Version control mini-bootcamp using Git

Created by Devin W. Silvia (September 2018)

Before we get started, note that any line that follow "$" will be a command that you enter on the command line.

With that said, let's begin.

PART ONE: Creating and maintaining a local repository

First, let's get create a "code" directory on our machine so that we can set up a new repository for testing out git. Once we create it, we jump into that directory

Navigate to the directory where you want to store your code and then:

```
$ mkdir code
```

```
$ cd code
```

Now, create a new directory to use for testing git and move into it:

```
$ mkdir git_test
```

```
$ cd git_test
```

Git is called using the command "git". To see what you can do with git, you can run "git help".

Now, let's we need to initialize our repository:

```
$ git init
```

This creates a special hidden directory called ".git", you can see it by running "ls -a". In this directory, git stores information about the repository and tracks changes. Generally, you won't have to do anything inside this directory.

By default, the repository won't track any files until you tell it to. Files that exist in the directory, but haven't been added to the repo (short for

repository) will be marked as an "untracked file" when you type "git status". To see the full list of markers used by "git status", run "git help status".
Note, "git status" will only show information about files that have been added, changed, or removed since the last time a commit was made to the repository.

An aside on text editors

If you don't already have a text editor that you are comfortable with, we're going to pick one now. Common ones include vi/vim and emacs, but there are many others. If you don't have a preference, I encourage you to try using Atom for now. To make sure that git uses the editor you want, you will need to run one of the following commands (or something similar for you editor of choice).

To use nano:

```
$ git config --global core.editor "nano"
```

To use Atom:

```
$ git config --global core.editor "atom --wait"
```

To use Emacs:

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

It is important to understand that you must have these text editors installed on your computer for these to work. If you don't want to use any of these you'll want to do a search to figure out how to correctly set the editor to the one that you want.

Continuing the tutorial…

Now, let's create a file and then add it to the repo (while running a few other commands along the way):

```
$ touch dummy_file.txt
```

(Note: "touch" is just an easy way to create an empty file)

```
$ git status
```

```
$ git add dummy_file.txt
```

```
$ git status
```

```
$ git commit
```

You should get an editor window that expects you to record a commit message. *NOTE*: You may not be familiar with the editor window that pops up if you didn't set it above. The common default is vi/vim. If you're not familiar with how to use this, you might have to google how to use vi/vim or just change the editor to something else. If you want to avoid the text editor all together, you can input your commit message on the command line:

```
$ git commit –m "Your commit message goes here."
```

You may wish to us `–m` for all future commits to avoid the text editor, unless you're writing a really long message! Commit messages are very important for keeping track of what changes when into a particular commit. Even though it is very tempting to be lazy with commit messages, detailed commit messages will be more useful to you down to the road than something along the lines "added more stuff to the repo".

USE QUALITY COMMIT MESSAGES.

For example, I might put something like:

> "Added the first file to the repository. This file is a dummy file that contains no information but was created for the purpose of testing out git."

Note, it is useful to put a short, yet descriptive, one-liner at the beginning of a commit and then put a more detailed explanation below. The first line will be used by git in the output of "git log" and can useful for keeping the log short but informative.

As I just suggested, to see the current history of the repo, we can run:

```
$ git log
```

As the number of commits build up, the output of "git log" will get long. The log outputs revision information from the most recent to the least recent. You should be able to hit enter to step through the git long in your terminal window.

Revision numbers are useful for moving between different versions of the code, we'll see an example of that in a bit.

Just to make sure we've locked down the basics of committing, let's add another file, but this time, open it using your preferred text editor (again, I'll be using emacs) and put in some information:

```
$ emacs not_just_a_dummy_file.txt
```

(Note: you can use your editor of choice here)

Make sure to write a couple sentences and then save and exit the file. Then:

```
$ git add not_just_a_dummy_file.txt
```

```
$ git commit
```

```
$ git log
```

At this point, we can see the commits we've made and everything should be looking good. Now, let's open up that file we just commited and change something:

```
$ emacs not_just_a_dummy_file.txt
```

Add, remove, or edit something in the file. Save and exit. Then:

```
$ git status
```

Git is showing that we've modified that file. That's great, but what if we want to see exactly what we changed? That's where "git diff" comes in:

```
$ git diff
```

Git shows us what files in the repo have been changed and how they've been changed -- very useful!

