The objective of this section is to briefly introduce you to the version control system Git and the repository website Github. The exposition is very brief and need-to-know - for a great tutorial on the many ins and outs of Git, check out `https://www.atlassian.com/git/tutorials/`. There is also an excellent book that's free online at `http://git-scm.com/book/en/v2`. Chapters 1-6 cover the essentials well.

Firstly, while we will cover working with Github, you will likely want to use an alternative site, Bitbucket, when working with code you want kept private. As far as I know, the functionality of the two sites is basically the same.[1]

Last section

First, let's look at an issue that came up last week. A few people had trouble with an error after using `setwd()` twice. For example, people used the RStudio navigation trick to call:

```
setwd("FirstDir/SecondDir/ThirdDir")
```

Then, when later sourcing their file, they called the same function again and got an error. What happened? This stems from the use of relative vs absolute file paths. What I taught you, and what you should almost always use, is relative file paths. These are paths that are "relative" to the current working directory. You can tell, because the first directory does not begin with "`/`" or "`C:/`".

If you have already called `setwd("FirstDir/SecondDir/ThirdDir")`, and you try to again, `R` will search for `FirstDir` within `ThirdDir`, find it's not there, and give you an error. Using `setwd()` within scripts can be dangerous. Remember, you can call things like

```
read.dta("DataFolder/auto.dta")
```

which reads in a file from a subfolder of the working directory. It is seldom necessary to call `setwd()` from within a script.

What is version control?

Version control refers to the management of changes to documents or programs. The main value of version control for us is the deliberate staging of our code so that we can look back at previous versions that are saved when we know they work. We can easily use a service like Dropbox to back things up, and save different versions, but we end up with far too many files going back for it to be very helpful if we don't remember the exact date we want to go back to. Dropbox is also much more awkward to use for this purpose.

Crucially, one needs a version control system to look back at previous states of the *entire project*. Dropbox, only allows the user to look back at old versions of individual files.

The use of an online repository system like Github or Bitbucket can also make collaboration quite easy to manage.

You may end up opting not to use an explicit versioning system if you deem the setup costs not worth the future benefits. If so, I urge you to, at the very least, use an automatic online backup.

[1] Private repositories on Bitbucket are free, but not on Github.

Downloading Git

You will start by downloading and installing Git for your operating system from the following link:

`http://git-scm.com/downloads`

There are several GUI-based clients that you may wish to use:

`http://git-scm.com/downloads/guis`

I won't cover any of these, as I know none of them well.[2] All Git commands can be accessed using the command line tool, so this is all you need to know. If you learn the command line tool first, it's highly likely that you'll easily understand a GUI based tool. If you find one you really like, that makes your life much easier, especially one that deals with conflicts well, please let me know.

For this section, you will also need a Github account. You can sign up at:

`https://github.com/`

Once you have signed up, you will also need to fork my repository for this section. To do this, sign in to Github, visit:

`https://github.com/ARE212TestKenny/Section2`,

and click the `Fork` button in the top right corner.

Starting and Configuring Git

Once Git is installed, it should run directly from the terminal in Mac or Linux, or using the Bash shell emulator that gets installed with Git on Windows (named "Git Bash" on my installation – this is just an emulator of a Linux style terminal). Git, as we're using it, is a command line utility, so doesn't require its own window as such.

All commands discussed in this section are to be typed into an open terminal.

The first job to do when you install Git is to set your user name and email. To do this type:

```
$ git config --global user.name "Your Name"
$ git config --global user.name are212ers@example.com
```

Next, we need to configure a text editor. Mac users can use TextEdit, Windows users should download something other than Notepad as it messes up reading `RETURN` characters sometimes. I use Emacs, which you can download from:

`http://vgoulet.act.ulaval.ca/en/emacs/windows/`

Another one people like is NotePad++:

`http://notepad-plus-plus.org/download/v6.7.4.html`

The command for configuring the text editor is then, for example:

```
$ git config --global core.editor emacs
```

If you get an error with Notepad++, try something like the following (all one line):

[2] I have used the built in RStudio commands a couple of times.

```
$ git config --global core.editor "'C:/Program Files (x86)/Notepad++/notepad++.exe'
    -multiInst -notabbar -nosession -noPlugin"
```

In this section, we won't actually make use of this text editor, as it's one extra thing that might not work for someone in class. When we "commit", you'll be able to check and see if it worked.

Starting a repository locally

There are two ways to get a repository started locally on your computer. The first begins with opening a terminal and navigating to a folder in which you want to start tracking files. To navigate folders you type:

```
$ cd Folder1/Folder2/Folder3
```

You can use the RStudio file navigation trick (use `TAB`) to help you remember the file path to where you want to go. You can also use the `ls` command to browse the files and folders in the current directory.

Now, you can initialize a repository using the command:

```
$ git init
```

This will create a subfolder named `.git` which contains absolutely everything Git needs for your project (this may be hidden - if you care to, google how to look at hidden folders on your system). What you see in your file system (outside the `.git` folder) is what's referred to as the "working directory"; this is the current version of the project that is being "checked out".

For now, we won't use `git init`. The other way to begin a repository is to clone one from another location. Most commonly, you will be cloning a repository from a remote location online. Let's do this. In the terminal, navigate to the folder one above the location you want to clone the repository into and type the following:

```
$ git clone https://github.com/<yourGithubUsername>/Section2 yourFolderName
```

This will create the new folder `yourFolderName` with your fork of the `Section2` repository in it. NOTE: you need to have forked my `Section2` repository on Github. Let's look a bit at what's happened. Navigate to `yourFolderName` in the terminal and use `ls` to list the files. You should see the following file names:

```
LICENSE  List of names.txt
```

Now use `ls -a` to look at the hidden files as well. You should now see the aforementioned `.git` folder where all the action happens. Let's not worry about what's inside this for now. Try the following commands individually:

```
$ git status
$ git log
```

You should be able to exit the `log` display by hitting `q`. You should read more about these commands in the book/tutorial - especially all the options for the log command. For example, you may prefer to use something like:

```
$ git log --pretty=oneline --abbrev-commit
```

Now, open the `List of names.txt` file using your favorite text editor, add your name, and save the file. Back in the terminal, let's do a `git status` again.

