Technical Foundations of Informatics

Michael Freeman and Joel Ross

May 31, 2017

Contents

About this Book

This book covers the foundation skills necessary to start ***writing computer programs to work with data*** using modern and reproducable techniques. It requires no technical background. These materials were developed for the **INFO 201: Technical Foundations of Informatics** course taught at the University of Washington Information School; however they have been structured to be an online resource for anyone hoping to learn to work with information using programmatic approaches.

This book is currently in **beta** status. Visit us on GitHub to contribute improvements.

Chapter 1

Setting up your Machine

We'll be using a variety of different software programs to write, manage, and execute the code that we write. Unfortunately, one of the most frustrating and confusing barriers to working with code is simply getting your machine properly set up. This chapter aims to provide sufficient information for setting up your machine, and troubleshooting the process.

Note that iSchool lab machines should have all appropriate software already installed and ready to use.

In short, you'll need to install the following programs: see below for more information / options.

- **Git**: A set of tools for tracking changes to computer code (especially when collaborating with others). This program is already installed on Macs.
  - **GitHub**: A web service for hosting code online. You don't actually need to *install* anything GitHub (it uses `git`), but you'll need to sign up for the service.
- **Bash**: A *command-line interface* for controlling your computer. `git` is a command-line program so you'll need a command shell to use it. Macs already have a Bash program called *Terminal*. On Windows, installing `git` will also install a Bash shell called *Git Bash*, or you can try the (experimental) Linux subsystem for Windows 10.
- **Atom**: A lightweight text editor that supports programming in lots of different languages.
  - You are welcome to use another text editor if you wish; some further suggestions are included.
- **R**: a programming language commonly used for working with data. This will be our primary language for the quarter. "Installing R" actually means installing tools that will let your computer understand and run R code.

- **RStudio**: An graphical editor for writing and running R code. This will soon become our primary development application.

The following sections have additional information about the purpose of each component, how to install it, and alternative configurations.

If you run into any installation/configuration challenges, please let others know on the slack channel so that others can anticipate the same issues.

1.1 Git

`git` is a version control system that provides a set of commands that allow you to manage changes to written code, particularly when collaborating with other programmers (much more on this in chapter 4). To start, you'll need to download and install the software. If you are on a Mac, `git` should already be installed.

If you are using a Windows machine, this will also install a program called Git Bash, which provides a text-based interface for executing commands on your computer. For alternative/additional Windows command-line tools, see below.

1.1.1 GitHub

GitHub is a website that is used to store copies of computer code that are being managed with `git` (think "Imgur for code"). Students in the INFO 201 course will use GitHub to turn in programming assignments.

In order to use GitHub, you'll need to create a free GitHub account, if you don't already have one. You should register a username that is identifiable as you (e.g., based on your name or your UW NetID). This will make it easier for others to determine out who contributed what code, rather than needing to figure out who 'LeetDesigner2099' is. This can be the start of a professional account you may use for the rest of your career!

1.2 Command-line Tools (Bash)

The command-line provides a text-based interface for giving instructions to your computer (much more on this in chapter 2). With this book, you'll largely use the command-line for navigating your computer's file structure, and executing commands that allows you to keep track of changes to the code you write (i.e., version control with `git`).

In order to use the command-line, you will need to use a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. In particular, we'll be working with the Bash shell, which provides a particular common set of commands common to Mac and Linux machines.

1.2.1 Command-line on a Mac

On a Mac you'll want to use the built-in app called **Terminal**. You can open it by searching via Spotlight (hit Cmd (⌘) and Spacebar together, type in "terminal", then select the app to open it), or by finding it in the Applications/Utilities folder.

1.2.2 Command-line on Windows

On Windows, we recommend using **Git Bash**, which you should have installed along with `git` (above). Open this program to open the command-shell. This works great, since you'll primarily be using the command-line for performing version control.

- Note that Windows does come with its own command-prompt, called the *DOS Prompt*, but it has a different set of commands and features. *Powershell* is a more powerful version of the DOS prompt if you really want to get into the Windows Management Framework. But Bash is more common in open-source programming like we'll be doing, and so we will be focusing on that set of commands.

Alternatively, the 64-bit Windows 10 Anniversary update (August 2016) *does* include a beta version of an integrated Bash shell. You can access this by enabling the subsystem for Linux and then running `bash` in the command prompt. This is currently (May 2017) "beta" technology, but will suffice for our purposes if you can get it running.

1.3 Text Editors

In order to produce computer code, you need somewhere to write it (and we don't want to write it in MS Word!). There are a variety of available programs that provide an interface for editing code. A major advantage of these programs is that they provide automatic formatting/coloring for easier interpretation of the code, along with cool features like auto-completion and integration with version control.

RStudio has a great built-in text editor, but you'll sometimes want to use another text editor which is lighter weight (e.g., runs faster), more robust, or supports a different programming language. There are lots of different text editors out there, all of which have slightly different appearances and features. You only need to download and use one of the following programs (we recommend

Atom as a default), but feel free to try out different ones to find something you like (and then evangelize about it to your friends!)

1.3.1 Atom

Atom is a text editor built by the folks at GitHub and has been gaining in popularity. As an open source project, people are continually building (and making available) interesting/useful extensions. Its built-in spell-check is a great feature, especially for documents that require lots of written text. It also has excellent support for Markdown, a markup language you'll be using regularly in this course.

Click the "Download" button to download the installer `.exe` file, then double-click on that to install the application.

Once you've installed Atom, the trick to using it effectively is to get comfortable with the Command Palette. If you hit `Cmd+Shift+P`, Atom will open a small window where you can search for whatever you want the editor to do. For example, if you type in `markdown` you can get list of commands related to Markdown files (including the ability to open up a preview).

For more information about using Atom, see the manual.

1.3.2 Visual Studio Code

Visual Studio Code (or VS Code; not to be confused with Visual Studio) is a free, open-source editor developed by Microsoft—yes, really. While it focuses on web programming and JavaScript, it readily supports lots of languages including Markdown and R and provides a number of extensions for adding even more features. It has a similar *command palette* to Atom, but isn't quite as nice for editing Markdown specifically. Although fairly new, it is updated regularly and has become one of my main editors for programming.

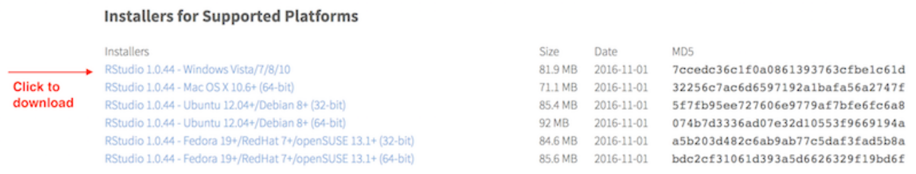
1.3.3 SublimeText

SublimeText is a very popular text editor with excellent defaults and a variety of available extensions (though you'll need to manage and install extensions to achieve the functionality offered by other editors out of the box). While the software can be used for free, every 20 or so saves it will prompt you to purchase the full version. This is my application of choice for when just want to write a plain text file.

1.4 RStudio

The primary programming language you will use throughout the course is `R`. It's a very powerful statistical programming language that is built to work well with large and diverse datasets. While you are able to execute `R` scripts without an interface, the **RStudio** application provides a wonderful way to engage with the `R` language.

