Cheatsheet: Useful (and common) Git commands

Git is an amazing and very powerful tool that is useful for managing your projects and tools. However, because of its power and usefulness, it can be confusing for beginners because of the need to use the command line (or terminal) to run Git commands as well as the large number of commands and options available. So, we put together the commands that are the most useful and most common — and the only ones you may ever use!

Git commands:

git clone <repository> — Take an existing git repository (aka project, aka folders and files) and duplicate/copy/clone it. For example, I have a folder called `diabetesAndObesity` that contains all of my data, analysis, and manuscript on my research regarding diabetes and obesity. If you wanted to collaborate with me on the project, you would run `git clone` on my project to copy all of the files and folders and Git history into your computer.

git remote add <name> <server-name> — `remote` is the name used to describe your Git project that you want to store on the server such as Github. Think of `remote` as your external hard drive, like Dropbox, or like the cloud. If you want to save your Git project files and folders, you can create a remote for it. If you use Github or some of cloud-based service, usually they provide pretty good instructions on what to do. For example, if you wanted to save (aka `push`) your Git project files to Github, you would copy this command into you terminal/shell/Git Bash:

```
git remote add origin git://github.com/yourname/yourproject.git
```

git init — Tells Git to create a repository in the folder you are in. Basically, `git init` tells Git to start watching your folder and the files within it.

git add <files> — Add the files you want Git to commit to the history. For example, if you edited a file and you wanted to commit that file, you would run `git add filename`. Same thing applies when you create a file.

git commit — Tells Git to save your `git add`'ed files to its history. This is the main purpose and use for Git. After you commit a file(s) and typed out a commit message (be detailed about what you did!!), Git will save (or take a "snap shot" of your files and put it into its history. Once committed, it is indefinitely saved, allowing you to go back to that point at any point in time!

git log — Displays the history of your repo and the messages you added to each commit, as well as the date of the change, and who made the change. This is analogous to a logbook for those in the basic science.

git status — Displays the current state of your folder that Git is watching. `git status` shows all the files that have been edited or created since the last commit.

git push <remote-name> <your-branch> — If you have a remote/server/Github/BitBucket git repository, you can push all of your Git history to the "cloud", so you will always have a backup of your most important work, aka your research!!

git pull — Pulls the git repository from your remote/server/Github/etc. This is really useful if you work across multiple computers or if you collaborate with several people on the same research project (or other project like these workshops!).

git checkout — Allows you to go back (or forward) throughout your Git history. If you wanted to go back a few commits, you would run `git checkout commitnumber`. You get the commit number by running git log and using the first few letters and numbers of the commit. We will likely show an example in the workshop.

git branch — This is another very useful command from Git, and one of its biggest strengths! This command basically is like making a branch on a tree, letting you experiment with your files and statistical analyses without having to worry about your main files. For example, lets say I have a SAS file with my analysis codes. This file I keep in my "master" branch. If I wanted to experiment with my code to try a different analysis, I could create a new branch to play around with my SAS code by running:

```
git branch experiment
git checkout experiment
```

Now I can edit the files all I want and still have my main SAS files untouched. I can get back by running the command: `git checkout master`

git diff <commit-id (optional)> <file> — Diff shows the differences between the files in your last commit with the files you are currently editting. This is useful if you forgot what you changed so you can write better commit messages.