` is not equal to `4`, but then the next condition in the ELIF statement is true since `5` is greater than `3`, so that echo command is executed and the rest of the statement is skipped. Try to guess what will happen if we use `2` as an argument:

```
bash simpleelif.sh 2
```

```
## You entered: 2, not what I was looking for.
```

Since `2` is neither equal to `4` nor greater than `5`, the code in the ELSE statement is executed.

You should also know that you can combine conditional execution, conditional expressions, and IF/ELIF/ELSE statements. The conditional execution operators AND (`&&`) and OR (`||`) can be used in an IF or ELIF statement. Let's look at an example using these operators in an IF statement:

```
#!/usr/bin/env bash
# File: condexif.sh

if [[ $1 -gt 3 ]] && [[ $1 -lt 7 ]]
then
  echo "$1 is between 3 and 7"
elif [[ $1 =~ "Jeff" ]] || [[ $1 =~ "Roger" ]] ||\
 [[ $1 =~ "Brian" ]]
then
  echo "$1 works in the Data Science Lab"
else
  echo "You entered: $1, not what I was looking f\
or."
fi
```

Now let's test this script with a few different arguments:

```
bash condexif.sh 2
bash condexif.sh 4
bash condexif.sh 6
bash condexif.sh Jeff
bash condexif.sh Brian
bash condexif.sh Sean


## You entered: 2, not what I was looking for.
## 4 is between 3 and 7
## 6 is between 3 and 7
## Jeff works in the Data Science Lab
## Brian works in the Data Science Lab
## You entered: Sean, not what I was looking for.
```

The conditional execution operators work just like they would on the command line. If the entire conditional expression evaluates to the equivalent of `true` then the code within the IF statement is executed, otherwise it is skipped.

Finally we should note that IF/ELIF/ELSE statements can be nested inside of other IF statements. Here's a small example of a program with nested statements:

```
#!/usr/bin/env bash
# File: nested.sh

if [[ $1 -gt 3 ]] && [[ $1 -lt 7 ]]
then
  if [[ $1 -eq 4 ]]
  then
          echo "four"
  elif [[ $1 -eq 5 ]]
  then
          echo "five"
  else
          echo "six"
  fi
else
  echo "You entered: $1, not what I was looking f\
or."
fi
```

Now let's run it a few times:

```
bash nested.sh 2
bash nested.sh 4
bash nested.sh 6


## You entered: 2, not what I was looking for.
## four
## six
```

In order to get to the inner IF statement, the conditions for the outer IF statement must be met first (the first argument for the script must be between 3 and 7). As you can see combining variables, arguments, conditional expressions, and IF statements allow you to write more powerful Bash programs.

Summary

- All Bash programs have an exit status. `true` has an exit status of 0 and `false` has an exit status of 1.
- Conditional execution uses two operators: AND (`&&`) and OR (`||`) which you can use to control what command get executed based on their exit status.
- Conditional expressions are always in double brackets (`[[ ]]`). They have exit an exit status of 0 if they contain a true assertion or 1 if they contain a false assertion.
- IF statements evaluate conditional expressions. If an expression is true then the code within an IF statement is executed, otherwise it is skipped.
- ELIF and ELSE statements also help control the flow of a Bash program, and IF statements can be nested within other IF statements.

Exercises

1. Write a Bash script that takes a string as an argument and prints "how proper" if the string starts with a capital letter.
2. Write a Bash script that takes one argument and prints "even" if the first argument is an even number or "odd" if the first argument is an odd number.
3. Write a Bash script that takes two arguments. If both arguments are numbers, print their sum, otherwise just print both arguments.
4. Write a Bash script that prints "Thank Moses it's Friday" if today is Friday. (Hint: take a look at the `date` program).

Arrays

Arrays in Bash are ordered lists of values. You can create a list from scratch by assigning it to a variable name. Lists are created with parentheses (`( )`) with a space separating each element in the list. Let's make a list of the plagues of Egypt:

```
plagues=(blood frogs lice flies sickness boils ha\
il locusts darkness death)
```

To retrieve the array you need to use **parameter expansion**, which involves the dollar sign and curly brackets (`${ }`). The positions of the elements in the array are numbered starting from zero. To get the first element of this array use `${plagues[0]}` like so:

```
echo ${plagues[0]}


## blood
```

Notice that the first element has an **index** of `0`. You can get any of the elements this way, for example the fourth element:

```
echo ${plagues[3]}


## flies
```

To get all of the elements of `plagues` use a star (`*`) between the square brackets:

```
echo ${plagues[*]}


## blood frogs lice flies sickness boils hail loc\
usts darkness death
```

You can also change an individual elements in the array by specifying their index with square brackets:

```
echo ${plagues[*]}
plagues[4]=disease
echo ${plagues[*]}
```

```
## blood frogs lice flies sickness boils hail loc\
usts darkness death
## blood frogs lice flies disease boils hail locu\
sts darkness death
```

To get only part of an array you have to specify the index you would like to start at, followed by the number of elements you would like to retrieve from the array, separated by colons:

```
echo ${plagues[*]:5:3}
```

```
## boils hail locusts
```

The above query essentially says: get 3 array elements starting from the sixth element of the array (remember, the sixth element has an index of 5).

You can find the length of an array using the pound sign (#):

```
echo ${#plagues[*]}
```

```
## 10
```

You can use the plus-equals operator (+=) to add an array onto the end of an array array:

```
dwarfs=(grumpy sleepy sneezy doc)
echo ${dwarfs[*]}
dwarfs+=(bashful dopey happy)
echo ${dwarfs[*]}


## grumpy sleepy sneezy doc
## grumpy sleepy sneezy doc bashful dopey happy
```

Summary

- Arrays are a linear data structure with ordered elements which can be stored in variables.
- The each element of an array has an index and the first index is 0.
- Individual elements of an array can be accessed using their index.

Exercises

1. Write a bash script where you define an array inside of the script, and the first argument for the script indicates the index of the array element that is printed to the console when the script is run.
2. Write a bash script where you define two arrays inside of the script, and the sum of the lengths of the arrays are printed to the console when the script is run.

Braces

Bash has a very handy tool for creating strings out of sequences called **brace expansion**. Brace expansion uses the curly brackets and two periods ({ .. }) to create a sequence of letters or numbers. For example to create a string with all of the numbers between zero and nine you could do the following:

```
echo {0..9}
```

```
## 0 1 2 3 4 5 6 7 8 9
```

In addition to numbers you can also create sequences of letters:

```
echo {a..e}
echo {W..Z}
```

```
## a b c d e
## W X Y Z
```

You can put strings on either side of the curly brackets and they'll be "pasted" onto the corresponding end of the sequence:

```
echo a{0..4}
echo b{0..4}c
```

```
## a0 a1 a2 a3 a4
## b0c b1c b2c b3c b4c
```

You can also combine sequences so that two or more sequences are pasted together:

```
echo {1..3}{A..C}
```

```
## 1A 1B 1C 2A 2B 2C 3A 3B 3C
```

If you want to use variables in order to define a sequence you need to use the `eval` command in order to create the sequence:

```
start=4
end=9
echo {$start..$end}
eval echo {$start..$end}
```

```
## {4..9}
## 4 5 6 7 8 9
```

You can combine sequences with a comma between brackets (`{,}`):

```
echo {{1..3},{a..c}}

## 1 2 3 a b c
```

In fact you can do this with any number of strings:

```
echo {Who,What,Why,When,How}?

## Who? What? Why? When? How?
```

Summary

- Braces allow you create string sequences and expansions.
- To use variables with braces you need to use the `eval` command.

Exercises

1. Create 100 text files using brace expansion.

Loops

`for`

Loops are one of the most important programming structures in the Bash language. All of the programs we've written so far are executed from the first line of the script until the last line, but loops allow you to repeat lines of code based on logical conditions or by following a sequence. The first kind of loop that we'll discuss is a FOR loop. **FOR loops** iterate through every element of a sequence that you specify. Let's take a look at a small example FOR loop:

```
#!/usr/bin/env bash
# File: forloop.sh

echo "Before Loop"

for i in {1..3}
do
        echo "i is equal to $i"
done

echo "After Loop"
```

Now let's execute this script:

```
bash forloop.sh
```

```
## Before Loop
## i is equal to 1
## i is equal to 2
## i is equal to 3
## After Loop
```

Let's walk through `forloop.sh` line-by-line. First `"Before Loop"` is printed before the FOR loop, then the loop begins. FOR loops start with the syntax `for [variable name] in [sequence]` followed by `do` on the next line. The variable name that you define immediately after `for` will take on a value inside of the loop that corresponds to an element in the sequence you provide after `in`, starting with the first element of the sequence, followed by every subsequent element. Valid sequences include brace expansions, explicit lists of strings,

arrays, and command substitutions. In this instance we're using the brace expansion `{1..3}` which we know expands to the string `"1 2 3"`. The code executed in each iteration of the loop is written between `do` and `done`. In the first iteration of the loop, the variable `$i` contains the value `1`. The string `"i is equal to 1"` is printed to the console. There are more elements in the brace expansion after `1`, so after reaching `done` the first time, the program starts executing back at the `do` statement. The second time through the loop variable `$i` contains the value `2`. The string `"i is equal to 2"` is printed to the console, then the loop goes back to the `do` statement since there are still elements left in the sequence. The `$i` variable is now equal to `3`, so `"i is equal to 3"` is printed to the console. There are no elements left in the sequence, so the program moves beyond the FOR loop and finally prints `"After Loop"`. Stop for a moment and edit this loop yourself. Try changing the brace expansion to include other sequences of numbers, letters, or words, then execute the modified code. Before you execute your modified program, write down what you think will be printed. How do the results of executing your program compare with your expectations?

Once you've experimented a little take a look at this example with several other kinds of sequence generating strategies:

```
#!/usr/bin/env bash
# File: manyloops.sh

echo "Explicit list:"

for picture in img001.jpg img002.jpg img451.jpg
do
        echo "picture is equal to $picture"
done

echo ""
echo "Array:"

stooges=(curly larry moe)

for stooge in ${stooges[*]}
do
        echo "Current stooge: $stooge"
done

echo ""
echo "Command substitution:"

for code in $(ls)
do
        echo "$code is a bash script"
done


bash manyloops.sh
```

```
## Explicit list:
## picture is equal to img001.jpg
## picture is equal to img002.jpg
## picture is equal to img451.jpg
##
## Array:
## Current stooge: curly
## Current stooge: larry
## Current stooge: moe
##
## Command substitution:
## bigmath.sh is a bash script
## condexif.sh is a bash script
## forloop.sh is a bash script
## letsread.sh is a bash script
## manyloops.sh is a bash script
## math.sh is a bash script
## nested.sh is a bash script
## simpleelif.sh is a bash script
## simpleif.sh is a bash script
## simpleifelse.sh is a bash script
## vars.sh is a bash script
```

The example above illustrates three other methods of creating sequences for FOR loops: typing out an explicit list, using an array, and getting the result of a command substitution. In each case a variable name is declared after the `for`, and the value of tha variable changes through each iteration of the loop until the corresponding sequence has been exhausted. Right now you should take a moment to write a few FOR loops yourself, generating sequences in all of the ways that we've gone over, just to reinforce your understanding of how a FOR loop works. Loops and conditional statements are two

of the most important structures that we have at our disposal as programmers.

`while`

Now that we've gotten a few FOR loops working let's move on to WHILE loops. The **WHILE loop** is truly the Reese's Peanut Butter Cup[14] of programming structures, combining parts of the FOR loop and the IF statement. Let's take a look at an example WHILE loop so you can see what I mean:

```
#!/usr/bin/env bash
# File: whileloop.sh

count=3

while [[ $count -gt 0 ]]
do
  echo "count is equal to $count"
  let count=$count-1
done
```

The WHILE loop begins first with the `while` keyword followed by a conditional expression. As long as the conditional expression is equivalent to `true` when an iteration of the loop begins, then the code within the WHILE loop will continue to be executed. Based on the code for `whileloop.sh` what do you think will be printed to the console when we run this script? Let's find out:

[14] https://youtu.be/O7oD_oX-Gio

```
bash whileloop.sh
```

```
## count is equal to 3
## count is equal to 2
## count is equal to 1
```

Before the WHILE the `count` variable is set to be 3, but then each time the WHILE loop is executed `1` is subtracted from the value of `count`. The loop then starts from the top again and the conditional expression is re-checked to see if it's still equivalent to `true`. After three iterations through the loop `count` is equal to `0` since `1` is subtracted from `count` in every iteration. Therefore the logical expression `[[ $count -gt 0 ]]` is no longer equal to `true` and the loop ends. By changing the value of the variable in the logical expression inside of the loop we're able to ensure that the logical expression will eventually be equivalent to `false`, and therefore the loop will eventually end.

If the logical expression is never equivalent to `false` then we've created an *infinite loop*, so the loop never ends and the program runs forever. Obviously we would like for our programs to end eventually, and therefore creating infinite loops is undesirable. However let's create an infinite loop so we know what to do if we get into a situation where our program won't terminate. With a simple "typo" we can change the program above so that it runs forever but substituting the minus sign `-` with a plus sign `+` so that `count` is always greater than zero (and growing) after every iteration.

```
#!/usr/bin/env bash
# File: foreverloop.sh

count=3

while [[ $count -gt 0 ]]
do
  echo "count is equal to $count"
  let count=$count+1              # We only chang\
ed this line!
done



## ...
## count is equal to 29026
## count is equal to 29027
## count is equal to 29028
## count is equal to 29029
## count is equal to 29030
## ...
```

If the program is working, then `count` is being incremented very rapidly and you're watching number wiz by in your terminal! Don't fret, you can terminate any program that's stuck in an infinite loop using `Control + C`. Use `Control + C` to get the prompt back so that we can continue.

When constructing WHILE loops, make absolutely sure that you've structured the program so that the loop will terminate! If the logical expression after `while` never becomes `false` then the program will run forever, which is probably not the kind of behavior you were planning for your program.

Nesting

Just like IF statements `for` and `while` loops can be nested within each other. In the example below a FOR loop is nested inside of another FOR loop.

```
#!/usr/bin/env bash
# File: nestedloops.sh

for number in {1..3}
do
  for letter in a b
  do
    echo "number is $number, letter is $letter"
  done
done
```

Based on what we know about FOR loops try to predict what this program will print out before we run the program. Now that you've written down or typed out your prediction let's run it.

```
bash nestedloops.sh
```

```
## number is 1, letter is a
## number is 1, letter is b
## number is 2, letter is a
## number is 2, letter is b
## number is 3, letter is a
## number is 3, letter is b
```

Let's closely examine what's going on here. The outer most FOR loop starts iterating through the sequence generated by `{1..3}`. On the first pass through the loop, the inner loop iterates through the sequence `a b` which first prints `number is 1, letter is a` followed by `number is 1, letter is b`. The first iteration of the outer loop is then finished and the whole process starts over with `number` having a value of `2`. This process continues going through the inner loop until the sequence for the outer loop is exhausted. I again strongly encourage you to pause for a moment and write some of your own nested loops based on the code above. Try to predict what your nested loop program will print before you run your program. If the printed result does not match your prediction trace your way through the program and try to figure out why. Don't just limit yourself to nested FOR loops, use nested WHILE loops, or FOR and WHILE loops in nested combinations.