To install the RStudio program, select the **installer** for your operating system from the downloads page. Make sure to download the ***free*** version:



Installers for Supported Platforms

Installers	Size	Date	MD5
RStudio 1.0.44 - Windows Vista/7/8/10	81.9 MB	2016-11-01	7ccedc36c1f0a0861393763cfbe1c61d
RStudio 1.0.44 - Mac OS X 10.6+ (64-bit)	71.1 MB	2016-11-01	32256c7ac6d6597192a1bafa56a2747f
RStudio 1.0.44 - Ubuntu 12.04+/Debian 8+ (32-bit)	85.4 MB	2016-11-01	5f7fb95ee727606e9779af7bfe6fc6a8
RStudio 1.0.44 - Ubuntu 12.04+/Debian 8+ (64-bit)	92 MB	2016-11-01	074b7d3336ad07e32d10553f9669194a
RStudio 1.0.44 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (32-bit)	84.6 MB	2016-11-01	a5b203d482c6ab9ab77c5daf3fad5b8a
RStudio 1.0.44 - Fedora 19+/RedHat 7+/openSUSE 13.1+ (64-bit)	85.6 MB	2016-11-01	bdc2cf31061d393a5d6626329f19bd6f

Figure 1.1: File to choose for downloading RStudio. Image may not show the latest version.

Once the download completes, double-click on the `.exe` or `.dmg` file to run the installer. Simply follow the steps of the installer, and you should be prepared to use RStudio.

By downloading RStudio, you will also install the R programming language, if it is not already present in your operating system. However, if there are any problems with that you can also install R separately. Click on the download page for your operating system in order to find a link to the installer. On a Mac you're looking for the `.pkg` file, and for Windows you want to look in the `base` section (follow the link to "install R for the first time").

Resources

Links to the recommended software are collected here for easy access:

- git (and Git Bash)
  - GitHub (sign up)
  - optional: Bash on Windows
- Atom
- R
- RStudio

Chapter 2

The Command Line

The **command-line** is an *interface* to a computer—a way for you (the human) to communicate with the machine. But unlike common graphical interfaces that use windows, icons, menus, and pointers, the command-line is *text-based*: you type commands instead of clicking on icons. The command-line lets you do everything you'd normally do by clicking with a mouse, but by typing in a manner similar to programming!

The command-line is not as friendly or intuitive as a graphical interface: it's much harder to learn and figure out. However, it has the advantage of being both more powerful and more efficient in the hands of expert users. (It's faster to type than to move a mouse, and you can do *lots* of "clicks" with a single command). Thus all professional developers interact with the command-line, particularly when working with large amounts of data or files.

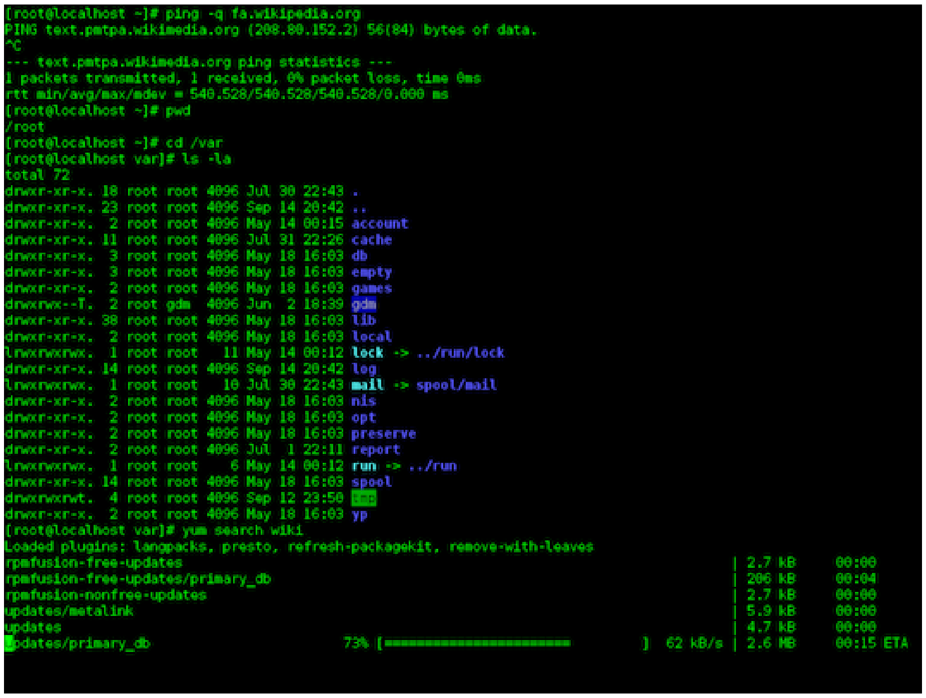
This chapter will give you a brief introduction to basic tasks using the command-line: enough to get you comfortable navigating the interface and able to interpret commands.

2.1 Accessing the Command-Line

In order to use the command-line, you will need to open a **command shell** (a.k.a. a *command prompt*). This is a program that provides the interface to type commands into. You should have installed a command shell (hereafter "the terminal") as part of setting up your machine.

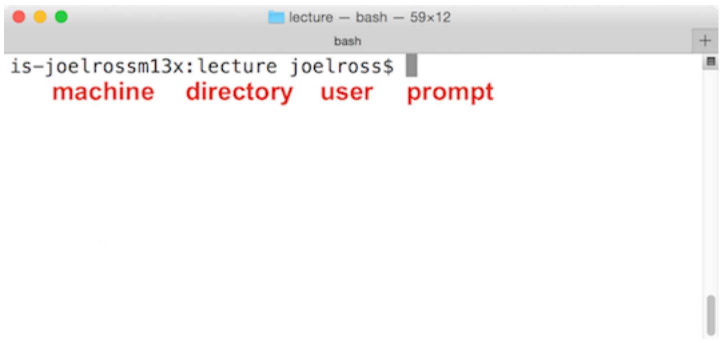
Once you open up the shell (Terminal or Git Bash), you should see something like this (red notes are added):