Now we can see a bit more about what `git status` tells us. You should see that the `List of names.txt` file has been changed but is "not staged for commit".

> *A commit refers to one of the deliberate stages of your project which you may want to revert to at some time. Think of it like a "mini-version".*[3]

Now let's "stage" the file we just changed. To do this, we type:

```
$ git add List\ of\ names.txt
```

Try a `git status` again. We can now see our changes are ready to be committed. To commit, we type:

```
$ git commit -m "Your commit message here"
```

Now, try a `git status` once more. We can see we have a clean working directory, but we're ahead of the branch `origin/master`, which refers to the online repository's (named `origin`) version of the branch `master`.

> *A branch is technically a movable pointer to a commit. Because each commit points to its parent(s), you may think of it as a list of commits defining a version history. Each project can have several branches.*

Now, let's create a new file in the working directory. Any way you know how, create a new text file, put some text in it, and save it using some unique name.

Let's do another `git status`. We can now see we have "untracked files" to be added, if we wish. We also add this new file using `git add`. This command will also stage the file. At the same time, let's do another commit.

```
$ git add yourFileName.txt
$ git commit -m "Adding a new file"
```

Next, we'll upload (or "push") our new version of the project to Github. To do this, we type:

```
$ git push origin master
```

and you input your user name and password as per the prompts. Note that `origin` and `master` are only used as these are the default names for the remote and local branches, respectively.

This completes the typical work flow of Git with Github! Let's stop for a moment and think about what we've done using this great figure from `http://git-scm.com/book/en/v2`.

[3] In practice, you numbered "versions" you see will be a subset of the commits in a project.

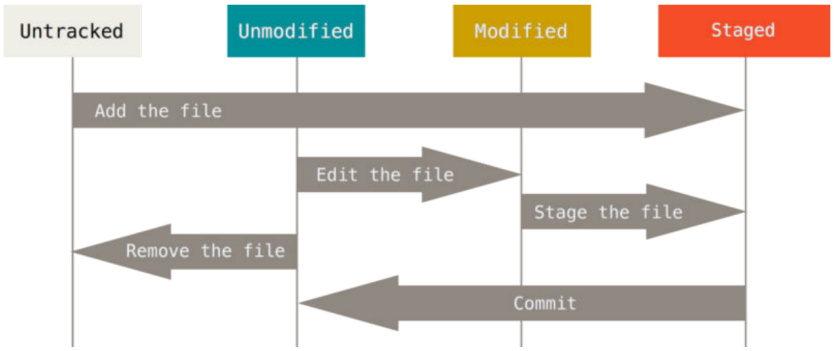

FIGURE 2-1

The lifecycle of the status of your files.

The figure illustrates the different stages a file can go through in the Git system. Firstly, files which are not tracked by Git are known as "untracked"; our new file that we created was untracked before we added it to the project. Unmodified files are part of the most recent commit and haven't changed; this was our `List of names.txt` file prior to us adding our name. Modified files are files that were part of a previous commit (or have recently been added), and have now been changed in the working directory; this was the `List of names.txt` file after we added our name. Staged files are tracked files that have been modified and added (or simply added) and are ready for a commit.

Your first pull request

This part will happen on Github. Make sure you're logged in to your account, go to the `Section2` repository and hit the `Pull Request` button just above the file list. What this does is asks my `ARE212TestKenny` account to merge in your changes to the project. I would then be able to get those changes into my working directory locally by navigating to my `ARE212TestKenny` folder in the terminal and typing:

```
$ git pull origin master
```

This command 1) fetches the new remote branch, and then 2) merges this remote branch into the master branch on my machine.

With so many people making pull requests at once, I have no idea how this is going to go. This is why ARE212 section is so much fun.

Checking out other commits

Checking out refers to changing the files in the working directory to an alternative state of the project. This can be previous versions or versions on other branches. If you want to go back to a specific commit on the current branch, you would get its short name (a 7-digit hex number) by using:

```
$ git log --oneline
```

and then checking it out using, for example:

```
$ git checkout 310e9ce
```

If you wish to begin a new branch from this point, you could then use, for example:

```
$ git branch testing
```

and then you could make commits to this branch just like the `master` branch that we've been working with. To merge this new branch back in, you would use something like:

```
$ git checkout master
$ git merge master testing
```

Any conflicts can be resolved in a text editor - Git nicely puts both versions into your file for you to compare. Your next commit will be a merge commit - one of the cases where a commit has two parents.

Last words

There's plenty more to Git. The section certainly doesn't cover enough to have the best experience with Git. For most problems you will encounter, you will be able to ask Google: "git <your problem>". Hasn't failed me yet. At the very least, you should read up on `.gitignore`, the `git add` options, and collaborative workflow with several branches (Chapter 5 in the book). I'll talk about these if we have time at the end of section.

What's my current system?

I currently use Dropbox as my main online backup system. I should probably use something more secure like Carbonite, but I recently paid for another year with Dropbox. Using an automatic backup system that keeps versions is a good compromise if you don't want too much messing around with getting started. For some more complicated projects I use Git with Github for storage and sharing. For projects I want to keep private, I use Git with BitBucket.