Besides nesting loops within each other you can also nest loops within IF statements and IF statements within loops. Let's take a look at an example:

```
#!/usr/bin/env bash
# File: ifloop.sh

for number in {1..10}
do
  if [[ $number -lt 3 ]] || [[ $number -gt 8 ]]
  then
    echo $number
  fi
done
```

Before we run this example try once more to guess what the output will be.

```
bash ifloop.sh
```

```
## 1
## 2
## 9
## 10
```

For each iteration of the loop above, the value of `number` was checked in the IF statement, and the `echo` command was only run if `number` was outside the range from `3` to `8`.

There are endless combinations for nesting IF statements and loops, but one good rule of thumb you should remember is that your nesting should never go more than two or possibly three layers deep. If you find yourself writing code with lots of nesting, you should consider restructuring your program. Deeply nested code is difficult to read and even more difficult to debug if your program contains mistakes.

Summary

- Loops allows you repeat sections of your program.
- FOR loops iterate through a sequence so that a variable that you assign takes on the value of every element of the sequence in every iteration of the loop.
- WHILE loops check a conditional statement at the beginning of every iteration. If the condition is equivalent to `true` then one iteration of the loop is executed and then the conditional statement is checked again. Otherwise the loop ends.
- IF statements and loops can be nested in order to make more powerful programming structures.

Exercises

- Write serval programs with three levels of nesting and include FOR loops, WHILE loops, and IF statements. Before you run your program try to predict what your program is going to print. If the result is different from your prediction try to figure out why.
- Enter the `yes` command into the console, then stop the program from running. Take a look at the `man` page for `yes` to learn more about the program.

Functions

Writing Functions

A function is a small piece of code that has a name. Writing functions allows us to re-use the same code multiple times across programs. Functions have the the following syntax:

```
function [name of function] {
  # code here
}
```

Pretty simple, right? Let's open up a new file called `hello.sh` so we can write our first simple function.

```
#!/usr/bin/env bash
# File: hello.sh

function hello {
  echo "Hello"
}

hello
hello
hello
```

The entire structure of the function including the `function` keyword, the name of the function, and the code for the function written inside of the brackets serves as the **function definition**. The function definition assigns the code within the function to the name of the function (`hello` in this case). After a function is defined it can be used like any other command. Using our `hello` command three times should be the equivalent of using `echo "Hello"` three times. Let's run this script to find out:

```
bash hello.sh
```

```
## Hello
## Hello
## Hello
```

It looks like this function works exactly like we expected.

Functions share lots of their behavior with individual bash scripts including how they handle arguments. The usual bash script arguments like `$1`, `$2`, and `$@` all work within a function, which allows you to specify function arguments. Let's create a slightly modified version of `hello.sh` which we'll call `ntmy.sh`:

```
#!/usr/bin/env bash
# File: ntmy.sh

function ntmy {
  echo "Nice to meet you $1"
}
```

In the file above notice that we're not using the `ntmy` function after we've defined it. That's because we're going to start using the functions that we define as command line programs. So far in this chapter we've been using the syntax of `bash [name of script]` in order to execute the contents of a script. Now we're going to start using the `source` command, which allows us to use function definitions in bash scripts as command line commands. Let's use `source` with this file so that we can then use the `ntmy` command:

```
source ntmy.sh
ntmy Jeff
ntmy Philip
ntmy Jenny


## Nice to meet you Jeff
## Nice to meet you Philip
## Nice to meet you Jenny
```

And just like that you've created your very own command! Once you close your current shell you'll lose access to the `ntmy` command, but in the next section we'll discuss how to set up your own commands so that you always have access to them.

Let's write a more complicated function. Imagine that we wanted to add up a sequence of numbers from the command line, but we had no way of knowing how many numbers would be in the sequence. What components would we need to write this function? First we would need a way to capture a list of arguments which can have variable length, second we would need a way to iterate through that list so we could add up each element, and we would need a way to store the cumulative sum of the sequence. These three requirements can be satisfied by using the `$@` variable, a FOR loop, and variable where we can store the sum. It's important to break down a larger goal into a series of individual components before writing a program, that way we more easily can identify which features and tools will be required. Let's write this program in a file called `addseq.sh`.

```
#!/usr/bin/env bash
# File: addseq.sh

function addseq {
  sum=0

  for element in $@
  do
    let sum=sum+$element
  done

  echo $sum
}
```

In the program above we initialize the `sum` variable to be `0` so that we can add other values in the sequence to `sum`. We then use a FOR loop to iterate through every element of `$@`, which is an array of all the arguments we provide to `addseq`. Finally we `echo` the value of `sum`. Let's `source` this program and test it out:

```
source addseq.sh
addseq 12 90 3
addseq 0 1 1 2 3 5 8 13
addseq
addseq 4 6 6 6 4
```

```
## 105
## 33
## 0
## 26
```

By breaking down a large problem we were able to write a nice little function!

Getting Values from Functions

Functions are used for two primary purposes: *computing values* and *side effects*. In the `addseq` command in the previous section we provide the command with a sequence of numbers and then the command provides us with the sum of the sequence which is a value that we're interested in. In this case we can see that `addseq` has computed a value based on a few input values. Many other commands, like `pwd` for example, return a value without affecting the state of the file on our computer. There are however functions like `mv` or `cp` which move and copy files on our computer. A side effect occurs whenever a function creates or changes files on our computer. These commands don't print any value if they succeed.

We'll often write functions in order to calculate some value, and it's important to understand how to store the result of a function in a variable so that it can be used later. Let's `source addseq.sh` and run it one more time:

```
source addseq.sh
addseq 3 0 0 7
```

```
## 10
```

If we look back at the code for `addseq.sh` we can see that we created a variable in the function called `sum`. When you create variables in functions those variables become **globally accessible**, meaning that even after the program is finished that variable retains its value in your shell. We can easily verify this by `echo`ing the value of `sum`:

```
echo $sum
```

```
## 10
```

This is an example of one strategy we can use to retrieve values that a function has calculated. Unfortunately this approach is problematic because it changes the values of variables that we might be using in our shell. For example if we were storing some other important value in a variable called `sum` we would destroy that value by accident by running `addseq`. In order to avoid this problem it's important that we use the `local` keyword when assigning variables within a function. The `local` keyword ensures that variables outside of our function are not overwritten by our function. Let's create a new version of `addseq` called `addseq2` which uses `local` when assigning variables.

```
#!/usr/bin/env bash
# File: addseq2.sh

function addseq2 {
  local sum=0

  for element in $@
  do
    let sum=sum+$element
  done

  echo $sum
}
```

Now let's `source` both files so we demonstrate how `local` helps us avoid overwriting variables.

```
source addseq.sh
source addseq2.sh
sum=4444
addseq 5 10 15 20
echo $sum


## 50
## 50
```

Our original `addseq` overwrites the value we assigned to `sum`. Now let's try `addseq2`.

```
sum=4444
addseq2 5 10 15 20
echo $sum


## 50
## 4444
```

By using `local` within our function the value of `sum` is preserved! In order to correctly capture the value of the result of `addseq2` we can use command substitution.

```
my_sum=$(addseq2 5 10 15 20)
echo $my_sum


## 50
```

Summary

- Functions start with the `function` keyword followed by the name of the function and curly brackets (`{}`).
- Functions are small, reusable pieces of code that behave just like commands.
- You can use variables like `$1`, `$2`, and `$@` in order to provide arguments to functions, just like a Bash script.
- Use the `source` command in order to read in a Bash script with function definitions so that you can use your functions in your shell.
- Use the `local` keyword to prevent your function from creating or modifying global variables.
- Be sure to `echo` the results of your function (if there are any) so that they can be captured with command substitution.

Exercises

Below this list of exercises you can find examples of how these programs should work when used on the command line.

1. Write a function called `plier` which multiplies together a sequence of numbers.
2. Write a function called `isiteven` that prints `1` if a number is even or `0` a number is not even.
3. Write a function called `nevens` which prints the number of even numbers when provided with a sequence of numbers. Use `isiteven` when writing this function.
4. Write a function called `howodd` which prints the percentage of odd numbers in a sequence of numbers. Use `nevens` when writing this function.
5. Write a function called `fib` which prints the number of fibonacci[15] numbers specified.

```
plier 7 2 3
```

```
## 42
```

```
isiteven 42
```

```
## 1
```

[15] https://en.wikipedia.org/wiki/Fibonacci_number

```
nevens 42 6 7 9 33
```

```
## 2
```

```
howodd 42 6 7 9 33
```

```
## .40
```

```
fib 4
```

```
## 0 1 1 2
```

```
fib 10
```

```
## 0 1 1 2 3 5 8 13 21 34
```

Writing Programs

The Unix Philosophy

Perhaps there are some design patters that you've been noticing since we started talking about Unix tools, and now we're going to discuss them explicitly. Unix tools were designed along a set of guidelines which are best summarized by Ken Thompson[16]'s idea that each Unix program should **do one thing well**. Following this rule when writing functions and programs accomplished several goals:

- Limiting a program to only doing one thing reduces the length of the program, and the shorter a program is the easier it is to fix if it contains bugs or if it needs to be revised.
- Writing short programs also helps the users of your code understand what's going on in your code in the event that they need to read your code. Reading a poem induces a different cognitive load compared to reading a novel.
- Folks who don't read the source code of your program (most users won't - they shouldn't have to) will be able to understand the inputs, outputs, and side effects of your program more easily.
- Using small programs to write a new program will increase the likelihood that the new program will also be small. **Composability** is the concept of stringing small programs together to create a new program.

[16] https://en.wikipedia.org/wiki/Ken_Thompson

The concept of composability in Unix is best illustrated by the use of the pipe operator (`|`) for creating pipelines of programs. When you're considering what inputs your program is going to have and what your program is going to print to the console you should consider whether or not your program might be used in a pipeline, and you should organize your program accordingly.

In the previous section we discussed the difference between functions that compute values and functions that produce side effects. You should notice that the side effect functions like `mv` and `cp` do not print any text to the console if they are successful. The concept of *quietness* is another important part of the Unix philosophy. Quietness in this case means that a function should not print to the console unless it is necessary, either to inform the user of a value (`pwd`), to display the result of a computation (`bc`), or to warn the user that an error has occurred.

Making Programs Executable

Let's take a detailed look at some of the code files in our current working directory:

```
ls -l | head -n 3
```

```
## -rw-rw-r-- 1 sean sean 138 Jun 26 12:51 addseq\
.sh
## -rw-rw-r-- 1 sean sean 146 Jun 26 14:45 addseq\
2.sh
## -rw-rw-r-- 1 sean sean 140 Jan 29 10:06 bigmat\
h.sh
```

The left column of this table contains a series of individual characters and dashes. The first hyphen (`-`) signifies that each of the entries in this list are files. If any of them were directories then instead of a hyphen there would be a `d`. Excluding the first hyphen we have the following string: `rw-rw-r--`. This string reflects the **permissions** that are set up for this file. There are three permissions that we can grant: the ability to **read** the file (`r`), **write** to or edit the file (`w`), or **execute** the file (`x`) as a program. These three permissions can be granted on three different levels of access which correspond to each of the three sets of `rwx` in the permissions string: the owner of the file, the group that the file belongs to, and everyone other than the owner and the members of a group. Since you created the file you are the owner of the file, and you can set the permissions for files that you own using the `chmod` command.

The `chmod` command takes two arguments. The first argument is a string which specifies how we're going to change permissions for a file, and the second argument is the path to the file. The first argument has to be composed in a very specific way. First we can specify which set of users we're going to change permissions for:

Character	Meaning
u	The owner of the file
g	The group that the file belongs to
o	Everyone else
a	Everyone above

We then need to specify whether we're going to add, remove, or set the permission:

Character	Meaning
+	Add permission
-	Remove permission
=	Set permission

Finally we specify what permission we're changing:

Character	Meaning
r	Read a file
w	Write to or edit a file
x	Execute a file

Let's use `echo` to write a very short program which we'll call `short`.

```
echo 'echo "a small program"' > short
```

Normally if we wanted to run `short` we would enter `bash short` into the console. If we make this file executable we would only need to enter `short` into the command line to run the program, just like a command! Let's take a look at the permissions for `short`.

```
ls -l short
```

```
## -rw-r--r--  1 sean  staff  23 Jun 28 09:47 sho\
rt
```

We want to make this file executable and we're the owner of this file since we created it. This means we can combine `u`, `+`, and `x` in order make `short` executable. Let's try it:

```
chmod u+x short
ls -l short
```

```
## -rwxr--r--  1 sean  staff  23 Jun 28 09:47 sho\
rt
```

We successfully added the `x`! To run an executable file we need to specify the path to the file, even if the path is in the current directory, meaning we need to prepend `./` to `short`. Now let's try running the program.

```
./short
```

```
## a small program
```

Looks like it works! There is one small detail we should add to this program though. Even though we've made our file executable, if we give our program to somebody else they might be using a shell that doesn't know how to execute our

program. We need to indicate how the program should be run by adding a special line of text to the beginning of our program called a **shebang**. The shebang always begins with `#!` followed by the path to the program which will execute the code in our file. The shebang for indicating that we want to use Bash is `#!/usr/bin/env bash`, which we've been adding to the start of our scripts for a while now! Let's rewrite this program to include the Bash shebang and then let's run the program.

```
echo '#!/usr/bin/env bash' > short
echo 'echo "a small program"' >> short


## a small program
```

Now our Bash script is ready to go!

Environmental Variables

We're one step away from being able to use our scripts and functions as shell commands, but first we need to learn about environmental variables. An environmental variable is a variable that Bash creates where data about your current computing environment is stored. Environmental variable names use all capitalized letters. Let's look at the values for some of these variables. The `HOME` variable contains the path to our home directory, and the `PWD` variable contains the path to our current directory.

```
echo $HOME
echo $PWD


## /Users/sean
## /Users/sean/Code
```

If we want one of our functions to be available always as a command then we need to change the `PATH` variable. Let's take a look at this variable first.

```
echo $PATH


## /usr/local/bin:/usr/bin:/bin:/usr/local/git/bin
```

The `PATH` variable contains a sequence of paths on our computer separated by colons. When the shell starts it searches these paths for executable files, and then makes those executable commands available in our shell. One approach to making our scripts available is to add a directory to the `PATH`. Bash scripts in the directory that are executable can be used as commands. We need to modify `PATH` every time we start a shell, so we can ammend our `~/.bash_profile` so that our directory for executable scripts is always in the `PATH`. To modify an environmental variable we need to use the `export` keyword.