This is the textual equivalent of having opened up Finder or File Explorer and having it show you the user's "Home" folder. The text shown lets you know:



```
[root@localhost ~]# ping -q fa.wikipedia.org
PING text.pmtpa.wikimedia.org (208.80.152.2) 56(84) bytes of data.
^C
--- text.pmtpa.wikimedia.org ping statistics ---
1 packets transmitted, 1 received, 0% packet loss, time 0ms
rtt min/avg/max/mdev = 540.528/540.528/540.528/0.000 ms
[root@localhost ~]# pwd
/root
[root@localhost ~]# cd /var
[root@localhost var]# ls -la
total 72
drwxr-xr-x. 18 root root 4096 Jul 30 22:43 .
drwxr-xr-x. 23 root root 4096 Sep 14 20:42 ..
drwxr-xr-x.  2 root root 4096 May 14 00:15 account
drwxr-xr-x. 11 root root 4096 Jul 31 22:26 cache
drwxr-xr-x.  3 root root 4096 May 18 16:03 db
drwxr-xr-x.  3 root root 4096 May 18 16:03 empty
drwxr-xr-x.  2 root root 4096 May 18 16:03 games
drwxrwx--T.  2 root gdm  4096 Jun  2 18:39 gdm
drwxr-xr-x. 38 root root 4096 May 18 16:03 lib
drwxr-xr-x.  2 root root 4096 May 18 16:03 local
lrwxrwxrwx.  1 root root   11 May 14 00:12 lock -> ../run/lock
drwxr-xr-x. 14 root root 4096 Sep 14 20:42 log
lrwxrwxrwx.  1 root root   10 Jul 30 22:43 mail -> spool/mail
drwxr-xr-x.  2 root root 4096 May 18 16:03 nis
drwxr-xr-x.  2 root root 4096 May 18 16:03 opt
drwxr-xr-x.  2 root root 4096 May 18 16:03 preserve
drwxr-xr-x.  2 root root 4096 Jul  1 22:11 report
lrwxrwxrwx.  1 root root    6 May 14 00:12 run -> ../run
drwxr-xr-x. 14 root root 4096 May 18 16:03 spool
drwxrwxrwt.  4 root root 4096 Sep 12 23:50 tmp
drwxr-xr-x.  2 root root 4096 May 18 16:03 yp
[root@localhost var]# yum search wiki
Loaded plugins: langpacks, presto, refresh-packagekit, remove-with-leaves
rpmfusion-free-updates                                                  | 2.7 kB     00:00
rpmfusion-free-updates/primary_db                                       | 206 kB     00:04
rpmfusion-nonfree-updates                                               | 2.7 kB     00:00
updates/metalink                                                        | 5.9 kB     00:00
updates                                                                 | 4.7 kB     00:00
updates/primary_db                    73% [=======================       ]  62 kB/s | 2.6 MB     00:15 ETA
```

Figure 2.1: An example of the command-line in action (from Wikipedia).



```
is-joelrossm13x:lecture joelross$
```

Figure 2.2: A newly opened command-line.

- What machine you're currently interfacing with (you can use the command-line to control different computers across a network or the Internet).
- What **directory** (folder) you are currently looking at (~ is a shorthand for the "home directory").
- What user you are logged in as.

After that you'll see the prompt, which is where you will type in your commands.

2.2 Navigating the Command Line

Although the command-prompt gives you the name of the folder we're in, you might like more detail about where that folder is. Time to send your first command! At the prompt, type:

```
pwd
```

This stands for **p**rint **w**orking **d**irectory (shell commands are highly abbreviated to make them faster to type), and will tell the computer to print the folder you are currently "in".

Fun fact: technically this command starts a tiny program (app) that does exactly one thing: prints the working directory. When you run a command, you're actually executing a tiny program! And when you run programs (tiny or large) on the command-line, it looks like you're typing in commands.

Folders on computers are stored in a hierarchy: each folder has more folders inside it, which have more folders inside them. This produces a tree structure:

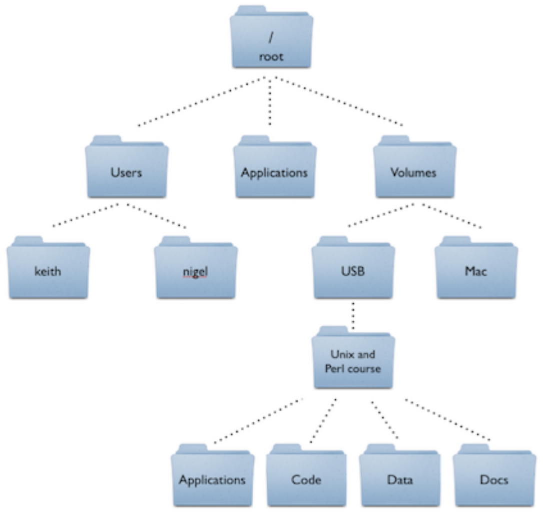


Figure 2.3: A Directory Tree, from Bradnam and Korf.

We describe what folder we are in putting a slash `/` between each folder in the tree: thus `/Users/iguest` means "the `iguest` folder, which is inside the `Users` folder".

At the very top (or bottom, depending on your point of view) is the **root** `/` directory-which has no name, and so is just indicated with that single slash. So `/Users/iguest` really means "the `iguest` folder, which is inside the `Users` folder, which is inside the root folder".

2.2.1 Changing Directories

What if you want to change folders? In a graphical system like Finder, you would just double-click on the folder to open it. But there's no clicking on the command-line.

This includes clicking to move the cursor to an earlier part of the command you typed. You'll need to use the left and right arrow keys to move the cursor instead!

Protip: The up and down arrow keys will let you cycle though your previous commands so you don't need to re-type them!

Since you can't click on a folder, you'll need to use another command:

```
cd folder_name
```

The first word is the **command**, or what you want the computer to do. In this case, you're issuing the command that means **c**hange **d**irectory.

The second word is an example of an **argument**, which is a programming term that means "more details about what to do". In this case, you're providing a *required* argument of what folder you want to change to! (You'll of course need to replace `folder_name` with the name of the folder).

- Try changing to the `Desktop` folder, which should be inside the home folder you started in—you could see it in Finder or File Explorer!
- After you change folders, try printing your currently location. Can you see that it has changed?

2.2.2 Listing Files

In a graphical system, once you've double-clicked on a folder Finder will show you the contents of that folder. The command-line doesn't do this automatically; instead you need another command:

```
ls [folder_name]
```

This command says to **lis**t the folder contents. Note that the *argument* here is in brackets (`[]`) to indicate that it is *optional*. If you just issue the **`ls`** command without an argument, it will list the contents of the current folder. If you include the optional argument (leaving off the brackets), you can "peek" at the contents of a folder you are not currently in.

The command-line can be not great about giving **feedback** for your actions. For example, if there are no files in the folder, then `ls` will simply show nothing, potentially looking like it "didn't work". Or when typing a **password**, the letters you type won't show (not even as `*`) as a security measure.

Just because you don't see any results from your command/typing, doesn't mean it didn't work! Trust in yourself, and use basic commands like `ls` and `pwd` to confirm any changes if you're unsure. Take it slow, one step at a time.

2.2.3 Paths

Note that both the **`cd`** and **`ls`** commands work even for folders that are not "immediately inside" the current directory! You can refer to *any* file or folder on the computer by specifying its **path**. A file's path is "how you get to that file": the list of folders you'd need to click through to get to the file, with each folder separated by a `/`:

`/Users/iguest/Desktop/myfile.txt`

This says to start at the root directory (that initial `/`), then go to `Users`, then go to `iguest`, then to `Desktop`, and finally to the `myfile.txt` file.

Because this path starts with a specific directory (the root directory), it is referred to as an **absolute path**. No matter what folder you currently happen to be in, that path will refer to the correct file because it always starts on its journey from the root.

Contrast that with:

`iguest/Desktop/myfile.txt`

Because this path doesn't have the leading slash, it just says to "go to the Desktop folder *from the current location*". It is known as a **relative path**: it gives you directions to a file *relative to the current folder*. As such, the relative path `iguest/Desktop/myfile.txt` path will only refer to the correct file if you happen to be in the `/Users` folder; if you start somewhere else, who knows where you'll end up!

You should **always** use relative paths, particularly when programming! That way file directions are more likely to work across computers (e.g., in case the username is different, making your home folder `janesmith` instead of `iguest`; with a relative path, `Desktop/myfile.txt` will work for either person).