If I'm happy with the changes, I can go ahead and commit the modified file.

```
$ git commit -a
```

(Make sure to put a useful commit message!)

Note: the `-a` tells git to stage the file and commit the change at the same time. Now we can check to see what we've done again:

```
$ git log
```

OK, great, we're making and tracking changes. What else can we do?

Thanks to version control, not only do we have access to the history of our repository, but we can also go back to any previous version of the code.

Let's say we wanted to see what the code was like before our last commit. Using "git log", we can get the revision number of the previous commit and then "checkout" the version of that code.

```
$ git checkout [first 6 digits of revision number]
```

Note: the first 6 digits of the revision number are in the "git log" file.

Git tells us that we are in a detached HEAD state, meaning that we are at a previous version of the repository, specifically the version we requested. It also provides useful information about what we can do from here. Take a look at `not_just_a_dummy_file.txt`. It should look like the version you committed before you made your change.

It is important to keep in mind that we have moved to a past version of the code, if we were to make a change now and commit it, we would be changing that old version and the commit process would create a new "HEAD" of sorts. We could then later save this code development as a new branch, if we wanted. You'll learn more about heads as you work with repositories more, but for now let's just get back to the most current version of the code:

```
$ git checkout master
```

By specifying "master" we jump back to the HEAD of the master branch, which is the most recent version of the code. As an aside, it is also possible to check out the previous version of a specific file. While checking out a previous version of the entire code base doesn't change the code, checking out the prior state of a specific file does, so be aware of this.

Now, let's make a couple more changes to the code. We'll add a new file with some text in it and edit `not_just_a_dummy_file.txt`:

```
$ emacs another_file.txt
```

Write a couple things in the file, save, and exit. Then:

```
$ git add another_file.txt
```

```
$ emacs not_just_a_dummy_file.txt
```

Make a change or two, save, and exit. Then:

```
$ git status
```

Now, we can see that we added `another_file.txt` and modified `not_just_a_dummy_file.txt` (though it's technically not staged for commit).

Let's say we think about the state of our code for a bit and we decide we don't like the changes me made to `not_just_a_dummy_file.txt`, but we don't want to go back in by hand and change it back, in this case we need to "revert" the file to the state it was in at our last commit:

```
$ git checkout -- not_just_a_dummy_file.txt
```

```
$ git status
```

Now we see that we still have the added file and `not_just_a_dummy_file.txt` is no longer marked as "modified". If we're happy with the current state of our repo, we can go ahead and commit again:

```
$ git commit
```

Finally, let's say we've gotten a little carried away with all these files we've generated and we want to thin things out by getting rid of `dummy_file.txt`. We could try doing:

```
$ rm dummy_file.txt
```

```
$ git status
```

Uh oh, `dummy_file.txt` is now marked with as "deleted". This indicates that the file was deleted, but not with a git command, so it is still being tracked. We need to actually properly deleted the file and then commit the change using "git rm":

```
$ git rm dummy_file.txt
```

```
$ git commit
```

That's it for exploring what you can do with a local repository. The next section will walk you through how to interact with a web-hosted repository, specifically the one we will be using for this course.

PART TWO: Using a GitHub hosted git repository

Now we're going to clone a repository that already exists remotely.
To do this, create a directory somewhere on your computer where you're OK with putting the repository and then do the following:

```
$ git clone https://github.com/msu-cmse-courses/cmse202-F18-git-practice
```

If you were to go to that URL, you would see the repository that already exists and you should see that the properties of the repository match the one that was just cloned. The clone command downloads a copy of the current state of the repo, as well as its history, from GitHub.