First let's create a new directory called `Commands` in our `Code` directory where we can keep our executable scripts. Then we'll add a line to our `~/.bash_profile` so that `Commands` is added to the `PATH`.

```
mkdir Commands
nano ~/.bash_profile
```

```
alias docs='cd ~/Documents'
alias edbp='nano ~/.bash_profile'

export PATH=~/Code/Commands:$PATH
```

Save ~/.bash_profile and close nano. Now let's source our Bash profile (we only need to do this once) and move short into the Commands directory. Then we should be able to use short as a command!

```
source ~/.bash_profile
short
```

```
## a small program
```

Looks like it works!

Alternatively to making individual scripts executable we can add a source command to our ~/.bash_profile so that we can use a Bash function on the command line. Let's use nano to open up our ~/.bash_profile again.

```
nano ~/.bash_profile
```

```
alias docs='cd ~/Documents'
alias edbp='nano ~/.bash_profile'

export PATH=~/Code/Commands:$PATH
source ~/Code/addseq2.sh
```

Save the ~/.bash_profile, quit nano, and now let's source our ~/.bash_profile so we can test if we can use addseq2.

```
source ~/.bash_profile
addseq2 9 8 7


## 24
```

Again it works! If you have multiple Bash functions that you'd like to be able to use on the command line then it's a good idea to define these functions in one of a few files so that you don't have to source every individual function that you want to have available.

Summary

- According to the Unix Philosophy you should keep your programs short, simple, and quiet.
- Use chmod to make your programs executable.
- You can modify your ~/.bash_profile in order to make scripts and functions available to use on the command line.
- Use export to change an environmental variable.

Exercises

Below this list of exercises you can find examples of how the programs described here should work when used on the command line.

1. Make a script executable.
2. Put that script in a directory that you create and make that directory part of your `PATH`.
3. Write a program called `range` that takes one number as an argument and prints all of the numbers between that number and 0.
4. Write a program called `extremes` which prints the maximum and minimum values of a sequence of numbers.

```
range 6
```

```
## 0 1 2 3 4 5 6
```

```
range -3
```

```
## -3 -2 -1 0
```

```
extremes 8 2 9 4 0 3
```

```
## 0 9
```

Git and GitHub

Proof rather than argument. - Japanese proverb

What are Git and GitHub?

Git is a command line program which allows you to track versions of any code or plain text documents that you create. Like the "track changes" feature of a word processor Git keeps track of who made particular changes, the time and date of those changes, and where the changes were made. If a critical file gets deleted by accident, or if you make a breaking change to your code and you want to try to figure out where the breaking change was made, you can use Git to restore the deleted file or find the new bug in your program. Git organizes groups of files that you're tracking into a **repository**, which is just a directory where all of the changes to files in that directory are tracked. Git can also help you collaborate with others when you're writing software. As Karl Broman[17] says (paraphrasing Mark Holder[18]): "Your closest collaborator is you six months ago, but you donâ€™t reply to emails."

GitHub is a website that provides remote Git repositories. A remote repository is just a Git repository that you're able to access over an internet connection. GitHub allows you to create public remote repositories for free, and anyone can see

[17] https://twitter.com/kwbroman
[18] https://twitter.com/mtholder

your code in these public repositories. If you want to keep your code private then you can pay GitHub for private remote repositories.

If you're working on code together with a friend GitHub can help you sync changes to code files between you and your friend. There's also a social and community aspect to GitHub, since you can watch other programmers develop their projects. GitHub also makes it easy to jump in and help somebody with their project. GitHub offers many other useful features which we will discuss at length.

Setting Up Git and GitHub

Before setting up Git, go to GitHub[19] and create a free account. Take note of which email address you use and which username you choose.

To see if you have Git installed open up your terminal and enter the following:

```
git --version
```

```
## git version 2.11.0 (Apple Git-81)
```

If you don't get a response back telling you the version of Git that you have installed then you need to install Git. You can find instructions for installing Git on your operating system here[20].

[19] https://github.com/

[20] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

Open up your shell once you have git installed and run `git --version` again to make sure that installation succeeded (you may need to restart your shell or your computer). After Git is installed we need to set up two environmental variables, but we only need to do this once. The first variable we need to set up with Git is your GitHub user name, and the second variable is the email address that you used to create your GitHub account:

```
git config --global user.name "myUserName"
git config --global user.email myName@email.com
```

Getting Started with Git

Let's create our first Git repository. First we need to create a directory:

```
cd
mkdir my-first-repo
cd my-first-repo
```

"Repo" in this case is just shorthand for "repository." To start tracking files with Git in a directory enter `git init` into the command line:

```
git init
```

```
## Initialized empty Git repository in /Users/sea\
n/my-first-repo/.git/
```

You've just created your first repository! Now let's create a file and start tracking it.

```
echo "Welcome to My First Repo" > readme.txt
```

Now that we've created a file in this Git repository, let's use `git status` to see what's going on in this repository. We'll be using `git status` continuously throughout this chapter in order to get information about the status of this Git repository.

```
git status
```

```
## On branch master
##
## Initial commit
##
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         readme.txt
##
## nothing added to commit but untracked files pr\
esent (use "git add" to track)
```

As you can see `readme.txt` is listed as an untracked file. In order to let Git know that you want to track this file we need to use `git add` with the name of the file that we want to track. Let's start tracking `readme.txt`:

```
git add readme.txt
```

Git now knows to track any changes to `readme.txt`. Let's see how the status of the repository has changed:

```
git status
```

```
## On branch master
##
## Initial commit
##
## Changes to be committed:
##   (use "git rm --cached <file>..." to unstage)
##
##         new file:   readme.txt
##
```

Git is now tracking `readme.txt`, or in Git-specific language `readme.txt` is now **staged**. Between the parentheses in the message above you can see that `git status` is giving us a tip about how to unstage (or un-track) this file, which we could do with `git rm --cached readme.txt`. Let's unstage this file just to see what happens:

```
git rm --cached readme.txt
```

```
## rm 'readme.txt'
```

```
git status
```

```
## On branch master
##
## Initial commit
##
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         readme.txt
##
```

Our repository is right back to the way it started with `readme.txt` as an unstaged file. Let's start tracking `readme.txt` again so we can move on to cooler Git features.

```
git add readme.txt
```

Now that Git is tracking `readme.txt` we need to create a milestone to indicate the changes that we made to `readme.txt`. In this case, the changes that we made were creating the file in the first place! This milestone is called a **commit** in Git. A commit logs the content of all of the currently staged files. Right now we only have `readme.txt` staged so let's commit the creation of this file. When making a Git commit, we need to write a commit message which is specified after the `-m` flag. The message should briefly describe what changes you've made since the last commit.

```
git commit -m "added readme.txt"
```

```
## [master (root-commit) 73e53ca] added readme.txt
##  1 file changed, 1 insertion(+)
##  create mode 100644 readme.txt
```

The message above confirms that the commit succeeded and it summarizes the changes that took place since the last commit. As you can see in the message we only changed one file, and we only changed one line in that file. Let's run `git status` again to see the state of our repository after we've made the first commit:

```
git status
```

```
## On branch master
## nothing to commit, working tree clean
```

All of the changes to the files in this repository have been committed! Let's add a few more files to this repository and commit them.

```
touch file1.txt
touch fil2.txt
ls
```

```
## file1.txt
## fil2.txt
## readme.txt
```

While we're at it let's also add a new line of text to `readme.txt`:

```
echo "Learning Git is going well so far." >> read\
me.txt
```

Now that we've added two more files and we've made changes to one file let's take a look at the state of this repository.

```
git status


## On branch master
## Changes not staged for commit:
##   (use "git add <file>..." to update what will\
 be committed)
##   (use "git checkout -- <file>..." to discard \
changes in working directory)
##
##         modified:   readme.txt
##
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         fil2.txt
##         file1.txt
##
## no changes added to commit (use "git add" and/\
or "git commit -a")
```

We can see that Git has detected that one file has been modified, and that there are two files in this directory that it is not tracking. Now we need to tell Git to track the changes to these files. We could tell Git to track changes to each file using `git add`, or since all of the files in this repository are `.txt` files we could use a wildcard and enter `git add *.txt` into the console. However if we want to track all of the changes to all of the files in our directory we should use the command `git add -A`.

```
git add -A
git status
```

```
## On branch master
## Changes to be committed:
##   (use "git reset HEAD <file>..." to unstage)
##
##         new file:   fil2.txt
##         new file:   file1.txt
##         modified:   readme.txt
##
```

Now the changes to all of the files in this repository are being tracked. Finally let's commit these changes:

```
git commit -m "added two files"
```

```
## [master 53a1983] added two files
##  3 files changed, 1 insertion(+)
##  create mode 100644 fil2.txt
##  create mode 100644 file1.txt
```

Darn it, now looking at this commit summary I realize that I have a typo in one of the names of my files! Thankfully we can undo the most recent commit with the command `git reset --soft HEAD~`:

```
git reset --soft HEAD~
git status
```

```
## On branch master
## Changes to be committed:
##   (use "git reset HEAD <file>..." to unstage)
##
##         new file:   fil2.txt
##         new file:   file1.txt
##         modified:   readme.txt
##
```

This repo is now in that exact same state it was right before we made the commit. Now we can rename `fil2.txt` to `file2.txt`, then let's look at the status of the repository again.

```
mv fil2.txt file2.txt
git status
```

```
## On branch master
## Changes to be committed:
##   (use "git reset HEAD <file>..." to unstage)
##
##         new file:   fil2.txt
##         new file:   file1.txt
##         modified:   readme.txt
##
## Changes not staged for commit:
##   (use "git add/rm <file>..." to update what w\
ill be committed)
##   (use "git checkout -- <file>..." to discard \
changes in working directory)
##
##         deleted:    fil2.txt
##
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         file2.txt
##
```

We previously told Git to track `fil2.txt`, and we can see that Git acknowledges that the file has been deleted. We can bring Git up to speed with what files it should be tracking with `git add -A`:

```
git add -A
git status
```

```
## On branch master
## Changes to be committed:
##   (use "git reset HEAD <file>..." to unstage)
##
##         new file:   file1.txt
##         new file:   file2.txt
##         modified:   readme.txt
##
```

Finally we got the file names right! Now let's make the correct commit:

```
git commit -m "added two files"
```

```
## [master 12bb9f5] added two files
##  3 files changed, 1 insertion(+)
##  create mode 100644 file1.txt
##  create mode 100644 file2.txt
```

That looks much better.

Summary

- Git tracks changes to plain text files (code files and text documents).
- A directory where changes to files are tracked by Git is called a Git repository.
- Change your working directory, then run `git init` to start a repository.
- You can track changes to a file using `git add [names of files]`.

- You can create a milestone about the state of your files using `git commit -m "message about changes since the last commit"`.
- To examine the state of files in your repository use `git status`.

Exercises

1. Start a repository in a new directory.
2. Create a new file in your new Git repository. Make sure Git is tracking the file and then create a new commit.
3. Make changes to the file, and then commit these changes.
4. Add two new files to your repository, but only commit one of them. What is the status of your repository after the commit?
5. Undo the last commit, add the untracked file, and redo the commit.

Important Git Features

Gitting Help, Logs, and Diffs

Git commands have their own `man` pages. You can access them with `git help [name of command]`. For example here's the start of the help page for `git status`:

```
git help status
```

```
GIT-STATUS(1)                                    Git \
Manual                                    GIT-STATUS(1\
)

NAME
       git-status - Show the working tree status

SYNOPSIS
       git status [<options>...] [--] [<pathspec>\
...]

DESCRIPTION
       Displays paths that have differences betwe\
en the index file and the current HEAD commit,
       paths that have differences between the wo\
rking tree and the index file, and paths in the
       working tree that are not tracked by Git (\
and are not ignored by gitignore(5)). The first
       are what you would commit by running git c\
ommit; the second and third are what you could
       commit by running git add before running g\
it commit.
```

Just like any other help page that uses `less`, you can return to the prompt with the `Q` key.

If you want to see a list of your Git commits, enter `git log` into the console:

```
git log
```

```
## commit 12bb9f53b10c9b720dac8441e8624370e4e071b6
## Author: seankross <sean@seankross.com>
## Date:   Fri Apr 21 15:23:59 2017 -0400
##
##     added two files
##
## commit 73e53cae75301ce9b2802107b1956447241bb17a
## Author: seankross <sean@seankross.com>
## Date:   Thu Apr 20 14:15:26 2017 -0400
##
##     added readme.txt
```

If you've made many commits to a repository you might need to press the `Q` key in order to get back to the prompt. Each commit has its time, date, and commit message recorded, along with a SHA-1 hash that uniquely identifies the commit.

Git can also help show the differences between unstaged changes to your files compared to the last commit. Let's add a new line of text to `readme.txt`:

```
echo "The third line." >> readme.txt
git diff readme.txt
```

```
## diff --git a/readme.txt b/readme.txt
## index b965f6a..a3db358 100644
## --- a/readme.txt
## +++ b/readme.txt
## @@ -1,2 +1,3 @@
##  Welcome to My First Repo
##  Learning Git is going well so far.
## +I added a line.
```

As you can see a plus sign shows up next to the added line. Now let's open up this file in a text editor so we can delete the second line.

```
nano readme.txt
# Delete the second line
git diff readme.txt
```

```
## diff --git a/readme.txt b/readme.txt
## index b965f6a..e173fdf 100644
## --- a/readme.txt
## +++ b/readme.txt
## @@ -1,2 +1,2 @@
##  Welcome to My First Repo
## -Learning Git is going well so far.
## +I added a line.
```

A minus sign appears next to the line we deleted. Let's take a look at the status of our directory at this point.

```
git status
```

```
## On branch master
## Changes not staged for commit:
##   (use "git add <file>..." to update what will\
 be committed)
##   (use "git checkout -- <file>..." to discard \
changes in working directory)
##
##         modified:   readme.txt
##
## no changes added to commit (use "git add" and/\
or "git commit -a")
```

If you read the results from `git status` carefully you can see that we can take this repository in one of two directions at this point. We can either `git add` the files we've made changes to in order to track those changes, or we can use `git checkout` in order to remove all of the changes we've made to a file to restore its content to what was present in the last commit. Let's remove our changes to see how this works.

```
cat readme.txt
```

```
## Welcome to My First Repo
## I added a line.
```

```
git checkout readme.txt
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
```

And as you can see the changes we made to `readme.txt` have been undone.