You can refer to the "current folder" by using a single dot **`.`**. So the command

```
ls .
```

means "list the contents of the current folder" (the same thing you get if you leave off the argument).

If you want to go *up* a directory, you use *two* dots: `..` to refer to the **parent** folder (that is, the one that contains this one). So the command

```
ls ..
```

means "list the contents of the folder that contains the current folder".

Note that `.` and `..` act just like folder names, so you can include them anywhere in paths: `../../my_folder` says to go up two directories, and then into `my_folder`.

Super Protip: Most command shells like Terminal and Git Bash support **tab-completion**. If you type out just the first few letters of a file or folder name and then hit the `tab` key, it will automatically fill in the rest of the name! If the name is ambiguous (e.g., you type `Do` and there is both a `Documents` and a `Downloads` folder), you can hit `tab` *twice* to see the list of matching folders. Then add enough letters to distinguish them and tab to complete! This will make your life better.

Also remember that you can use a tilde `~` as shorthand for the home directory of the current user. Just like `.` refers to "current folder", ~ refers to the user's home directory (usually `/Users/USERNAME`). And of course, you can use the tilde as part of a path as well.

2.3 File Commands

Once you're comfortable navigating folders in the command-line, you can start to use it to do all the same things you would do with Finder or File Explorer, simply by using the correct command:

Command	Behavior
`mkdir`	**m**a**k**e a **dir**ectory
`rm`	**r**e**m**ove a file or folder
`cp`	**c**o**p**y a file from one location to another
`open`	Mac: opens a file or folder
`start`	Windows: opens a file or folder
`cat`	con**cat**enate (combine) file contents and display the results
`history`	show previous commands executed

Be aware that many of these commands **won't print anything** when you run them. This often means that they worked; they just did so quietly. If it *doesn't* work, you'll know because you'll see a message telling you so (and why, if you read the message). So just because you didn't get any output doesn't mean you

Figure 2.4:

did something wrong—you can use another command (such as **`ls`**) to confirm that the files or folders changed the way you wanted!

2.3.1 Learning New Commands

How can you figure out what kind of arguments these commands take? You can look it up! This information is available online, but many command shells (but *not* Git Bash, unfortunately) also include their own manual you can use to look up commands!

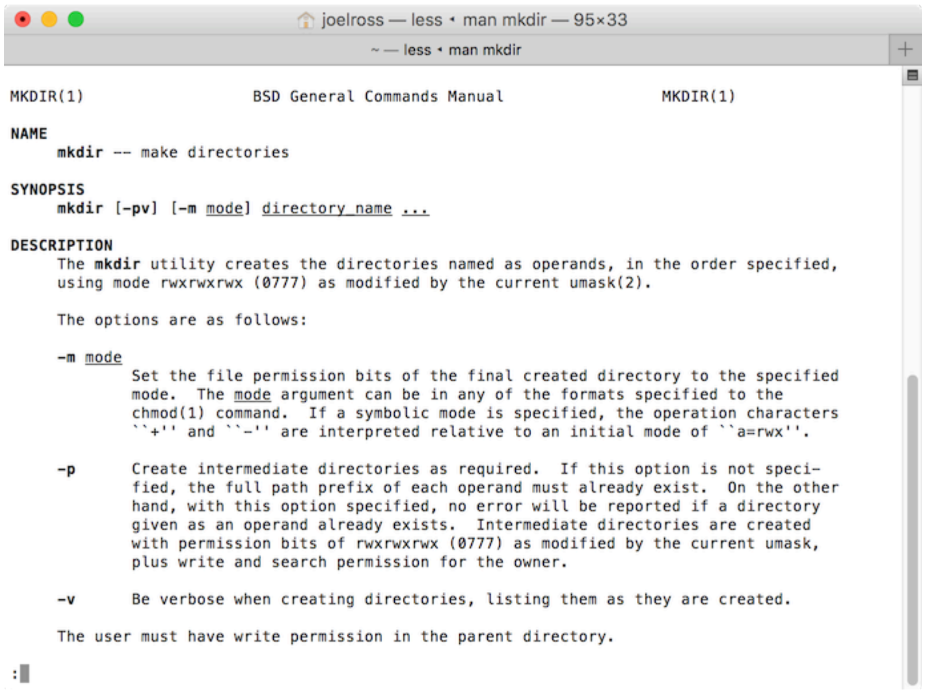
```
man mkdir
```

Will show the **man**ual for the **`mkdir`** program/command.

Because manuals are often long, they are opened up in a command-line viewer called `less`. You can "scroll" up and down by using the arrow keys. Hit the `q` key to **q**uit and return to the command-prompt.

If you look under "Synopsis" you can see a summary of all the different arguments this command understands. A few notes about reading this syntax:

- Recall that anything in brackets `[]` is optional. Arguments that are not in brackets (e.g., `directory_name`) are required.
- **"Options"** (or "flags") for command-line programs are often marked with a leading dash `-` to make them distinct from file or folder names. Options may change the way a command-line program behaves—like how you might set "easy" or "hard" mode in a game. You can either write out



```
MKDIR(1)                  BSD General Commands Manual                  MKDIR(1)

NAME
     mkdir -- make directories

SYNOPSIS
     mkdir [-pv] [-m mode] directory_name ...

DESCRIPTION
     The mkdir utility creates the directories named as operands, in the order specified,
     using mode rwxrwxrwx (0777) as modified by the current umask(2).

     The options are as follows:

     -m mode
             Set the file permission bits of the final created directory to the specified
             mode.  The mode argument can be in any of the formats specified to the
             chmod(1) command.  If a symbolic mode is specified, the operation characters
             ``+'' and ``-'' are interpreted relative to an initial mode of ``a=rwx''.

     -p      Create intermediate directories as required.  If this option is not speci-
             fied, the full path prefix of each operand must already exist.  On the other
             hand, with this option specified, no error will be reported if a directory
             given as an operand already exists.  Intermediate directories are created
             with permission bits of rwxrwxrwx (0777) as modified by the current umask,
             plus write and search permission for the owner.

     -v      Be verbose when creating directories, listing them as they are created.

     The user must have write permission in the parent directory.

:
```

Figure 2.5: The `mkdir` man page.

each option individually, or combine them: **`mkdir -p -v`** and **`mkdir -pv`** are equivalent.

  - Some options may require an additional argument beyond just indicating a particular operation style. In this case, you can see that the `-m` option requires you to specify an additional `mode` parameter; see the details below for what this looks like.

- Underlined arguments are ones you choose: you don't actually type the word `directory_name`, but instead your own directory name! Contrast this with the options: if you want to use the `-p` option, you need to type `-p` exactly.

Command-line manuals ("man pages") are often very difficult to read and understand: start by looking at just the required arguments (which are usually straightforward), and then search for and use a particular option if you're looking to change a command's behavior.

For practice, try to read the man page for `rm` and figure out how to delete a folder and not just a single file. Note that you'll want to be careful, as this is a good way to break things.

2.4 Dealing With Errors

Note that the syntax of these commands (how you write them out) is very important. Computers aren't good at figuring out what you meant if you aren't really specific; you can't forget spaces or anything.

Try another command: **`echo`** lets you "echo" (print out) some text. Try echoing `"Hello World"` (which is the traditional first computer program):

```
echo "Hello world"
```

What happens if you forget the closing quote? You keep hitting "enter" but you just get that `>` over and over again! What's going on?