Before we move on, we're going to set up a couple of git config options to ensure that git knows who we are by specifying our name and email address:

```
$ git config --global user.name "Your Name"
```

(Note, this can also just be your GitHub username)

```
$ git config --global user.email youraddress@emailserver
```

Those will set a global user name and email on your machine, but you can also set them for specific repositories by running the command without "--global" in a repository that lives on your machine.

Now that you've cloned a repository, you can make changes to it in the exact same way that we outline in Part 1. However, making changes to the version hosted on GitHub is a little bit different. Let's see how that works.

Starting from the root directory of the repository:

```
$ cd vc_testing
```

This is the directory where we will test out the process of working with remote repositories. Each of you should create new directory using your GitHub ID as the name.

To set this up for today, do the following:

```
$ mkdir <GitHub ID>
```

Note: use your personal GitHub ID where it says .

```
$ cd <GitHub ID>
```

Now create a file called `push_test.txt` in that directory. Once you've created the file, add it to the repository and commit the addition.

Did everything go smoothly? Check on the current state of your repo.

Now, if you're confident everything looks good, you're going to "push" this change to the class repo. If you're not convinced that everything is right, ask me to check!

Let's push that change:

```
$ git push
```

Git should prompt you for your GitHub username and password, once you enter it, the file you created will be pushed up to the class repo. If this goes through with out an error, check that your file is now on the web version repo and jump down to "**Pulling new changes:**" below.

Otherwise…

If you are not the first person to push a new file, you may get an "rejected" message saying that there are other changes on the remote repository. This is normal and part of collaborative repository development. If you get his message skip down to "**If you get an 'rejected' message:**" below.

Pulling new changes:

Once another student (or several) pushes their own file, we'll be able to test out how to pull changes that have been made to the remote repo.

Assuming other's have pushed files, let's pull:

```
$ git pull
```

It important to realize that for earlier versions of git, "git pull" does not

actually change your local version of the repository, it merely acquires information as to the list of changes that have been made. If you want to incorporate those changes, you have to "update" the repo with the fetch command:

```
$ git fetch
```

If you have the 2.0+ version of git, this will automatically update your repository.

You should now see other student directories in the `vc_testing` folder as well as their version of `push_test.txt`. Read through the following section to understand what would have happened if you hadn't been the first student to push your changes.

If you get an 'rejected' message:

Now, what happens if you're currently working on a repository, you've made a commit, but someone else was simultaneously working on the same repository and they pushed their changes before you did? You just encountered the result. Since git doesn't want to foolishly overwrite the work someone else has committed to the remote repo, it requires that you first pull the remote changes, merge them into your version, and then let's you push the merged version.

So, let's do that:

```
$ git pull
```

Git will pull the changes and let you know that you've pulled down a new "master". This is a result of changes that other students in the class have made. The 2.0+ version of git will automatically initiate a merge and give you the opportunity to create a unique merge command, if you want. You can simply saved the default message in this case. For older versions of git, you'll need to do the merge manually:

```
$ git merge
```

Assuming that the merge process doesn't run into any conflicts (we'll certainly run into that down the road), Git will update your code with the changes you pulled down and remind you to commit the merge. For commits involved merges, it is relatively common practice to include a commit message that says "Merged." or "Merging changes."

Do that commit now and don't forget to push your merge! (Warning: you may run into further "rejected" messages since we're all actively pushing changes, just repeat the above process until everything gets merged in.)

At this point, you should have a decent understanding of how to push and pull changes to a remote repository. If you didn't yet have to deal with a merge, you certainly will at some point, so make sure you followed the directions outlined above. At this point you should ask any questions you have about what we've done so far. If you want more practice with Git, check out the links in Part 3.

PART THREE: Explore on your own!

Now let's work through the tutorials available on Software Carpentry (http://software-carpentry.org). There is one for git, located here:

(http://swcarpentry.github.io/git-novice/)

That will walk you through git in a way similar to this short guide, but should give you a deeper understand of its functionality/power.

Good luck and happy coding!