Ignoring Files

Sometimes you might have files that you never want Git to track, for example binary files that are generated as by-products of running code (PDFs or images), or secrets like passwords or API keys. A file in your Git repository called `.gitignore` can list names of files and sub-folders, or simple regular expressions (whatever you can use with `ls`) in order to specify files which should never be tracked. Each line of a `.gitignore` file should specify a file or group of files that should not be tracked by Git. Let's make a `.gitignore` file to make sure that we never track image files in this repository:

```
touch toby.jpg
git status
```

```
## On branch master
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         toby.jpg
##
## nothing added to commit but untracked files pr\
esent (use "git add" to track)
```

Now that we've added an image to our repository, let's add a `.gitignore` file to make sure Git doesn't track these kinds of files.

```
echo "*.jpg" > .gitignore
git status
```

```
## On branch master
## Untracked files:
##   (use "git add <file>..." to include in what \
will be committed)
##
##         .gitignore
##
## nothing added to commit but untracked files pr\
esent (use "git add" to track)
```

Now we can see that Git has detected the new `.gitignore` file, but it doesn't see `toby.jpg`. Let's add and commit our `.gitignore` file:

```
git add -A
git commit -m "added gitignore"


## [master adef548] added gitignore
##  1 file changed, 1 insertion(+)
##  create mode 100644 .gitignore
```

Now if we add another `.jpg` file, Git will not see the file:

```
touch bernie.jpg
git status


## On branch master
## nothing to commit, working tree clean


ls


## bernie.jpg
## toby.jpg
## file1.txt
## file2.txt
## readme.txt
```

Summary

- `git help` allows you to read the `man` pages for specific Git commands.
- `git log` will show you your commit history.
- `git diff` displays what has changed between the last commit and your current untracked changes.
- You can specify a `.gitignore` file in order to tell Git not to track certain files.

Exercises

1. Look at the help pages for `git log` and `git diff`.
2. Add to the `.gitignore` you already started to include a specific file name, then add that file to your repository.
3. Create a file that contains the Git log for this repository. Use `grep` to see which day of the week most of the commits occurred on.

Branching

Branching is one of the most powerful features that Git offers. Creating different Git branches allows you to work on a particular feature or set of files independently from other "copies" of a repository. That way you and a friend can work on different parts of the same file on different branches, and then Git can help you elegantly merge your branches and changes together.

You can list all of the available branches with the command `git branch`:

```
git branch
```

```
## * master
```

The star (*) indicates which branch you're currently on. The default branch that is created is always called *master*. Usually people use this branch as the working version of the software that they are writing, while they develop new and potentially unstable features on other branches.

To add a branch we'll also use the `git branch` command, followed the name of the branch we want to create:

```
git branch my-new-feature
```

Now let's enter `git branch` again to confirm that we've created the branch:

```
git branch
```

```
## * master
## my-new-feature
```

We can make `my-new-feature` the current branch using `git checkout` with the name of the branch:

```
git checkout my-new-feature
```

```
## Switched to branch 'my-new-feature'
```

```
git branch
```

```
##   master
## * my-new-feature
```

If we look at `git status` we can also see that it will tell us which branch we're on:

```
git status
```

```
On branch my-new-feature
nothing to commit, working tree clean
```

We can switch back to the `master` branch using `git checkout`:

```
git checkout master
```

```
## Switched to branch 'master'
```

```
git branch
```

```
## * master
##   my-new-feature
```

Now we can delete a branch by using the `-d` flag with `git branch` and the name of the branch we want to delete:

```
git branch -d my-new-feature
```

```
## Deleted branch my-new-feature (was adef548).
```

```
git branch
```

```
## * master
```

Let's create a new branch for adding a section to the `readme.txt` in our repository. We can create a new branch and switch to that branch at the same time using the command `git checkout -b` and the name of the new branch we want to create:

```
git checkout -b update-readme
```

```
## Switched to a new branch 'update-readme'
```

Now that we've created and switched to a new branch, let's make some changes to a file. As you might be expecting right now we'll add a new line to `readme.txt`:

```
echo "I added this line in the update-readme bran\
ch." >> readme.txt
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
```

Now that we've added a new line let's commit these changes:

```
git add -A
git commit -m "added a third line to readme.txt"
```

```
## [update-readme 6e378a9] added a third line to \
readme.txt
##  1 file changed, 1 insertion(+)
```

Now that we've made a commit on the `update-readme` branch, let's switch back to the `master` branch, and then we'll take a look at `readme.txt`:

```
git checkout master
```

```
## Switched to branch 'master'
```

Now that we're on the `master` branch let's quickly glance at `readme.txt`:

```
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
```

The third line that we added is gone! Don't fret, the line that we added isn't gone forever. We committed the change to this file while we were on the `update-readme` branch, so the updated file is safely in that branch. Let's switch back to that branch just to make sure:

```
git checkout update-readme
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
```

And the third line is back! Let's add and commit yet another line while we're on this branch:

```
echo "It's sunny outside today." >> readme.txt
git add -A
git commit -m "added weather info"
```

```
## [update-readme d7946e9] added weather info
##  1 file changed, 1 insertion(+)
```

This is a small example of how to use Git branching, but you can see how you can make incremental edits to plain text (usually code files) without effecting the `master` branch (the tested and working copy of your software) and without effecting any other branches. You can imagine how this system could be used for multiple people to work on the same codebase at the same time, or how you could develop and test multiple software features without them interfering with each other. Now that we've made a couple of changes to `readme.txt`, let's combine those changes with what we have in the `master` branch. This is made possible by a Git **merge**. Merging allows you to elegantly combine the changes that have been made between two branches. Let's merge

the changes we made in the `update-readme` branch with the `master` branch. Git incorporates other branches into the current branch by default. When you're merging, the current branch is also called the **base** branch. Let's switch to the `master` branch so we can merge in the changes from the `update-readme` branch:

```
git checkout master
```

```
## Switched to branch 'master'
```

To merge in the changes from another branch we need to use `git merge` and the name of the branch:

```
git merge update-readme
```

```
## Updating adef548..d7946e9
## Fast-forward
##  readme.txt | 2 ++
##  1 file changed, 2 insertions(+)
```

```
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
## It's sunny outside today.
```

It looks like you've merged your first branch in Git! Branching is part of what makes Git so powerful since it enables parallel developments on the same code base. But what if there are two commits in two separate branches that make different edits to the same line of text? When this occurs it is called a **conflict**. Let's create a conflict so we can learn how they can be resolved.

First we'll switch to the `update-readme` branch. Use `nano` to edit the last line of `readme.txt`, then commit your changes:

```
git checkout update-readme
nano readme.txt
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
## It's cloudy outside today.
```

Notice that we changed "sunny" to "cloudy" in the last line.

```
git add -A
git commit -m "changed sunny to cloudy"
```

Now that our changes are committed on the `update-readme` branch, let's switch back to `master`:

```
git checkout master
```

Let's change the same line of code using `nano`:

```
nano readme.txt
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
## It's windy outside today.
```

Now let's commit these changes:

```
git add -A
git commit -m "changed sunny to windy"
```

We've now created two commits that directly conflict with each other. On the `update-readme` branch the last line says `It's cloudy outside today.`, while on the `master` branch the last line says `It's windy outside today.`. Let's see what happens when we try to merge `update-readme` into `master`.

```
git merge update-readme
```

```
## Auto-merging readme.txt
## CONFLICT (content): Merge conflict in readme.t\
xt
## Automatic merge failed; fix conflicts and then\
 commit the result.
```

Uh-oh, there's a conflict! Let's check the status of the repo right now:

```
git status
```

```
## On branch master
## You have unmerged paths.
##   (fix conflicts and run "git commit")
##   (use "git merge --abort" to abort the merge)
##
## Unmerged paths:
##   (use "git add <file>..." to mark resolution)
##
##         both modified:   readme.txt
##
## no changes added to commit (use "git add" and/\
or "git commit -a")
```

If you're getting used to reading the result of `git status`, you can see that it often offers suggestions about what steps you should take next. Git is indicating that both versions of readme.txt have modified the same text. Let's take a look at `readme.txt` to see what's going on there:

```
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
## <<<<<<< HEAD
## It's windy outside today.
## =======
## It's cloudy outside today.
## >>>>>>> update-readme
```

The first three lines of this file look normal, then things get interesting! The line between `<<<<<<< HEAD` and `=======` shows the version of the conflicted line on the current branch. In Git terminology the `HEAD` represents the most recent commit on the branch which is currently checked out (which is `master` in this case). The line between `=======` and `>>>>>>> update-readme` shows the version of the line on the `update-readme` branch. In order to resolve this conflict, all we need to do is open `readme.txt` with `nano` so we can delete the lines we want to get rid of. In this case let's keep the "cloudy" version.

```
nano readme.txt
cat readme.txt
```

```
## Welcome to My First Repo
## Learning Git is going well so far.
## I added this line in the update-readme branch.
## It's cloudy outside today.
```

Now we can commit the resolution of this conflict.

```
git add -A
git commit -m "resolved conflict"
```

You're now familiar with this basics of Git! If you want to go into further depth with your study of Git I highly recommend the free and open source book Pro Git[21].

Summary

- Git branching allows you and others to work on the same code base together.
- You can create a branch with the command `git branch [name of branch]`.
- To switch to a branch use `git checkout [name of branch]`.
- You can combine a branch with your current branch by using `git merge`.

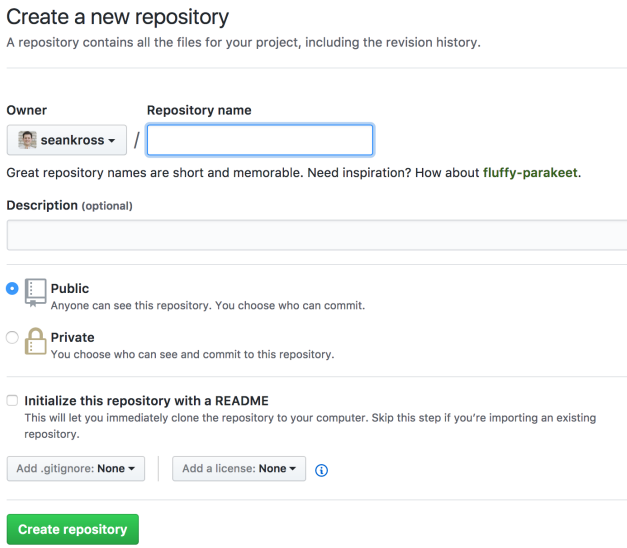
Exercises

1. Start a new branch.
2. Switch to that branch and add commits to it. Switch to an older branch and then merge the new branch into your current branch.
3. Purposefully create and resolve a merge conflict.

[21] https://git-scm.com/book/

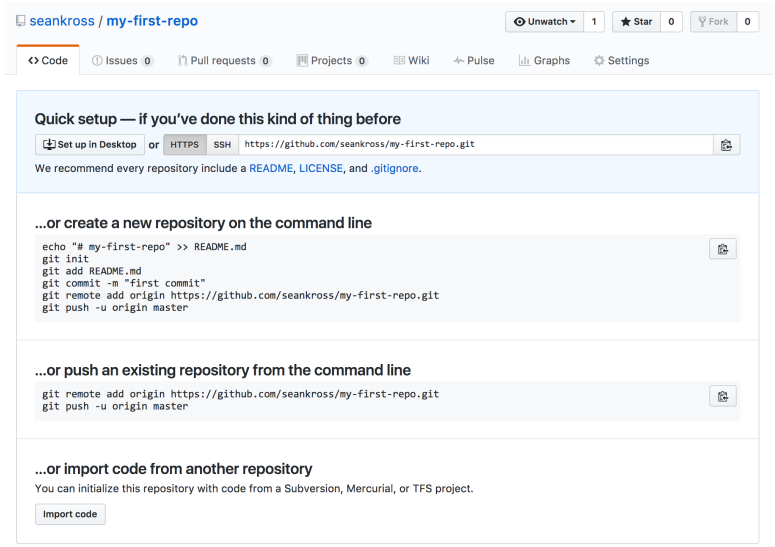
GitHub

Now that you know the basics of using Git, let's talk about how you can share your work and start collaborating online using GitHub. As an added bonus, by the end of this chapter you will have created your very own website! To get started go to GitHub[22] and sign in with the credentials we set up at the beginning of the chapter. After you sign in you should see a plus-sign near the top-right corner of your web browser. Click the plus-sign and a little menu should appear, then click "New repository." You should now see a screen that looks like this:



In the text box under **Repository name** type `my-first-repo` and then click the green **Create repository** button. Now you should see a page like this:

[22] https://github.com/



GitHub offers a few suggestions about what to do with our new remote repository. We've already been using a local Git repository, and what GitHub provides is a **remote** Git repository. A remote Git repository is just a Git repository stored on a computer is that always turned on and connected to the internet, so it can act as a central point where we can share and sync our changes to files with our friends and colleagues. We can see which remote repositories our local repository is connected to with the `git remote` command while we have our working directory set to `my-first-repo`:

```
git remote
```

Nothing is printed to the console since you haven't set any remotes up yet! Now let's add your new GitHub repository as a remote in your local repository:

```
git remote add origin https://github.com/seankros\
s/my-first-repo.git
```

In the command above `git remote add` adds a new remote to your local repository, `origin` is the name we're assigning to this remote repository, and `https://github.com/seankross/my-first-repo.git` is the URL of the remote repository. You should of course substitute `seankross` for your GitHub user name so that it corresponds to your remote repository URL. Later I'll explain why "origin" is the name we chose for this remote. Let's run `git remote` again to confirm that we added the `origin` remote successfully:

```
git remote
```

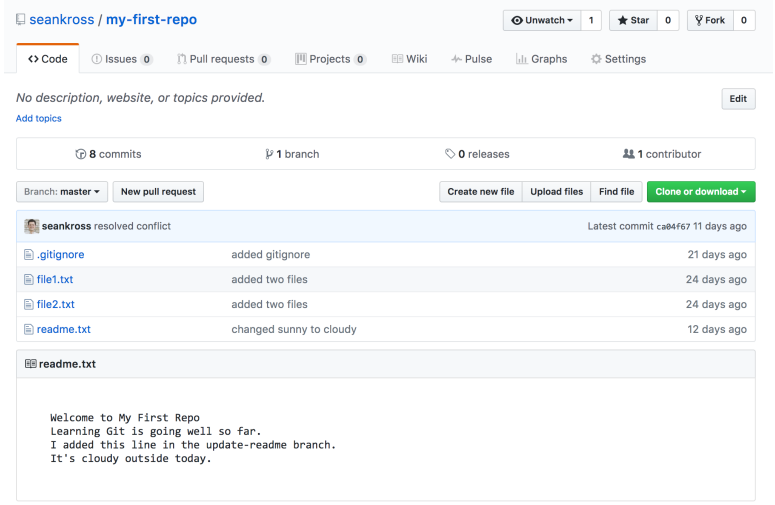
```
## origin
```

Now that we've added our GitHub remote, let's perform our first Git **push**. A Git push updates a remote repository with all of the commits that we've made to our local Git repository. This first Git push you do when setting up a remote on GitHub with a local repository is a little different from future Git pushes. We'll need to use the `-u` flag in order to set `origin` as the default remote repository so we don't have to provide its name every time we want to interact with it. Enter the following command, modified so that you're using your GitHub user name:

```
git push -u origin master
```

```
## Counting objects: 23, done.
## Delta compression using up to 4 threads.
## Compressing objects: 100% (19/19), done.
## Writing objects: 100% (23/23), 1.88 KiB | 0 by\
tes/s, done.
## Total 23 (delta 9), reused 0 (delta 0)
## remote: Resolving deltas: 100% (9/9), done.
## To https://github.com/seankross/my-first-repo.\
git
##  * [new branch]      master -> master
## Branch master set up to track remote branch ma\
ster from origin.
```

The command above pushed all of our commits to the remote repository on GitHub, and it set up the `master` branch of the `origin` remote repository as the default remote repository. Looking back at the web page for your repository on GitHub, it should look something like this:



One neat feature of GitHub is that readme files are rendered on the repository page so you can write documents which explain the contents of your repository. Let's get more creative with these readme documents by learning a small language called Markdown.