- Because you didn't "close" the quote, the shell thinks you are still typing the message you want to echo! When you hit "enter" it adds a *line break* instead of ending the command, and the `>` marks that you're still going. If you finally close the quote, you'll see your multi-line message printed!

IMPORTANT TIP If you ever get stuck in the command-line, hit **`ctrl-c`** (The `control` and `c` keys together). This almost always means "cancel", and will "stop" whatever program or command is currently running in the shell so that you can try again. Just remember: "**`ctrl-c`** to flee".

(If that doesn't work, try hitting the `esc` key, or typing `exit`, `q`, or `quit`. Those commands will cover *most* command-line programs).

Resources

- Learn Enough Command Line to be Dangerous
- Video series: Bash commands
- List of Common Commands (also here)

Chapter 3

Markdown

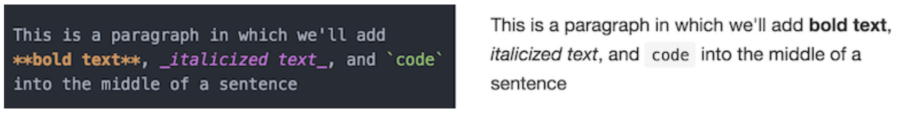
Markdown syntax provides a simple way to describe the desired formatting of text documents. In fact, this book was written using Markdown! With only a small handful of options, Markdown allows you to format to your text (like making text **bold**, or *italics*), as well as provide structure to a document. There are a number of programs and service that support the *rendering* of Markdown, including GitHub, Slack, and StackOverflow (though note the syntax may vary slightly across programs). In this chapter, you'll learn the basics of Markdown syntax, and how to leverage it to produce readable code documents.

3.1 Writing Markdown

Markdown is a lightweight markup language that is used to format and structure text. It is a kind of "code" that you write in order to *annotate* plain text: it lets the computer know that "this text is bold", "this text is a heading", etc. Compared to other markup languages, Markdown is easy to write and easy to read without getting in the way of the text itself. And because it's so simple to include, it's often used for formatting in web forums and services (like Wikipedia or StackOverflow).

3.1.1 Text Formatting

At its most basic, Markdown is used to declare text formatting options. You do this by adding special symbols (punctuation) *around* the text you wish to "mark". For example, if you want to make text *italiced*, you would surround that text with underscores (`_`) so it looks like `_italicized text_`. You can see how this looks in the below example (code on the left, rendered version on the right):



```
This is a paragraph in which we'll add
**bold text**, _italicized text_, and `code`
into the middle of a sentence
```

This is a paragraph in which we'll add **bold text**, *italicized text*, and `code` into the middle of a sentence

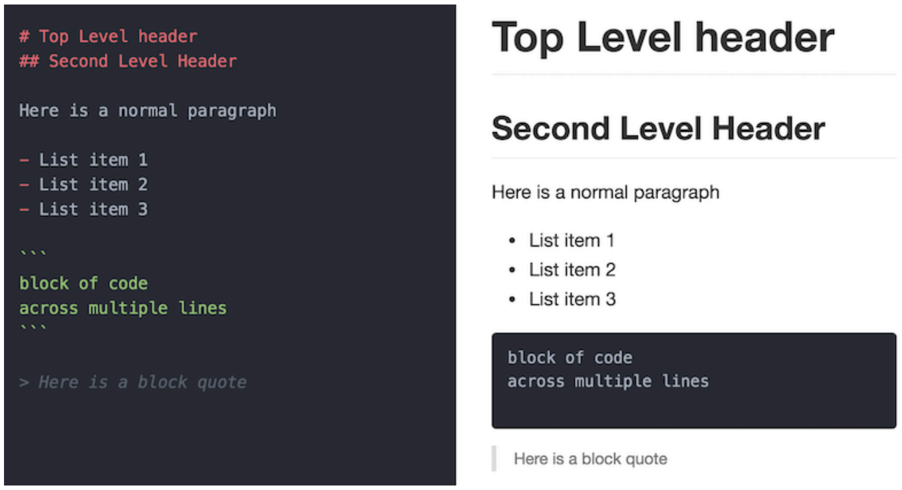
Figure 3.1: Markdown text formatting.

There are a few different ways you can format text:

Syntax	Formatting
`_text_`	*italicized* with underscores
`**text**`	**bolded** with two asterisks
`` `text` ``	inline `code` with backticks

3.1.2 Text Blocks

But Markdown isn't just about adding **bold** and *italics* in the middle of text—it also enables you to create distinct blocks of formatted content (such as a header or a chunk of code). You do this by adding a single symbol in front of the text. Consider the below example:



~~~
# Top Level header
## Second Level Header

Here is a normal paragraph

- List item 1
- List item 2
- List item 3

```
block of code
across multiple lines
```

> Here is a block quote
~~~

Top Level header

Second Level Header

Here is a normal paragraph

- List item 1
- List item 2
- List item 3

```
block of code
across multiple lines
```

> Here is a block quote

Figure 3.2: Markdown block formatting.

As you can see, the document (right) is produced using the following Markdown shorthand:

Syntax	Format
`#`	Header (use `##` for 2nd-level, `###` for 3rd, etc.)
```` ``` ````	Code section (3 back ticks)
`-`	Bulleted/unordered lists (hyphens)
`>`	Block quote

And as you might have guessed from this document, Markdown can even make tables, create hyperlinks, and include images!

For more thorough lists of Markdown options, see the resources linked below.

Note that Slack will allow you to use Markdown as well, though it has slightly different syntax. Luckily, the client gives you hints about what it supports:

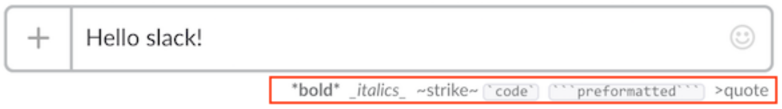


Figure 3.3: Markdown in Slack.

3.2 Rendering Markdown

In order to view the *rendered* version of your Markdown-formatted syntax, you need to use a program that converts from Markdown into a formatted document. Luckily, GitHub will automatically render your Markdown files (which end with the **`.md`** extension), and Slack or StackOverflow will automatically format your messages.

However, it can be helpful to preview your rendered Markdown before posting code. The best way to do this is to write your marked code in a text-editor that supports preview rendering, such as **Atom**.

- To preview what your rendered content will look like, simply open a Markdown file (`.md`) in Atom. Then use the command palette (or the shortcut `ctrl-shift-m`) to toggle the **Markdown Preview**. And once this preview is open, it will automatically update to reflect any changes to the text!
- Note that you can also use the command palette to **Toggle Github Style** for the Markdown preview; this will make the rendered preview look the same as it will when uploaded to GitHub!

Other options for rendering Markdown include:

- Many editors (such as Visual Studio Code) include automatic Markdown rendering, or have extensions to provide that functionality.
- Stand-alone programs such as Macdown (Mac only) will also do the same work, often providing nicer looking editor windows.
- There are a variety of online Markdown editors that you can use for practice or quick tests. Dillinger is one of the nicer ones, but there are plenty of others if you're looking for something more specific.

- There are also a number of Google Chrome Extensions that will render Markdown files for you. For example, Markdown Reader, provides a simple rendering of a Markdown file (note it may differ slightly from the way GitHub would render the document). Once you've installed the Extension, you can drag-and-drop a `.md` file into a blank Chrome tab to view the formatted document. Double-click to view the raw code.

Resources

- Original Markdown Source
- GitHub Markdown Basics
- Slack Markdown
- StackOverflow Markdown

Chapter 4

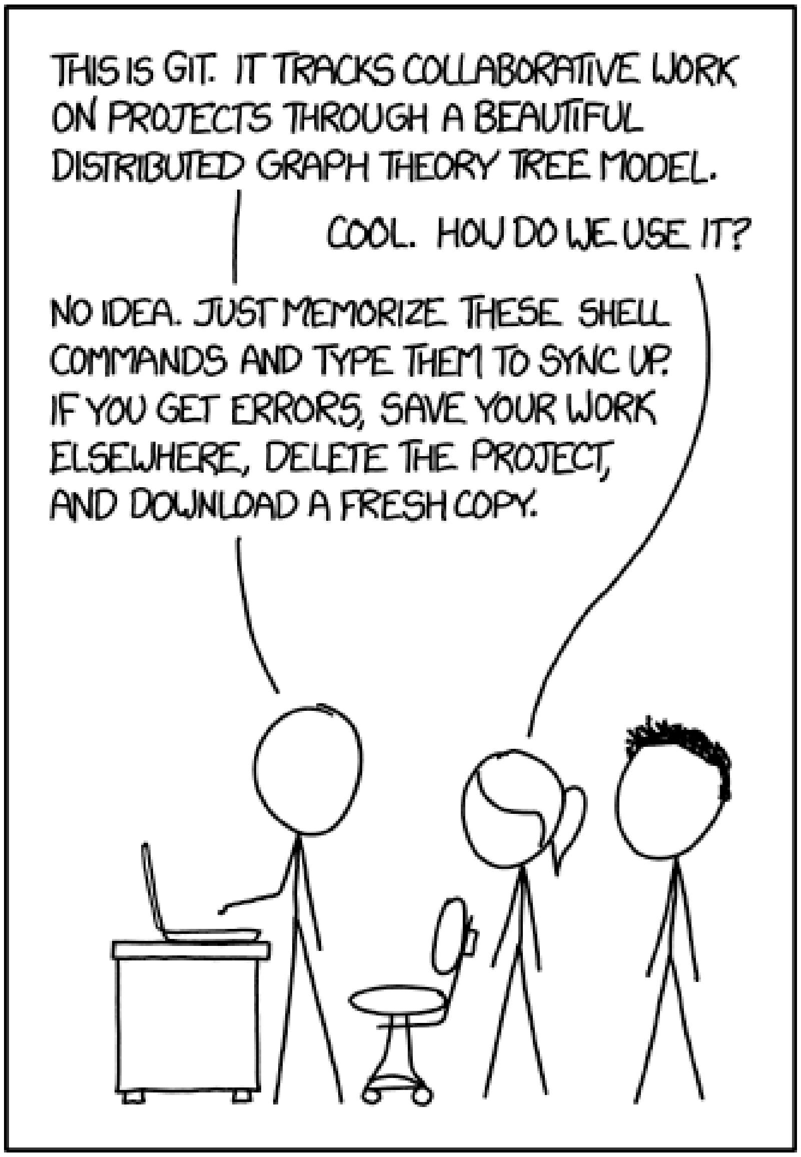
Git and GitHub

A frightening number of people still email their code to each other, have dozens of versions of the same file, and lack any structured way of backing up their work for inevitable computer failures. This is both time consuming and error prone.

And that is why they should be using **git**.

This chapter will introduce you to `git` command-line program and the GitHub cloud storage service, two wonderful tools that track changes to your code (`git`) and facilitate collaboration (GitHub). Git and GitHub are the industry standards for the family of tasks known as **version control**. Being able to manage changes to your code and share it with others is one of the most important technical skills a programmer can learn, and is the focus of this (lengthy) chapter.

4.1 What is this *git* thing anyway?


THIS IS GIT. IT TRACKS COLLABORATIVE WORK ON PROJECTS THROUGH A BEAUTIFUL DISTRIBUTED GRAPH THEORY TREE MODEL.
COOL. HOW DO WE USE IT?
NO IDEA. JUST MEMORIZE THESE SHELL COMMANDS AND TYPE THEM TO SYNC UP. IF YOU GET ERRORS, SAVE YOUR WORK ELSEWHERE, DELETE THE PROJECT, AND DOWNLOAD A FRESH COPY.

Git is an example of a **version control system**. Eric Raymond defines version control as

> A version control system (VCS) is a tool for managing a collection of program code that provides you with three important capabilities: **reversibility**, **concurrency**, and **annotation**.

Version control systems work a lot like Dropbox or Google Docs: they allow multiple people to work on the same files at the same time, to view and "roll back" to previous versions. However, systems like git different from Dropbox in a couple of key ways:

1. New versions of your files must be explicitly "committed" when they are ready. Git doesn't save a new version every time you save a file to disk. That approach works fine for word-processing documents, but not for programming files. You typically need to write some code, save it, test it, debug, make some fixes, and test again before you're ready to save a new version.
2. For text files (which most all programming files are), git tracks changes *line-by-line*. This means it can easily and automatically combine changes from multiple people, and gives you very precise information what what lines of code changes.

Like Dropbox and Google Docs, git can show you all previous versions of a file and can quickly rollback to one of those previous versions. This is often helpful in programming, especially if you embark on making a massive set of changes, only to discover part way through that those changes were a bad idea (we speak from experience here).

But where git really comes in handy is in team development. Almost all professional development work is done in teams, which involves multiple people working on the same set of files at the same time. Git helps the team coordinate all these changes, and provides a record so that anyone can see how a given file ended up the way it did.

There are a number of different version control systems in the world, but git is the de facto standard—particularly when used in combination with the cloud-based service GitHub.

4.1.1 Git Core Concepts

To understand how git works, you need to understand its core concepts. Read this section carefully, and come back to it if you forget what these terms mean.

- **repository (repo)** A database containing all the committed versions of all your files, along with some additional metadata, stored in a hidden subdirectory named `.git` within your project directory. If you want to sound cool and in-the-know, call this a "repo."
- **commit** A set of file versions that have been added to the repository (saved in the database), along with the name of the person who did the commit, a message describing the commit, and a timestamp. This extra tracking information allows you to see when, why, and by whom changes were made to a given file. Committing a set of changes creates a "snapshot" of what that work looks like at the time—it's like saving the files, but more so.

- **remote** A link to a copy of this same repository on a different machine. Typically this will be a central version of the repository that all local copies on your various development machines point to. You can push (upload) commits to, and pull (download) commits from, a remote repository to keep everything in sync.
- **merging** Git supports having multiple different versions of your work that all live side by side (in what are called **branches**), whether those versions are created by one person or many collaborators. Git allows the commits saved in different versions of the code to be easily *merged* (combined) back together without you needing to manually copy and paste different pieces of the code. This makes it easy to separate and then recombine work from different developers.

4.1.2 Wait, but what is GitHub then?

Git was made to support completely decentralized development, where developers pull commits from each other's machines directly. But most professional teams take the approach of creating one central repository on a server that all developers push to and pull from. This repository contains the authoritative version the source code, and all deployments to the "rest of the world" are done by downloading from this centralized repository.

Teams can setup their own servers to host these centralized repositories, but many choose to use a server maintained by someone else. The most popular of these in the open-source world is GitHub. In addition to hosting centralized repositories, GitHub also offers other team development features, such as issue tracking, wiki pages, and notifications. Public repositories on GitHub are free, but you have to pay for private ones.

In short: GitHub is a site that provides as a central authority (or clearing-house) for multiple people collaborating with git. Git is what you use to do version control; Github is one possible place where repositories of code can be stored.

4.2 Installation & Setup

This chapter will walk you through all the commands you'll need to do version control with git. It is written as a "tutorial" to help you practice what you're reading!

If you haven't yet, the first thing you'll need to do is install git. You should already have done this as part of setting up your machine.

You'll need configure the installation, telling git who you are so you can commit changes to a repository. You can do this by using the `git` command with the `config` option (e.g., running the `git config` command):

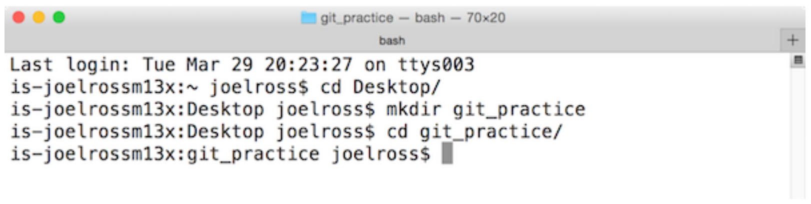
```
# enter your full name (without the dashes)
git config --global user.name "your-full-name"

