An Introduction to Working with Git, GitHub, and RStudio

1 What's the Point?

The point of GitHub is version control. GitHub is useful for many things, but the platform was designed with the express purpose of providing effective version control for collaborative coding projects, so that is what it's best at.

It's useful (and accurate) to think of GitHub having three different parts: the remote repository, the staging area, and the local repository. Repositories are folders dedicated to a single project.

- The remote repository is hosted by GitHub; it's kept on their servers. For us, the users, the remote repository is where we keep the latest, agreed upon, up-to-date code. Everyone can see this code.
- The local repository is hosted on our own computer. It is essentially a copy the remote repository where we can make any changes we like without affecting the code on the remote repository. Only the local user can see this code.
- The staging area is where we put changes we made to the code in our local repository that we think should be made officially to the remote repository. When code is in the staging, everyone can see the code, particularly the changes made. We can then agree on the changes that should officially be made to the remote repository.

The above is the basic workflow of GitHub. First, we copy the remote repository to our local machines. Then we make any changes we think are necessary. Changes are submitted to the staging area. Finally, we decide which changes to keep and update the remote repository.

Whenever changes are made, we document it by including a message. We can then review the changes and messages to get a history of how the code has developed.

1.1 Git v. GitHub

It may seem as if I'm using the two interchangeably, but they're not actually the same. Git is the language and source control program, while GitHub is the cloud service where we store code. We use git commands to interact with GitHub. The distinction is not terribly important, but hopefully it will clear up any confusion about why I switch back and forth between the two as we go.

2 Installing Git

Before using git, we have to install it on our machines.

For Mac :

1. Open your terminal. To do this, hit the command key and space bar and start typing terminal. When the terminal appears as the highlighted option, hit return. This will open your terminal.

2. If you don't have Homebrew installed, it's time to install it. Homebrew is a package that simplifies package installations in Mac. If you think you already have it, you can skip to the next step and see if it it works. If the next step doesn't work, come back here. To install Homebrew, simply copy and paste the following code into your terminal and hit return:

```
/bin/bash -c "$(curl -fsSL https://raw.githubusercontent.com/Homebrew/install
    /master/install.sh)" brew doctor
```

3. Now we can install git. Again, copy and paste the following code into your terminal and hit return:

```
brew install git
```

4. Git is installed!

For Windows :

1. Click here to navigate to the git homepage.
2. In the downloads box, click the Windows options. Git for Windows will automatically begin downloading.
3. Go to where the installer downloaded, and double click to start installing.
4. I recommend keeping all the default settings EXCEPT in potentially one place: your default text editor. The default in Windows is Notepad++, but if you would rather use something else (e.g., SublimeText, Sweave, vim, emacs, etc.), this is the place to change it. For everything else, it's best to stick to the defaults.
5. Once you hit finish, git is installed!

3 Terminal and Command Line Basics

The terminal is how you talk to your computer. The command line is where you speak (type). Git is a language for the command line. It is possible to use git and GitHub with a **g**raphical **u**ser **i**nterface (a GUI), but it's not worth learning. Despite any qualms or trepidation feelings you might have, the most straightforward way to use git is with the command line in your terminal. There are four general, non-git commands to know first.

`pwd` This is how you find your current working directory.

`ls` Show everything in the directory.

`cd DirectoryName` Change your directory; moves to the directory "directory_name." To go "backwards", i.e., move to the directory above the one you're in, is `cd ..`

`mkdir DirectoryName` Create a new directory called "directory_name"

These commands help you navigate around your computer. The working directory is where your code is running and (if you don't specify otherwise) where anything you save is kept. Directories are like folders used to organize your computer. For example, within your `Documents` folder, you might have two sub-folders, `Literature` and `WorkingPapers`[1]. Say `Literature` has all the articles you've downloaded in it and `WorkingPapers` is where you save all the articles you're writing that will eventually be accepted at AJPS. `Documents` is a working directory, but both `Literature` and `WorkingPapers` can be as well.

The following examples will demonstrate how to use the above four commands to see where you are, see what is in your folder/directory, and how to change between them. The symbols `~%` indicate the start of the terminal line where we begin typing. The user (we) do **not** need to type these symbols or anything prior to them. What we type is in burnt orange[2]. Key words are bold-ed. Lines without `~%` are what the computer returns. Comments are in slate gray; these are for additional clarification and should **not** be typed out by the user. The examples will go over what we type in the command line and what the computer will return.

3.1 Examples of Command Line Basics

- Checking which directory you are currently in:

```
username@computername ~ % pwd  This is an example comment; do not type words in slate gray
/Users/username  An example of what the computer might return
```

- Changing into the `Documents` directory and checking to see where we are now:

```
username@computername ~ % cd Documents
username@computername Documents ~ % pwd
/Users/username/Documents
```

- Looking to see what's inside the `Documents` directory:

```
username@computername Documents ~ % ls
Literature WorkingPapers
```

- Changing directories into the `Literature` directory:

```
username@computername Documents ~ % cd Literature
username@computername Literature ~ % pwd
/Users/username/Documents/Literature
```

- Now say we want to go into our `WorkingPapers` directory. First, we need to go back to the `Documents` directory. Then, we can go into the `WorkingPapers` directory.

[1] If you're going to be working from the terminal, it's good practice to make filenames without spaces. Spaces have meaning in the command line, so if the name of you file you want to call from the command line has spaces in it, the terminal will get confused.

[2] Burnt orange is defined at this useful website.

```
username@computername Literature ~ % cd ..
username@computername Documents ~ % cd WorkingPapers
username@computername WorkingPapers ~ % pwd
/Users/username/Documents/WorkingPapers
```

- We can actually do the above in one step. I'll demonstrate this by switching back to the `Literature` folder from the `WorkingPapers` folder.

```
username@computername WorkingPapers ~ % cd ../Literature
username@computername Literature ~ % pwd
/Users/username/Documents/Literature
```

- Finally, say you want to create a new folder in `Literature` just for articles relating to MENA called `MENApapers`.

```
username@computername Literature ~ % mkdir MENApapers
username@computername Literature ~ % ls
MENApapers
```

- Bonus: you can look inside a directory you're not currently in, as long as you tell your computer where to look. For example, if we were back in the `Documents` directory, we could ask the computer what's in the `Literature` directory.

```
username@computername Literature ~ % cd ..
username@computername Documents ~ % ls Literature/
MENApapers
```

4 Git and RStudio

A note on jargon: So far I've talked a lot about directories and folders. They can essentially be used interchangeably. In git-lingo they're called *repositories*. This makes sense, because you are storing, or repositing, your code on GitHub. The place you reposit your code is a repository. In practice, you navigate repositories just like directories. Cool kids call them "repos."

4.1 Git Setup

Now that we're all comfortable using the command line, we can get started with git. The first thing we want to do is set our global environments. We do this so whenever we checkout a repository[3] or submit changes to a file, our name is associated with it. Then, anyone can see who changed what. The global environments we want to set are our user name and email. The commands to do this follow:

```
~ % git config --global user.name "User Name"  Enter your own user name in quotes
~ % git config --global user.email "user.email@princeton.edu"  Enter the email associated with your
    GitHub account
```

[3] Information on this coming soon...