Markdown

Markdown is a markup language. Markup languages are sets of rules for adding decorative features to text. The most popular markup language is HTML, but you might have also heard of XML and LaTeX. Markdown is a powerful markup language because it's small, intuitive, and readable when it's written as plain text. GitHub transforms Markdown files (which end in the file extension `.md`) into simple HTML web pages in your repository. If there is a file called `README.md` in any folder in your repository, then that file is rendered to HTML and displayed on GitHub. Let's create a `README.md` file

for our repository. First we'll destroy the plain text readme file we already have:

```
rm readme.txt
```

I've included a Markdown file below that attempts to explain some of Markdown's features. Copy the plain text below, create a new file called `README.md` with `nano`, paste the text in, and then save the file.

```
# This is a large heading

## This is a smaller heading

And as **imagination** bodies forth,
The forms of things *unknown*, the poetâ€™s pen,
Turns them to shapes and gives to airy nothing,
A local *habitation* and a **name**.

- This is
- an unordered
- list

1. This is
2. an ordered
3. list

Here is `some code` in the middle of a sentence.

```
This is
a block
```
```

```
of code
```

Here is how you make [a link](https://www.wikiped\
ia.org/).

![This is an image.](https://github.com/yihui/xar\
ingan/releases/download/v0.0.2/karl-moustache.jpg\
)

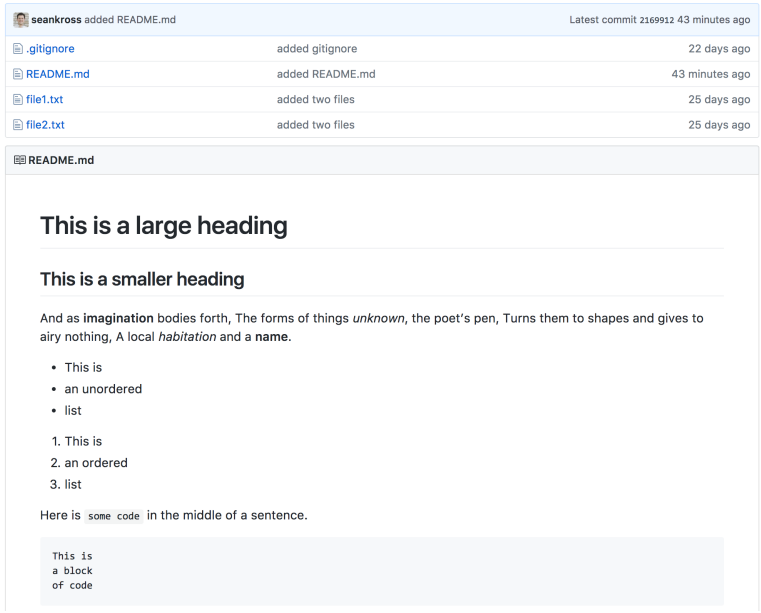
```
nano README.md
```

Now let's add our changes, make a commit, and push those changes to our remote repository:

```
git add -A
git commit -m "added README.md"
git push
```

```
## Counting objects: 3, done.
## Delta compression using up to 4 threads.
## Compressing objects: 100% (3/3), done.
## Writing objects: 100% (3/3), 659 bytes | 0 byt\
es/s, done.
## Total 3 (delta 0), reused 0 (delta 0)
## To https://github.com/seankross/my-first-repo.\
git
##    ca04f67..2169912  master -> master
```

Since we set up a default remote repository the first time we pushed, we can now simply enter `git push` in order to send our latest commits to the `master` branch on the `origin` remote. Now the page on GitHub for your repository should look something like this:



We've got a much more complex readme file! Notice how the plain text that we wrote has been rendered according to a few rules:

- Pound signs (`#`, `##`) make headings.
- A word surrounded by single asterisks (`*word*`) makes that word *italicized*.
- A word surrounded by double asterisks (`**word**`) makes that word **bold**.

- You can create lists with hyphens (-) or numbers (`1.`, `2.`, `3.`).
- Code can be placed in the middle of a line with single backticks (`` `code` ``).
- A code block can be created by putting code in between a set of triple backticks (‘).
- You can insert a link with brackets and parentheses (`[Link text here](http://jhu.edu)`).
- You can use an image with an exclamation point, and a link to an image (`![Alt text here](http://jhu.edu/jeff.jpg)`)

Personally I really enjoy writing with Markdown, to the point where I wrote this entire book[23] in Markdown! We're going to be using Markdown for the rest of this chapter, so I suggest that you take a few minutes to play around with the syntax in this in-browser Markdown editor[24]. For more information about Markdown see GitHub's helpful *Mastering Markdown*[25] guide.

Pull Requests

The next two features of GitHub we're going to discuss - **pull requests** and **forking** - are what make GitHub so great. A pull request allows you to interactively compare two different branches before you merge them so you can either go ahead

[23] https://github.com/seankross/the-unix-workbench/blob/master/docs/06-Git-and-GitHub.md

[24] https://jbt.github.io/markdown-editor/#TVFLbtwwDN3rFCxm0xgTOU133WXXA2QXFDDHYizWEmlIctzZ
LwDPEeuYA8hYZkIImFgmZw7HP6r1YwpUXmvPkkArNB1nHFiwcYqXQcnDUwVXrW0eHTPkfYwV9BXaNGIxlh
+qrCQGGEtUvd8hqZQIy4mhtZq4jeLLIdcziB6ETu6J0g6YoIu4onb3yEueLTJBDN1nXfu/
rqIRSiwipZAhYJ9E9fm3Cd/Qzz6HXIFfPb/EF/tu19iqJoJRg00AMtlg8whJNo3tDORNJKRrOkwDO4qinCyOWdnmJ1
O1jbG2pX/p+2za/8cwLBUavZervTPzDy80agd0B8u+UiVtcT37U3J85rtz/
wGLXQukLJcJKtQ9mQFIM/duDf/CP/Ywl3Wdda8Mxkv++THd/AA==

[25] https://guides.github.com/features/mastering-markdown/

with the merge or provide feedback to whoever opened the pull request. Essentially a pull request allows a person to ask another person if they're willing to incorporate changes on one branch into another branch. This social coding transaction may involve you and a collaborator, you and a stranger, or you might open a pull request on your own repository just as a method of staying organized.

Since I can't guarantee that you have a collaborator I'll show you how to open a pull request on your own repository. First in your local `my-first-repo` repository let's switch over to the `update-readme` branch.

```
git checkout update-readme
```

```
## Switched to branch 'update-readme'
```

Let's take a look at what's currently on this branch:

```
ls
```

```
## bernie.jpg
## toby.jpg
## file1.txt
## file2.txt
## readme.txt
```

It looks like we haven't updated this branch to be current with the `master` branch. We can easily do this by merging in the `master` branch.

```
git merge master
```

```
## Updating 5aa94fa..2169912
## Fast-forward
##  README.md  | 28 ++++++++++++++++++++++++++
##  readme.txt |  4 ----
##  2 files changed, 28 insertions(+), 4 deletion\
s(-)
##  create mode 100644 README.md
##  delete mode 100644 readme.txt
```

Now the `master` and `update-readme` branches are identical. Let's clean up this directory so that you can make a little personalized Markdown project. First let's delete all of the files in this directory that we don't really need, meaning everything except `README.md`.

```
rm *.txt
rm *.jpg
ls
```

```
## README.md
```

Now that we've cleaned up our repository let's open up `README.md` with `nano`. Delete everything that's written there and write a few lines about yourself. In the block of text below you can see what I've written in `README.md`.

```
# Sean Kross

### Geography

I live in the city of Baltimore, in the state of \
Maryland, in the United States
of America.

### Reading

Three of my favorite books are:

- *Mindstorms* by Seymour Papert
- *Welcome to the Monkey House* by Kurt Vonnegut
- *Persepolis* by Marjane Satrapi

### Food

Last night I dreamt about eating in these restaur\
ants:

1. Linger in Denver.
2. Azura in Jerusalem.
3. Gemma in New York City.

### Contact

The best way to get in touch with me is [on Twitt\
er](https://twitter.com/seankross).
```

Once you've written up a few fun things about yourself, add your changes, and make a new commit.

```
git add -A
git commit -m "made readme more personal"
```

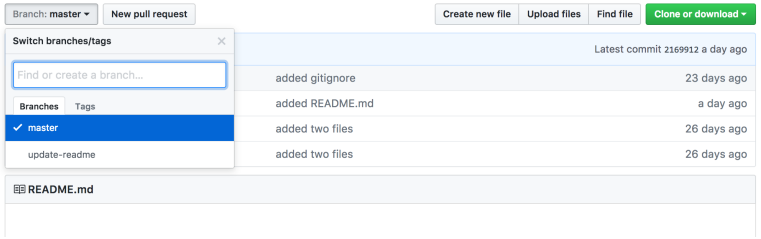
Like a local Git repository, remote repositories on GitHub can have multiple branches. Let's push this commit to the `update-readme` branch on GitHub:

```
git push origin update-readme
```

```
## Counting objects: 3, done.
## Delta compression using up to 4 threads.
## Compressing objects: 100% (3/3), done.
## Writing objects: 100% (3/3), 630 bytes | 0 byt\
es/s, done.
## Total 3 (delta 0), reused 0 (delta 0)
## To https://github.com/seankross/my-first-repo.\
git
##  * [new branch]      update-readme -> update-r\
eadme
```

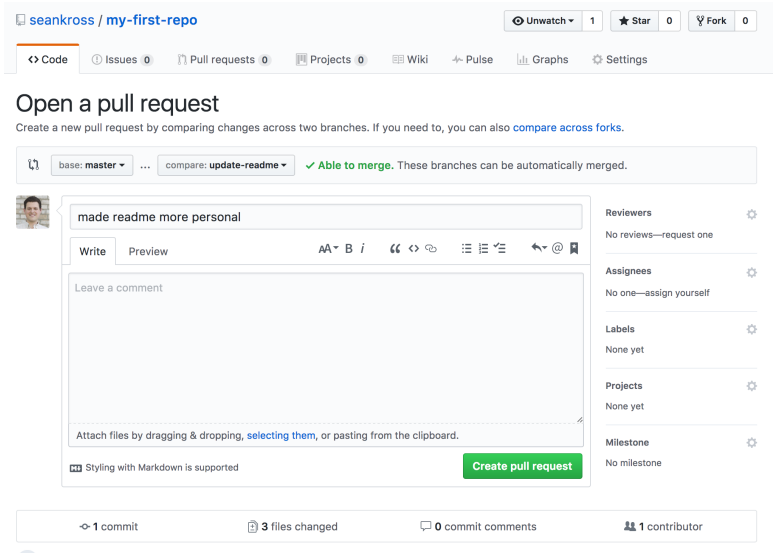
Notice that we needed to specify which remote we were pushing to since GitHub didn't previously know about the existence of the `update-readme` branch. When you perform a `git push`, only the commits on the current branch are sent to the remote repository. That way you can create local branches that cannot be accessed from the remote repository, unless you explicitly push them to GitHub.

Now let's go back to the GitHub page for our repository. On the left side of the page you should see a button that says "Branch: master." Click on that button and a little drop-down menu should appear, as you can see below:



Click "update-readme" in the menu in order to view the files in that branch. You should see that the `README.md` files are different! You can switch back and forth between looking at branches using this menu.

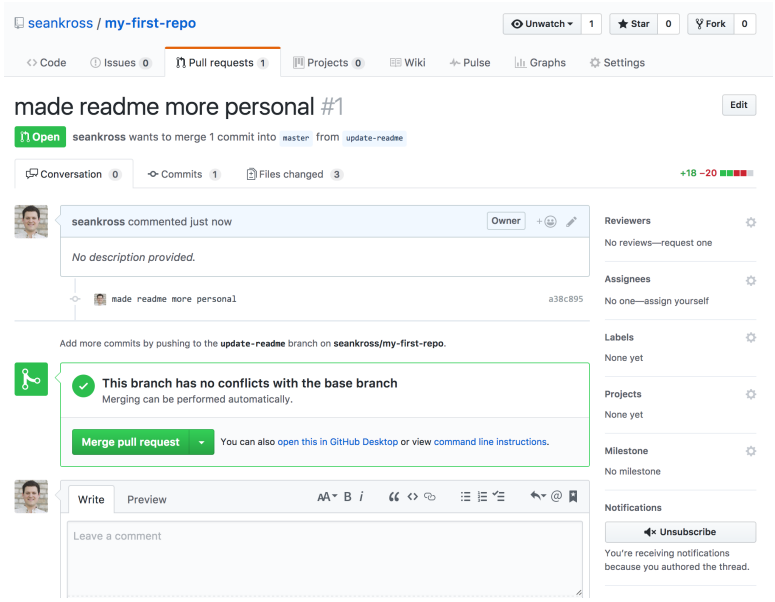
Now that you've pushed an updated branch to GitHub, let's open a pull request. A pull request is like a guided `git merge` that is facilitated by GitHub. To start the pull request click the "New pull request" button next to the branch button (see the upper left corner of the image above). That button should take you to a page like this:



There are a few important details on this page, so let's go through them. First under the "Open a pull request" heading you can see the names of two branches. The branch name after "base:" shows the branch that changes are being merged into (in this case the `master` branch), and the branch name after "compare:" shows the branch that has the changes (in this case the `update-readme` branch).