# enter your email address (the one associated with your GitHub account)
git config --global user.email "your-email-address"
```

Setting up an SSH key for GitHub on your own machine is also a huge time saver; just follow the instructions on that page.

4.2.1 Creating a Repo

The first thing you'll need in order to work with git is to create a **repository**. A repository acts as a "database" of changes that you make to files in a directory.

In order to have a repository, you'll need to have a directory of files. Create a new folder `git_practice` on your computer's Desktop. Since you'll be using the command-line for this course, you might as well practice creating a new directory with that:



```
Last login: Tue Mar 29 20:23:27 on ttys003
is-joelrossm13x:~ joelross$ cd Desktop/
is-joelrossm13x:Desktop joelross$ mkdir git_practice
is-joelrossm13x:Desktop joelross$ cd git_practice/
is-joelrossm13x:git_practice joelross$
```

Figure 4.1: Making a folder with the command-line.

You can turn this directory *into* a repository by telling the `git` program to run the `init` action:

```
# run IN the directory of project (make sure pwd is correct!)
git init
```

This creates a new *hidden* folder called `.git` inside of the current directory (it's hidden so you won't see it in Finder, but if you use `ls -a` (list with the **a**ll option) you can see it there). This folder is the "database" of changes that you will make—git will store all changes you commit in this folder. The presence of the `.git` folder causes that directory to become a repository; we refer to the whole directory as the "repo" (an example of synechdoche).

- Note that because a repo is a single folder, you can have lots of different repos on your machine. Just make sure that they are in separate folders; folders that are *inside* a repo are considered part of that repo, and trying to treat them as a separate repository causes unpleasantness. **Do not put one repo inside of another!**

4.2.2 Checking Status

Now that you have a repo, the next thing you should do is check its **status**:

```
git status
```

The `git status` command will give you information about the current "state" of the repo. For example, running this command tells us a few things:

- That you're actually in a repo (otherwise you'll get an error)
- That you're on the `master` branch (think: line of development)
- That you're at the initial commit (you haven't committed anything yet)
- That currently there are no changes to files that you need to commit (save) to the database
- *What to do next!*

That last point is important. Git status messages are verbose and somewhat awkward to read (this is the command-line after all), but if you look at them carefully they will almost always tell you what command to use next.

If you are ever stuck, use `git status` to figure out what to do next!

This makes `git status` the most useful command in the entire process. Learn it, use it, love it.

4.3 Making Changes

Since `git status` told you to create a file, go ahead and do that. Using your favorite editor, create a new file `books.md` inside the repo directory. This Markdown file should contain a *list* of 3 of your favorite books. Make sure you save the changes to your file to disk (to your computer's harddrive)!

4.3.1 Adding Files

Run `git status` again. You should see that git now gives a list of changed and "untracked" files, as well as instructions about what to do next in order to save those changes to the repo's database.

The first thing you need to do is to save those changes to the **staging area**. This is like a shopping cart in an online store: you put changes in temporary storage before you commit to recording them in the database (e.g., before hitting "purchase").

We add files to the staging area using the `git add` command:

```
git add filename
```

(Replacing `filename` with the name/path of the file/folder you want to add). This will add a single file *in its current saved state* to the staging area. If you change the file later, you will need to re-add the updated version.

You can also add all the contents of the directory (tracked or untracked) to the staging area with:

```
git add .
```

(This is what I tend to use, unless I explicitly don't want to save changes to some files.)

Add the `books.md` file to the staging area. And of course, now that you've changed the repo (you put something in the staging area), you should run `git status` to see what it says to do. Notice that it tells you what files are in the staging area, as well as the command to *unstage* those files (remove them from the "cart").

4.3.2 Committing

When you're happy with the contents of your staging area (e.g., you're ready to purchase), it's time to **commit** those changes, saving that snapshot of the files in the repository database. We do this with the `git commit` command:

```
git commit -m "your message here"
```

The `"your message here"` should be replaced with a short message saying what changes that commit makes to the repo (see below for details).

WARNING: If you forget the `-m` option, git will put you into a command-line *text editor* so that you can compose a message (then save and exit to finish the commit). If you haven't done any other configuration, you might be dropped into the ***vim*** editor. Type **`:q`** (**colon** then **q**) and hit enter to flee from this horrid place and try again, remembering the `-m` option! Don't panic: getting stuck in *vim* happens to everyone.

4.3.2.1 Commit Message Etiquette

Your commit messages should be informative about what changes the commit is making to the repo. `"stuff"` is not a good commit message. `"Fix critical authorization error"` is a good commit message.

Commit messages should use the **imperative mood** (`"Add feature"` not `"added feature"`). They should complete the sentence:

> If applied, this commit will **{your message}**

Other advice suggests that you limit your message to 50 characters (like an email subject line), at least for the first line—this helps for going back and looking at previous commits. If you want to include more detail, do so after a blank line.

A specific commit message format may also be required by your company or project team. See this post for further consideration of good commit messages.

Finally, be sure to be professional in your commit messages. They will be read by your professors, bosses, coworkers, and other developers on the internet. Don't join this group.

After you've committed your changes, be sure and check `git status`, which should now say that there is nothing to commit!

4.3.3 Commit History

You can also view the history of commits you've made:

```
git log [--oneline]
```

This will give you a list of the *sequence* of commits you've made: you can see who made what changes and when. (The term **HEAD** refers to the most recent commit). The optional `--oneline` option gives you a nice compact version. Note that each commit is listed with its SHA-1 hash (the random numbers and letters), which you can use to identify each commit.

4.3.4 Reviewing the Process

This cycle of "edit files", "add files", "commit changes" is the standard "development loop" when working with git.

In general, you'll make lots of changes to your code (editing lots of files, running and testing your code, etc). Then once you're at a good "break point"—you've got a feature working, you're stuck and need some coffee, you're about to embark on some crazy changes—you will add and commit your changes to make sure you don't lose any work and you can always get back to that point.

4.3.4.1 Practice

For further practice using git, perform the following steps:

1. **Edit** your list of books to include two more books (top 5 list!)
2. **Add** the changes to the staging area
3. **Commit** the changes to the repository

Be sure and check the status at each step to make sure everything works!

4.4 GitHub and Remotes

Now that you've gotten the hang of git, let's talk about GitHub. GitHub is an online service that stores copies of repositories in the cloud. These repositories can be *linked* to your **local** repositories (the one on your machine, like you've been working with so far) so that you can synchronize changes between them.

- The relationship between git and GitHub is the same as that between your camera and Imgur: **git** is the program we use to create and manage repositories; GitHub is simply a website that stores these repositories. So we use git, but upload to/download from GitHub.

Repositories stored on GitHub are examples of **remotes**: other repos that are linked to your local one. Each repo can have multiple remotes, and you can synchronize commits between them.

Each remote has a URL associated with it (where on the internet the remote copy of the repo can be found), but they are given "alias" names (like browser bookmarks). By convention, the remote repo stored on GitHub's servers is named **`origin`**, since it tends to be the "origin" of any code you've started working on.

Remotes don't need to be stored on GitHub's computers, but it's one of the most popular places to put repos.

4.4.1 Forking and Cloning

In order to use GitHub, you'll need to **create a free GitHub account**, which you should have done as part of setting up your machine.

Next, you'll need to download a copy of a repo from GitHub onto your own machine. **Never make changes or commit directly to GitHub**: all development work is done locally, and changes you make are then uploaded and *merged* into the remote.

Start by visiting **https://github.com/info201/github_practice**. This is the web portal for an existing repository. You can see that it contains one file (`README.md`, a Markdown file with a description of the repo) and a folder containing a second file. You can click on the files and folder to view their source online, but again you won't change them there!

Just like with Imgur or Flickr or other image-hosting sites, each GitHub user has their own account under which repos are stored. The repo linked above is under the course book account (`info201`). And because it's under our user account, you won't be able to modify it—just like you can't change someone else's picture on Imgur. So the first thing you'll need to do is copy the repo over to *your **own** account on GitHub's servers*. This process is called **forking** the repo (you're creating a "fork" in the development, splitting off to your own version).

- To fork a repo, click the **"Fork"** button in the upper-right of the screen:

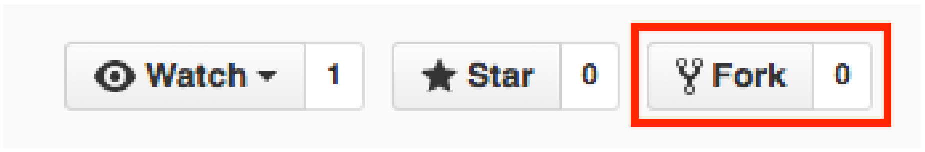


Figure 4.2: The fork button on GitHub's web portal.

  This will copy the repo over to your own account, so that you can upload and download changes to it!

  Students in the INFO 201 course will be forking repos for class and lab execises, but *not* for homework assignments (see below)

Now that you have a copy of the repo under your own account, you need to download it to your machine. We do this by using the `clone` command:

```
git clone [url]
```

This command will create a new repo (directory) *in the current folder*, and download a copy of the code and all the commits from the URL you specify.

- You can get the URL from the address bar of your browser, or you can click the green "Clone or Download" button to get a popup with the URL. The little icon will copy the URL to your clipboard. **Do not** click "Open in Desktop" or "Download Zip".
- Make sure you clone from the *forked* version (the one under your account!)

  Warning also be sure to `cd` out of the `git_practice` directory; you don't want to `clone` into a folder that is already a repo; you're effectively creating a *new* repository on your machine here!

Note that you'll only need to `clone` once per machine; `clone` is like `init` for repos that are on GitHub—in fact, the `clone` command *includes* the `init` command (so you do not need to init a cloned repo).

4.4.2 Pushing and Pulling

Now that you have a copy of the repo code, make some changes to it! Edit the `README.md` file to include your name, then `add` the change to the staging area and `commit` the changes to the repo (don't forget the `-m` message!).

Although you've made the changes locally, you have not uploaded them to GitHub yet—if you refresh the web portal page (make sure you're looking at the one under your account), you shouldn't see your changes yet.

In order to get the changes to GitHub, you'll need to `push` (upload) them to GitHub's computers. You can do this with the following command:

```
git push origin master
```

This will push the current code to the `origin` remote (specifically to its `master` branch of development).

- When you cloned the repo, it came with an `origin` "bookmark" to the original repo's location on GitHub!

Once you've **pushed** your code, you should be able to refresh the GitHub webpage and see your changes to the README!

If you want to download the changes (commits) that someone else made, you can do that using the `pull` command, which will download the changes from GitHub and *merge* them into the code on your local machine:

```
git pull
```

Because you're merging as part of a `pull`, you'll need to keep an eye out for **merge conflicts**! These will be discussed in more detail in chapter 14.

Pro Tip: always `pull` before you `push`. Technically using `git push` causes a merge to occur on GitHub's servers, but GitHub won't let you push if that merge might potentially cause a conflict. If you `pull` first, you can make sure your local version is up to date so that no conflicts will occur when you upload.

4.4.3 Reviewing The Process

Overall, the process of using git and GitHub together looks as follows:

4.5 Course Assignments on GitHub

For students in INFO 201: While class and lab work will use the "fork and clone" workflow described above, homework assignments will work slightly differently. Assignments in this course are configured using GitHub Classroom, which provides each student *private* repo (under the class account) for the assignment.

Each assignment description in Canvas contains a link to create an assignment repo: click the link and then **accept the assignment** in order to create your own code repo. Once the repository is created, you should **`clone`** it to your local machine to work. **Do not fork your asssignment repo**.

DO NOT FORK YOUR ASSIGNMENT REPO.

After `cloning` the assignment repo, you can begin working following the workflow described above:

1. Make changes to your files

2. **Add** files with changes to the staging area (`git add .`)
3. **Commit** these changes to take a repo (`git commit -m "commit message"`)
4. **Push** changes back to GitHub (`git push origin master`) to turn in your work.

Repeat these steps each time you reach a "checkpoint" in your work to save it both locally and in the cloud (in case of computer problems).

4.6 Command Summary

Whew! You made it through! This chapter has a lot to take in, but really you just need to understand and use the following half-dozen commands:

- `git status` Check the status of a repo
- `git add` Add file to the staging area
- `git commit -m "message"` Commit changes
- `git clone` Copy repo to local machine
- `git push origin master` Upload commits to GitHub
- `git pull` Download commits from GitHub

Using git and GitHub can be challenging, and you'll inevitably run into issues. While it's tempting to ignore version control systems, **they will save you time** in the long-run. For now, do your best to follow these processes, and read any error messages carefully. If you run into trouble, try to understand the issue (Google/StackOverflow), and don't hesitate to ask for help.

Resources

- Git and GitHub in Plain English
- Atlassian Git Tutorial
- Try Git (interactive tutorial)
- GitHub Setup and Instructions
- Official Git Documentation
- Git Cheat Sheet