In the text boxes below you can write a title for your pull request (the default title in this case is the name of the last commit) and you can write comments about the pull request which you can format with Markdown. If you're collaborating with somebody else on a project it's important to write good comments so that your collaborators know what changes you made in the branch you are requesting to merge. If you scroll down the page you can see a line-by-line comparison of the changes in the "compare" branch compared to the "base" branch. When you've finished going over these

changes click the green "Create pull request" button in order to open the pull request. You should now see a screen like this:



Congratulations on opening your first pull request! Let's take a look at what's happening on this page. Below the title of the pull request we can see three tabs called **Conversation**, **Commits**, and **Files changed**. In the **Conversation** tab we can add comments to the pull request which can be formatted with Markdown. The **Commits** tab lists the commits that have been made to the "compare" branch in this pull request. Finally the **Files changed** tab shows the same line-by-line comparison we saw before.

Usually when you're working with collaborators there's a great deal of discussion that occurs after you open a pull request. Git commits that are pushed to the "compare" branch (`update-readme` in the case) of the GitHub repository will be

reflected in a pull request even after the request has been opened. This way changes that are made as a result of the discussion can be easily incorporated. Once you're ready go back to the **Conversation** tab and click the green "Merge pull request" button, then click the green "Confirm merge" button that appears. This will `git merge` the "compare" branch into the "base" branch on our remote repository. You just merged your first pull request! Now click near the top left corner of this page on the **<> Code** tab, and you should see that the changes from the `update-readme` branch have been merged into `master`.

When working on a remote GitHub repository with many other folks theses pull requests and merges can happen without you being involved at all, if the commits effect parts of the code that you're not working on. Still it's important to keep your local repository up to date with the latest changes in the remote repository. Let's go back to your terminal where you have `my-first-repo` set as the current working directory.

When working on a remote GitHub repository with many other folks thess pull requests and merges can happen without you being involved at all if the commits effect parts of the code that you're not working on. Still it's important to keep your local repository up to date with the latest changes in the remote repository. Let's go back to your terminal where you have `my-first-repo` set as the current working directory. First let's switch to the `master` branch.

```
git checkout master
```

Now let's update our local master branch with the commits that have been merged into the master branch on our remote

repository. We can accomplish this with the command `git pull`:

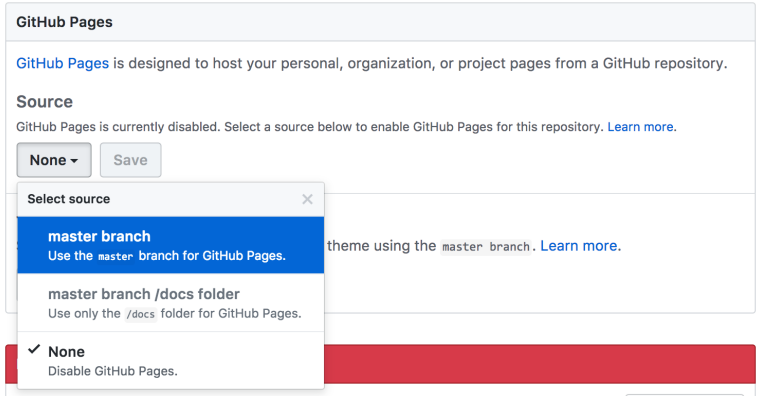
```
git pull
```

```
## remote: Counting objects: 1, done.
## remote: Total 1 (delta 0), reused 0 (delta 0),\
 pack-reused 0
## Unpacking objects: 100% (1/1), done.
## From https://github.com/seankross/my-first-repo
##    2169912..b9217f6  master     -> origin/mast\
er
## Updating 2169912..b9217f6
## Fast-forward
##  README.md | 38 ++++++++++++++++++-----------\
--------
##  file1.txt |  0
##  file2.txt |  0
##  3 files changed, 18 insertions(+), 20 deletio\
ns(-)
##  delete mode 100644 file1.txt
##  delete mode 100644 file2.txt
```

With `git pull` Git finds the `master` branch on the `origin` remote repository and updates our local repository with the new commits. You've now completed the full pull request life cycle! In the **Forking** section of this chapter we'll come back to how discussing how GitHub super-charges pull requests in order to foster a greater coding community.

Pages

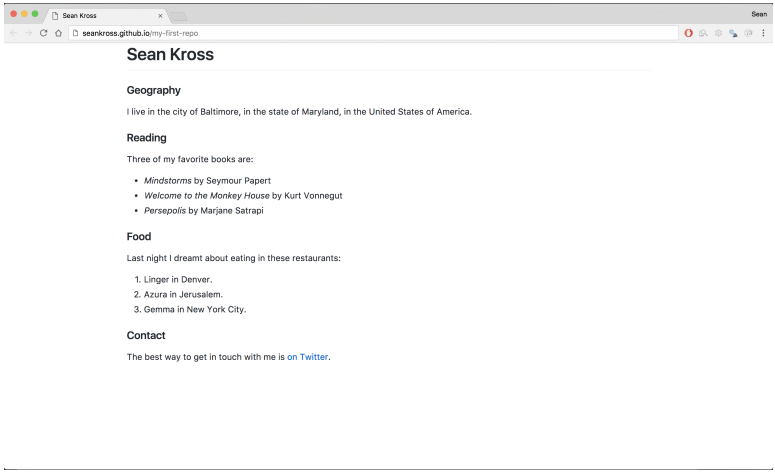
For now we're going to take a little detour to discuss GitHub Pages[26]. GitHub Pages allows you to create and host a website on GitHub using only Git and Markdown. Go back to your `my-first-repo` repository page on GitHub on click the *Settings* tab at the top. Scroll down the page until you see a box that says **GitHub Pages**. Then click on the drop-down menu that says **None**. You should see a screen like this:



Click *master branch* and then click *Save*. Now go to the website [your-github-username].github.io/my-first-repo (in my case the address is seankross.github.io/my-first-repo[27]) and you should see your very own website!

[26] https://pages.github.com/

[27] http://seankross.github.io/my-first-repo



How cool is that!? If you want to change your new website all you need to do is edit your `README.md` then commit and push the changes! Websites like these are great for showing off projects, providing software documentation, creating online resumes, or writing a blog! GitHub pages websites can be as simple as a few Markdown documents, or if you're know some web programming you can turn them into complex websites. For more information about GitHub Pages you can check out the documentation here[28].

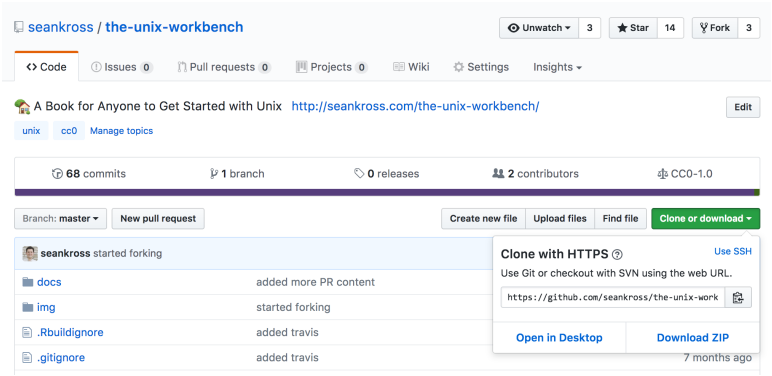
Forking

If you're reading this book it's fairly safe to say that you probably interact with software every day. Software powers your computer, the internet, your phone, and this book. Considering all of the software you use, have you ever thought about modifying that software? Perhaps you could write some code to add a new feature you think would be useful or to

[28] https://pages.github.com/

fix a glitch that you've noticed. GitHub makes modifying other people's software easy through the process of **forking**. Forking a GitHub repository copies somebody else's GitHub repository into your GitHub account. You can then modify this copy of their software however you like. After you've added some commits to your copy of the repository you can keep the new commits to yourself, share them with others, or you can open up a **pull request** for your new commits to be merged into the original source repository. This original source repository (the repository you forked) is often called the *upstream* repository.

Let's try forking a repository now. Go to https://github.com/seankross/the-unix-workbench and click the **Fork** button in the upper right corner. GitHub will then ask you what account you want to use to fork the repository. If you're a new GitHub user then you only have one account, so select that account. After a brief "please wait" screen you should then see the forked repository at the URL https://github.com/[your-github-username]/the-unix-workbench. You've now forked the repository for this book! In order to get get a local copy of the repository you'll need to use the `git clone` command. Cloning a repository copies a Git repository to your computer while keeping track of the remote repository that it originated from. Let's clone your fork of my repository now. On the right side of the repository page you should see a green button that says **Clone or download**. Click that button and the following menu should appear:



Click on the little clipboard icon which will copy the Git URL. Now go back to the terminal and change your working directory to your home directory.

```
cd
pwd
```

```
## /Users/sean
```

Now let's clone the repository. Type `git clone` into the terminal and then paste in the Git URL we copied from GitHub:

```
git clone https://github.com/[your-github-usernam\
e]/the-unix-workbench.git
```

```
## Cloning into 'the-unix-workbench'...
## remote: Counting objects: 669, done.
## remote: Compressing objects: 100% (7/7), done.
## remote: Total 669 (delta 2), reused 8 (delta 2\
), pack-reused 660
## Receiving objects: 100% (669/669), 4.00 MiB | \
4.55 MiB/s, done.
## Resolving deltas: 100% (510/510), done.
```

Now `cd` into your cloned repository.

```
cd the-unix-workbench
```

You've just successfully completed your first Git clone! Like we mentioned before, cloning has the advantage of keeping track of the remote repository that should be associated with the local repository. Let's test this by entering `git remote` with the added `-v` flag:

```
git remote -v
```

```
## origin        https://github.com/[your-github-usernam\
e]/the-unix-workbench.git (fetch)
## origin        https://github.com/[your-github-usernam\
e]/the-unix-workbench.git (push)
```

As you can see the default name of the remote repository after you clone that repository is `origin`. Now that you've cloned your fork you should add a commit! One change that I suggest is adding your name to `guestbook.md`. Let's do this now:

```
echo "- Sean Kross" >> guestbook.md # Add your ow\
n name of course!
cat guestbook.md


## # Guest Book
##
## - Sean Kross
```

Now add, commit, and push your changes:

```
git add guestbook.md
git commit -m "added my name to guestbook.md"
git push
```

Now that you've added your name to the guest book you can merge your change into my version of the guest book by opening up a new pull request as described in the previous section. If your pull request to the guest book is merged into the upstream repository (by me) then you will have completed the full GitHub lifecycle!

The process of forking a repository, making changes, and then opening a pull request is a very powerful workflow for seeing changes that you want made in the software world. Many large and important software projects have repositories on GitHub including operating systems[29], programming languages[30], and even Git itself[31]! If there's a change you want to see in a public GitHub repository, fork that repository and make the change!

[29] https://github.com/torvalds/linux
[30] https://github.com/golang/go
[31] https://github.com/git/git

Summary

- You can use GitHub you create and host remote Git repositories.
- A remote Git repository is a Git repository that is always connected to the internet.
- List remote repositories with `git remote`.
- Add remote repositiories with `git remote add [name-of-remote] https://github.com/[username]/[repo-name].git`
- Add commits to your remote repository with `git push [name-of-remote] [name-of-branch]` or just `git push` if you've set up a default remote and branch.
- To merge commits on a remote repository into your local repository use `git pull [name-of-remote] [name-of-branch]` or just `git pull` if you've set up a default remote and branch.
- A pull request allows you to interactively compare two different branches before you merge them.
- GitHub Pages allows you to host websites written in Markdown for free!
- Forking a repository allows you to make changes to a copy of a public repository. You can then open a pull request if you think your changes should be merged into the upstream repository!

Exercises

1. Create a new repository on GitHub. Clone your repository and add a `README.md` file. Push this file to GitHub and create a GitHub Pages website for this repository.

2. Fork an existing repository (try one of mine https://github.com/seankro and try to identify something valuable you could contribute. Make changes or additions to that repository, then open a pull request.
3. Read through GitHub's Guides[32].

[32] https://guides.github.com/

Nephology

I saw a city in the clouds. - Dagobahnian proverb

Introduction to Cloud Computing

Nephology is the study of clouds. Few modern technology concepts (other than maybe data science and artificial intelligence) have been hyped as loudly as "the cloud." The cloud is simply a computer which we can access over the internet. In this chapter we'll set up a cloud computer and we'll learn the basics of interacting with one.

To get the most out of this chapter you're going to need your credit or debit card, or a PayPal[33] account. We're going to be using DigitalOcean[34], a company which we can rent cloud computers from. Throughout this chapter I might refer to cloud computers as **servers** (computers connected to the internet) or as **droplets**, which is a marketing term DigitalOcean uses to refer to their servers (a droplet is *not* a technical term). Renting from DigitalOcean won't cost you any money since I'm giving you a coupon for two free months of service! There are several companies that offer similar services compared to DigitalOcean, but in my opinion they have the best user interface and the most transparent pricing model.

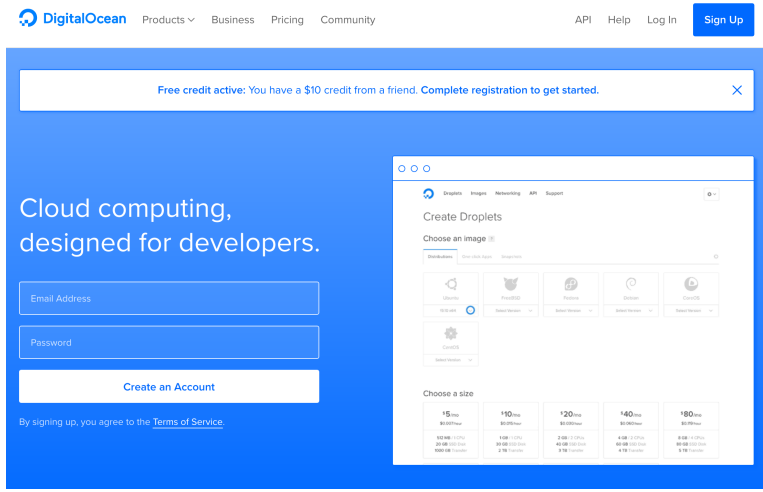
[33] https://www.paypal.com

[34] https://m.do.co/c/530d6cfa2b37

Warning: *At the end of this chapter we will discuss how to shut down any servers we've started on DigitalOcean. If you don't shut down your server after two months then your account will be charged real money for using DigitalOcean. Please be sure to shut down any servers you start after you are finished using them.*

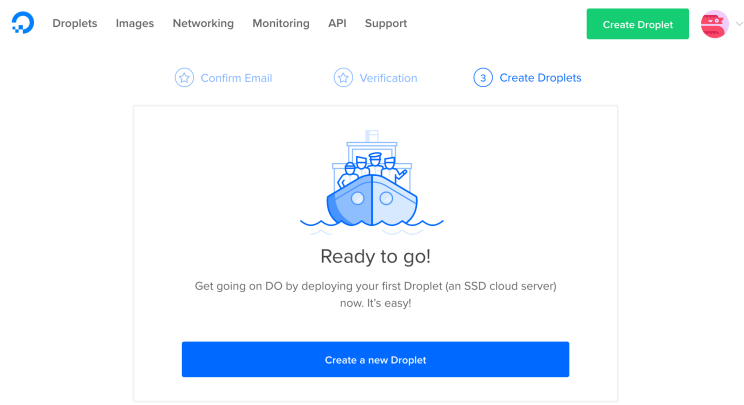
Setting Up DigitalOcean

To get started with DigitalOcean we need to rent a server from their website. Click this link[35] to sign up for DigitalOcean in order to get two free months of server use. (If you don't use this link then you don't get two free months). Click **Sign Up** in the upper right corner, then enter your email address and choose your password.

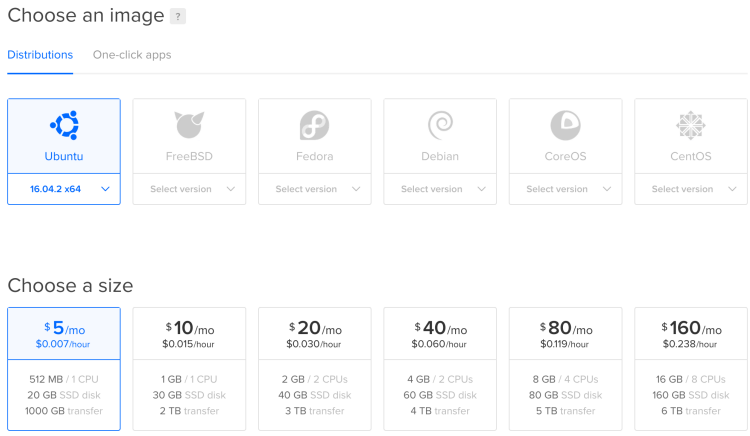


[35] https://m.do.co/c/530d6cfa2b37

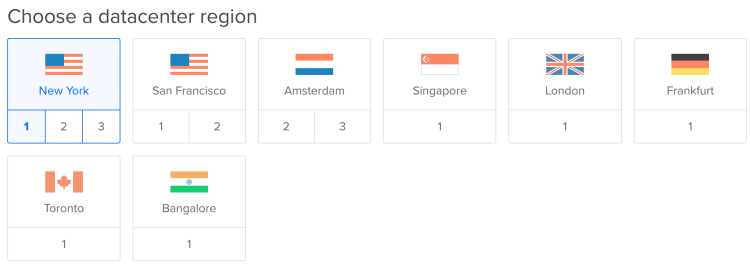
Check your email for a message from DigitalOcean and click the enclosed link to confirm your email. Then you'll need to enter your credit or debit card information, or your Paypal account details. As long as you close down all of your servers less than two months after you start them you will not be charged. After entering in your payment information you should see this screen:



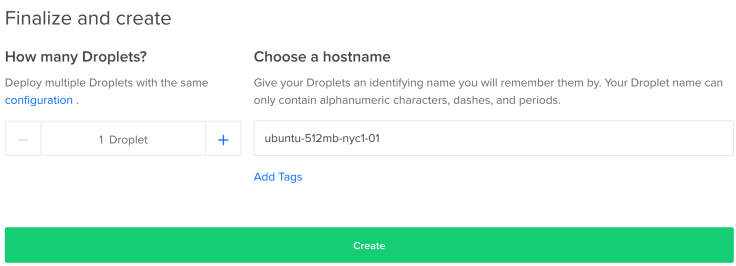
Click the big blue **Create a new Droplet** button which should then bring you to a screen where you can customize the server you're going to be renting. Make sure you have Ubuntu chosen as your distribution, and select the $5 per month size option:



Scroll down the page and then select the region that is geographically closest to you. Some regions have multiple data centers, it doesn't matter which data center you pick. Currently I'm in Baltimore, Maryland, USA, so I'm going to pick the number 1 data center in New York:



Finally at the bottom of the page click the big green **Create** button in order to start your server!



It will take a minute to launch the server, but once launched you should receive an email from Digital Ocean with the details about your new server. Included in this email you should find the IP address of your server, the default username (which should be `root`) and a randomly generated password that you will need to connect to your server for the first time. Once you've received this email open up a new terminal.

Connecting to the Cloud

We can connect to computers on the internet with the `ssh` program, which stands for **S**ecure **Sh**ell. The `ssh` command provides a command line interface to whichever computer we point it to. A computer that is connected to the internet has an address (just like a house has an address) which is specified by an **IP address**. The command for connecting to a computer with `ssh` generally looks like this:

```
ssh [username]@[IP address]
```

Let's connect to our DigitalOcean server using `ssh`. Enter the following command in the terminal substituting the IP address you received from DigitalOcean for the IP address I'm using in this example:

```
ssh root@159.203.134.88
```

```
## The authenticity of host '159.203.134.88 (159.\
203.134.88)' can't be established.
## ECDSA key fingerprint is SHA256:UhtoIx/3c6/MmA\
IE+H8w5oGE06PsbXdzRRsAUhKtjhs.
## Are you sure you want to continue connecting (\
yes/no)?
```

Type `yes` and then press Enter.

```
## Warning: Permanently added '159.203.134.88' (E\
CDSA) to the list of known hosts.
## root@159.203.134.88's password:
```

This password should be in the email you received from DigitalOcean. Copy and paste the password into the terminal, then press Enter.

```
## You are required to change your password immed\
iately (root enforced)
## Welcome to Ubuntu 16.04.2 LTS (GNU/Linux 4.4.0\
-78-generic x86_64)
##
##  * Documentation:  https://help.ubuntu.com
##  * Management:     https://landscape.canonical\
.com
##  * Support:        https://ubuntu.com/advantage
##
##   Get cloud support with Ubuntu Advantage Clou\
d Guest:
```

```
##     http://www.ubuntu.com/business/services/cl\
oud
##
## 0 packages can be updated.
## 0 updates are security updates.
##
##
##
## The programs included with the Ubuntu system a\
re free software;
## the exact distribution terms for each program \
are described in the
## individual files in /usr/share/doc/*/copyright.
##
## Ubuntu comes with ABSOLUTELY NO WARRANTY, to t\
he extent permitted by
## applicable law.
##
## Changing password for root.
## (current) UNIX password:
```

We now need to create a new password for this server. First paste in the old password and press Enter. Then think of a new, strong password and enter it into the console. Then enter the new password again to confirm. After entering in the new password we should have a prompt! Press enter a few times to make sure that you get the prompt back each time. We're in!

Now we have access to all of the Unix commands we would normally have:

```
pwd
```

```
## /root
```

In order to disconnect from the server and return to your machine use `logout`.

```
logout
```

```
## Connection to 159.203.134.88 closed.
```

To reconnect to the server use `ssh` again:

```
ssh root@159.203.134.88
```

```
## root@159.203.134.88's password:
```

Enter your password and you should get the prompt back for your cloud server.

Summary

- `ssh` connects you to computers that are connected to the internet. The template for the command to connect is `ssh [username]@[IP address]`.
- To disconnect from an `ssh` session use the `logout` command.

Cloud Computing Basics

Moving Files In and Out of the Cloud

So now that we have a cloud computer, what can we do with it? One thing we can do is store and retrieve files from a cloud computer. The program `scp` allows us to copy local files to a server and it allows us to copy files on a server to our local computer. First let's connect to our server so we can create a file there:

```
ssh root@159.203.134.88
```

```
## root@159.203.134.88's password:
## # (Enter your password)
```

```
mkdir textfiles
echo "From the server" > textfiles/server-file.txt
logout
```

```
## Connection to 159.203.134.88 closed.
```

Now that we're back at the prompt on our local machine let's try getting `server-file.txt` from our server. The arguments for copying files from a server with `scp` have the following general structure:

```
scp [username]@[IP address]:path/to/file/on/serve\
r path/on/my/computer
```

This copies the file located on the server at `path/to/file/on/server` to a local path at `path/on/my/computer`. In the same way we can copy an entire folder from a server using the `-r` flag:

```
scp -r [username]@[IP address]:path/to/folder/on/\
server folder/on/my/computer
```

Let's try doing this now from our local computer. Enter your password when asked to do so:

```
cd
pwd


## /Users/sean/


mkdir Cloud
cd Cloud
scp root@159.203.134.88:/root/textfiles/server-fi\
le.txt downloaded.txt


## root@159.203.134.88's password:
## server-file.txt                                 \
          100%   16     1.2KB/s   00:00
```

```
cat downloaded.txt


## From the server
```

It worked! Now let's try uploading a file to our server. The arguments for doing this are just the swapped arguments for downloading a file from a server:

```
scp path/on/my/computer [username]@[IP address]:p\
ath/to/file/on/server
```

Let's create a file and upload it to our server:

```
echo "from local computer" > to-upload.txt
scp to-upload.txt root@159.203.134.88:/root/textf\
iles/uploaded-file.txt


## root@159.203.134.88's password:
## to-upload.txt                                  \
          100%   20     1.8KB/s   00:00
```

Now let's log in to our server and we'll see if it's there:

```
ssh root@159.203.134.88
cat textfiles/uploaded-file.txt
```

```
## from local computer
```

Looks like it worked! Keeping files in the cloud allows you to work with the same files in the same workspace as long as you have access to a terminal and `ssh`.

Talking to Other Servers

There are tons of servers out there on the internet! The way you're probably used to talking to a server is through a web browser, but there are other ways we can talk to servers on the command line. One of the most popular command line programs for talking to other servers is `curl`. The `curl` command allows us to send requests and information to other servers.

One easy task that we can use `curl` for is downloading files that are available online. For example, this entire book and all of the files associated with it are hosted on a server! You can find the Markdown file for one of the first chapters of this book here[36]. To download a file with `curl`, we simply need to provide the `-O` flag and the URL of the file:

```
curl -O http://website.org/textfile.txt
```

Let's try downloading the Markdown file from my website:

```
curl -O http://seankross.com/the-unix-workbench/0\
1-What-is-Unix.md
```

[36] http://seankross.com/the-unix-workbench/01-What-is-Unix.md

```
##   % Total    % Received % Xferd  Average Speed\
   Time    Time     Time  Current
##                                 Dload  Upload\
   Total   Spent    Left  Speed
## 100  1198  100  1198    0     0  13681      0 \
--:--:-- --:--:-- --:--:-- 13770


head -n 5 01-What-is-Unix.md


## # What is Unix?
##
## Unix is an operating system and a set of tools\
. The tool we'll be using the
## most in this book is a shell, which is a compu\
ter program that provides a
## command line interface. You've probably seen a\
 command line interface in the
```

Looks like we got the file! The `curl` command is also commonly used for communicating with **API**s. API stands for **a**pplication **p**rogramming **i**nterface. APIs are a set of rules which allow us to communicate with computer programs or with servers on the web. GitHub has a large API[37] which allows us to find out information about GitHub's users and repositories. Let's use `curl` to look at what programming languages are used by some of my repositories. Let's start with the repository for this book:

[37] https://developer.github.com/v3/

```
curl https://api.github.com/repos/seankross/the-u\
nix-workbench/languages
```

```
{
  "CSS": 2615,
  "TeX": 22
}
```

It looks like most of the repository is dedicated to making the book website look pretty! Take a look at the URL in the `curl` command above, and let's dissect it a little bit. The API itself is located at `https://api.github.com/`. Then each word in the rest of the url acts as a sort of argument. We're interested in `repos` in this case, specifically a repo belonging to the username `seankross` called `the-unix-workbench`, and we want to know about which `languages` are used in that repo. Let's take a look at one more of my repositories just to see how the response can be different:

```
curl https://api.github.com/repos/seankross/lego/\
languages
```

```
{
  "R": 4197,
  "Shell": 442
}
```

Use of `curl`, especially when coupled with using APIs can become very complicated and much more advanced content has been written on the subject. Let's get a little more in depth

by looking through some of the examples from httpbin.org[38]. This website allows us to send requests to it with `curl`, and it will return to us a structured version of whatever information we sent. This is useful for debugging our `curl` commands. First let's send a request which should return our IP address:

```
curl http://httpbin.org/ip
```

```
{
  "origin": "159.203.134.88"
}
```

Looks like we're getting the response we expect. Before we go on I should clarify: `curl` sends **HTTP requests**. HTTP is a technology for sending information over a network, and HTTP powers much of how the internet works. There are different categories of HTTP requests, and the categories are often called **verbs**. When we use `curl` without any flags we are sending a **GET** request (GET is an HTTP verb). A GET request is a message that says to a server: "Hi, I live at [IP address]. Would you mind sending some information about yourself to that IP?" In the case above we asked for our own IP address, which httpbin.org knew to just send back to us.

Let's send a general HTTP GET request to http://httpbin.org/get:

```
curl http://httpbin.org/get
```

[38] http://httpbin.org/

```
{
  "args": {},
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.47.0"
  },
  "origin": "159.203.134.88",
  "url": "http://httpbin.org/get"
}
```


The text that we get back from the request specifies four information groups: `args`, `headers`, `origin`, and `url`. The `origin` shows our own IP address, and `url` shows where we sent the request. The `headers` group shows some interesting information, including the `User-Agent` which shows that httpbin.org knows that we sent this request with `curl`. Notice that the `args` group is empty. The `args` group is short for *arguments*, which hints at the fact that we can provide arguments in an HTTP request, just like arguments we would use for a function, or the arguments we used in the GitHub API.

In the general case we can provide arguments to an HTTP API by putting a question mark (`?`) after the API's URL. Let's try this out:

```
curl http://httpbin.org/get?Baltimore
```

```
{
  "args": {
    "Baltimore": ""
  },
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.47.0"
  },
  "origin": "159.203.134.88",
  "url": "http://httpbin.org/get?Baltimore"
}
```


Looks like `"Baltimore"` showed up in `args`! For most HTTP APIs we need to give names to our arguments, unlike most arguments in Bash. We can specify an argument's name with the template `[argument name]=[argument value]`. Let's take a look at a simple example:

```
curl http://httpbin.org/get?city=Baltimore
```


```
{
  "args": {
    "city": "Baltimore"
  },
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.47.0"
```

```
  },
  "origin": "159.203.134.88",
  "url": "http://httpbin.org/get?city=Baltimore"
}
```

Now we can see that in `args` there's a correspondence between `city` and `Baltimore`. We can add more named arguments by separating them with an ampersand (`&`):

```
curl "http://httpbin.org/get?city=Baltimore&state\
=Maryland"
```

```
{
  "args": {
    "city": "Baltimore",
    "state": "Maryland"
  },
  "headers": {
    "Accept": "*/*",
    "Connection": "close",
    "Host": "httpbin.org",
    "User-Agent": "curl/7.47.0"
  },
  "origin": "159.203.134.88",
  "url": "http://httpbin.org/get?city=Baltimore&s\
tate=Maryland"
}
```

Perhaps you could imagine building a server that accepts HTTP requests, and sends back different information depending on what arguments are provided (for example, send

back a weather report given a location). Building these kinds of servers is an advanced topic that is outside the scope of this book, but there are lots of resources out there if you're interested in building your own HTTP API on a web server.

Automating Tasks

One of the most compelling features about any web server is that it's always powered on and always connected to the internet. This means that we can instruct our server to perform tasks automatically, without us needing to enter a command into a shell. One of the most commonly used programs for executing *other programs* with a regular frequency is called `cron`. Let's take a look at how to use `cron` to schedule a program to be run.

If you're not already connected to the server use `ssh` to connect.

```
ssh root@159.203.134.88
```

The `cron` program is part of a family of programs called **daemons**. A daemon is a program that is always running in the background of our computer. First, let's see if `cron` is running. We can get a list of all running programs with the `ps` command while using the `-A` flag:

```
ps -A
```

```
##  PID TTY          TIME CMD
##    1 ?        00:00:13 systemd
##    2 ?        00:00:00 kthreadd
##    3 ?        00:00:03 ksoftirqd/0
##    5 ?        00:00:00 kworker/0:0H
##    7 ?        00:00:11 rcu_sched
##    8 ?        00:00:00 rcu_bh
##    9 ?        00:00:00 migration/0
## ...
```

You probably have a huge list of programs in your terminal now! Instead of sifting through this listing line-by-line, let's pipe the output of this command to `grep` and we'll look for `cron`:

```
ps -A | grep "cron"
```

```
##  1273 ?        00:00:01 cron
```

Looks like the `cron` daemon is running! In order to assign programs to be executed with `cron` we need to edit a special text file called the `cron` table. Before we edit the `cron` table we need to select the default text editor. If you like using `nano` (the text editor we've been using throughout this book) then enter `select-editor` into the console, type in the number that corresponds to `nano` (usually `2`) and then press enter:

```
select-editor
```

```
## Select an editor.  To change later, run 'selec\
t-editor'.
##   1. /bin/ed
##   2. /bin/nano        <---- easiest
##   3. /usr/bin/vim.basic
##   4. /usr/bin/vim.tiny
##
## Choose 1-4 [2]:
```

Now that we've chosen a text editor we can edit the `cron` table using the command `crontab -e` (**cron tab**le **e**dit) which will automatically open `nano` with the appropriate file.

```
crontab -e
```

```
# Edit this file to introduce tasks to be run by \
cron.
#
# m h  dom mon dow   command
```

Let's go over the layout of the `cron` table. First you should notice that any text after a pound sign (`#`) is a comment, so it's not seen by `cron` (just like bash comments). The `cron` table has six columns:

1. Minute (`m`)
2. Hour (`h`)
3. Day of Month (`dom`)
4. Month (`mon`)
5. Day of Week (`dow`)

6. Command to be run (`command`)

Each column is separated by a single space in the table. The first five columns allow you to specify when you want a particular command to be run. Only certain values are valid in each column:

1. Minute: `00 - 59` (A particular minute in an hour)
2. Hour: `00 - 23` (0 is the midnight hour)
3. Day of Month: `01 - 31` (1 is the first day of the month)
4. Month: `01 - 12` (1 is January)
5. Day of Week `0 - 6` (0 is Sunday)

There are also a few other characters that are valid in the `cron` table. The most commonly used character is a star (*) which represents *all* of the possible values in a column. So a star in the Minute column means "run every minute," and a star in the Hour column means "run during every hour." Knowing this let's make our first entry in the `cron` table. If we want a command to be executed every minute, during every hour, on every day of the month, during every month, on every day of the week, then we can put stars in all of the first five columns, followed by the command that we want to run. In this case the command that `cron` will run every minute will be `date >> ~/date-file.txt`, which will append the date and time when the command is executed to a file in our home directory called `date-file.txt`. This is what your `cron` table should look like before you save and exit from `nano`:

```
# Edit this file to introduce tasks to be run by \
cron.
#
# m h  dom mon dow   command
* * * * * date >> ~/date-file.txt
```

Save and exit `nano` just like you would for a regular text file and then wait a little bit! After a minute has gone by use `cat` to look at `~/date-file.txt`:

```
cd
cat date-file.txt
```

```
## Thu Jun  8 18:50:01 UTC 2017
```

Look like our entry in the `cron` table is working! Wait another minute and then look at the file again:

```
cat date-file.txt
```

```
## Thu Jun  8 18:50:01 UTC 2017
## Thu Jun  8 18:51:01 UTC 2017
```

Unless we delete the line that we entered in the `cron` table, the output from `date` will be appended to `date-file.txt` every minute.

The single line of bash `date >> ~/date-file.txt` is a much simpler program than we would probably use in a `cron` table, though it's good for illustrating how a `cron` table works. If

you want to do more complex tasks with `cron` it's better for `cron` to execute a bash script that you've written in advance. That way you can just specify `bash /path/to/script.sh` in the last column of the table.

Using stars in all columns is the simplest line of a `cron` table, so let's look at some examples of more complex table entries:

```
# m h  dom mon dow   command
00 * * * * bash /path/to/script.sh     # Runs eve\
ry hour at the start of the hour
00 12 * * * bash /path/to/script.sh    # Runs eve\
ry day at noon
* 12 * * * bash /path/to/script.sh     # Runs eve\
ry minute between 12pm and 12:59pm
00 00 05 * * bash /path/to/script.sh   # Runs the\
 5th day of every month at midnight
00 00 * 07 * bash /path/to/script.sh   # Runs eve\
ry day in the month of July at midnight
00 00 * * 2 bash /path/to/script.sh    # Runs eve\
ry Tuesday at midnight
```

Besides numbers and the star there are a few other characters that you can use in `cron` table columns including a hyphen (`-`) for specifying ranges and a comma (`,`) for specifying lists of items. For example `00-29` in the Minutes column would specify the first thirty minutes of an hour, while `1,5` in the Day of Week column would specify Monday and Friday.

Let's take a look at another example of a `cron` table that uses hyphens and ranges so you can get a sense of how each character works.

```
# m h  dom mon dow   command
00-04 * * * * bash /path/to/script.sh       # Run\
s every minute for the first five minutes of ever\
y hour
00 00 * * 0,6 bash /path/to/script.sh       # Run\
s at midnight every Saturday and Sunday
00 03 01-15 * * bash /path/to/script.sh     # Run\
s at 3am for the first fifteen days of every mont\
h
00,30 * * * * bash /path/to/script.sh       # Run\
s at the start and middle of every hour
00 00,12 * * * bash /path/to/script.sh      # Run\
s every day at midnight and noon
00 * 01-07 01,06 * bash /path/to/script.sh  # Run\
s at the start of every hour for the first seven \
days of the months of January and June
```

A program that is being run by `cron` is only as powerful as your imagination can stretch! If you're familiar with the social network Twitter[39] then you might have come across some Twitter accounts which create posts automatically like Emoji Aquarium[40], Old Fruit Pictures[41], or Endless Screaming[42]. Many of these "bot" accounts are powered by `cron`, which uses Twitter's HTTP API to post tweets regularly.

Summary

- `scp` copies files between a cloud computer and your

[39] https://twitter.com/
[40] https://twitter.com/emojiaquarium
[41] https://twitter.com/pomological
[42] https://twitter.com/infinite_scream

personal computer. Use the `-r` flag in order to copy directories.

- `curl` allows you to send HTTP requests to other servers. Use the `-O` flag to download files with `curl`.
- `ps -A` lists all of the programs running in the background of your computer.
- `cron` allows you to schedule when programs are run. Use `crontab -e` in order to edit the `cron` table.

Exercises

1. Write a bash script that takes a file path as an argument and copies that file to a designated folder on your server.
2. Find a file online that changes periodically, then write a program to download that file every time it changes.
3. Try creating your own Twitter or GitHub bot with the Twitter API[43] or the GitHub API[44].

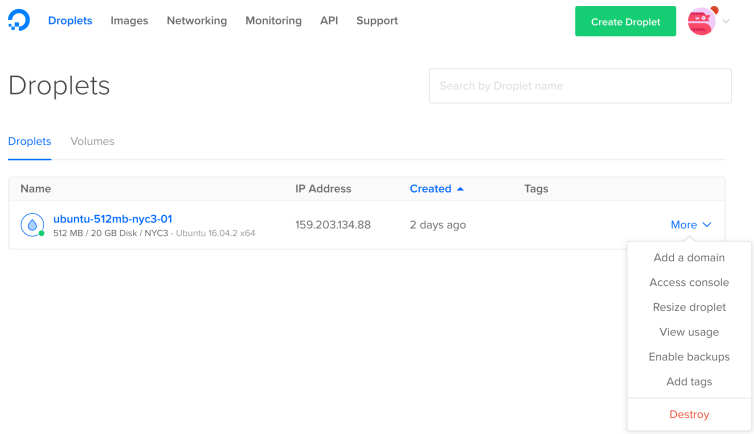
Shutting Down a Server

In order to avoid using more DigitalOcean credits than we have to or being charged for using the service make sure to **destroy** any DigitalOcean droplets that you started. When you destroy a droplet all files on the droplet are gone forever, so please be sure that you don't have any important information on a droplet before you destroy it. If there are files you want to save on your droplet use `scp` in order to copy them to

[43] https://dev.twitter.com/rest/public

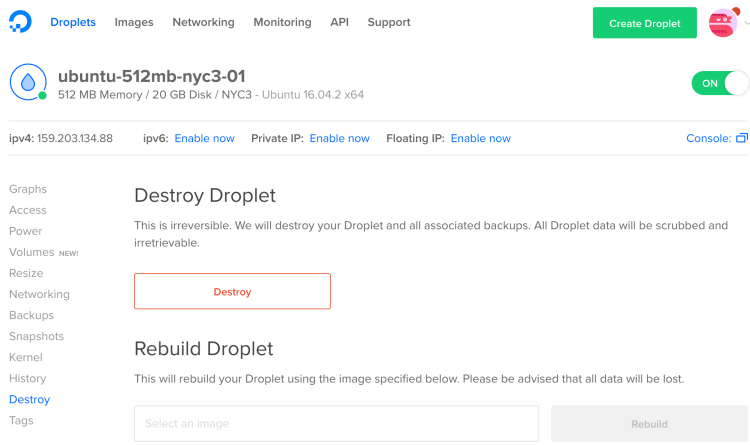
[44] https://developer.github.com/v3/

your local machine. Let's walk through destroying a droplet. Go back to DigitalOcean[45] and log in. You should then see a listing of all of your droplets:

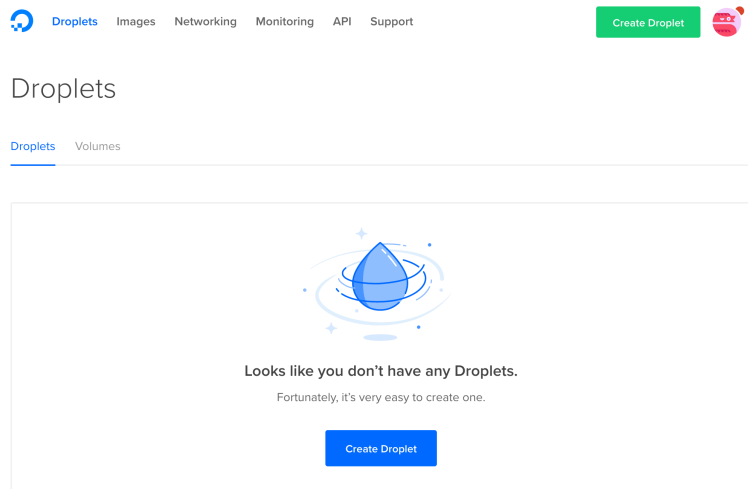


Click on **More** on the right side of the screen and a menu should appear. Click **Destroy** at the bottom of this menu. Then this screen should appear:

[45] https://m.do.co/c/530d6cfa2b37



Click the **Destroy** button and then click **Confirm** in the menu that pops up. Your droplet should now be destroyed. If you have no active droplets then the main droplets page should look like this:



Congratulations on your new cloud computing skills!

Start Building

Next Steps

We've reached the end of this book, which means that you now have a formidable foundation in using Unix. Congratulations! As the title of this book suggests Unix serves mostly as a workbench - a set of tools for building amazing digital creations. However, what you can create with Unix is usually not made out of Unix's constituent parts. You might use a hammer and saw to build a good birdhouse, but the birdhouse itself isn't made out of hammers. Working knowledge of Unix is best complemented by knowing at least one other programming language. Here are a few suggestions about how you can continue your computing and programming education.

Python[46] is an approachable and essential language for anyone interested in computing. If you don't have any programming experience outside of this book, I very highly recommend learning Python. My favorite book on the subject is Learn Python the Hard Way[47] by Zed Shaw. Philip Guo's Python Tutor[48] allows you to visualize how Python is working under-the-hood, which allows you to develop a better intuition about the code you're writing. The pairing of those two resources is currently the best way to learn how to write

[46] https://www.python.org/

[47] https://learnpythonthehardway.org/book/

[48] http://pythontutor.com/

software outside of formal university or corporate training. Once you have some Python experience you can try building your own HTTP API with Flask[49].

R[50] is a general purpose programming language designed for folks who are interested in data science, analysis, modeling, and visualization. R is also fantastic for making digital documents: this book was created with R! If you want to get started with R I recommend the book R Programming for Data Science[51], the Swirl[52] software package, and the interactive R tutorial website called DataCamp[53].

JavaScript[54] is the main language that powers the internet and it forms the backbone of web application programming. The Mozilla Development Network has wonderful tutorials[55] about HTML, CSS, and JavaScript. Usually you use JavaScript to manipulate HTML and CSS, so learning about all three is important! I also recommend NodeSchool[56] for purely learning about JavaScript with Unix.

Giving Feedback

Thank you so much for reading this book! If you'd like to discuss the book or you have any feedback I would love to hear from you. The best way to contact me is on Twitter[57].

[49] http://flask.pocoo.org/
[50] https://www.r-project.org/
[51] https://leanpub.com/rprogramming
[52] http://swirlstats.com/
[53] https://www.datacamp.com/
[54] https://developer.mozilla.org/en-US/docs/Web/JavaScript
[55] https://developer.mozilla.org/en-US/docs/Web/Tutorials
[56] https://nodeschool.io/
[57] https://twitter.com/seankross

Now that you know how to use Git and GitHub you can submit changes that you think should be made to this book including fixing typos and correcting errors. You can find the repository for this book here[58]. Fork the repository, and make your changes to the appropriate `.Rmd` file (just treat it like a regular Markdown file). Add, commit, and push your changes, then send me a pull request! While you're on the GitHub if you wouldn't mind giving this book's repository a **Star**[59] I would really appreciate it so that others can find this book more easily.

[58] https://github.com/seankross/the-unix-workbench
[59] https://github.com/seankross/the-unix-workbench