Git Basics Cheatsheet

You should get increasingly comfortable looking into and learning technology on your own, but this document will help you start diving into how to use `git`, which is pervasive in CS.

Basic Gitflow:

- **Connecting your remote repository and connecting it to your local repo:**
  - **`git clone <repository-url>`** - This will clone the files that exist on the remote repository onto your local machine, creating a directory named after the repository you created on github. In order to push code from the repository that you've now cloned onto your machine, you need to do the next command
  - **`git push -u origin -`** This command is essential setting up your local repo to now be able to push code to the remote repository.
- **Prepping your commit**
  - **`git status`** - check that all your files have been git added and there are no unexpected files/changes. This will display the current state of your working directory and what files you're staging in your commit. It will show untracked files (files that haven't been pushed to your git remote repository), files with changes not staged for commit, and the files that do have changes that are staged for commit.
  - **`git diff --stat`** - further make sure that you remember what is being committed, and it matches your intuition. This will show the list files that have been modified since your last commit, the number of insertions and deletions for each file, and a final summary line of the total number of lines changed, total insertions and total deletions
  - **`git diff`** - This will produce all the metadata of the files that are being compared and the exact line by line code changes between your last commit and the commit that you're staging to push. This is helpful to remind yourself what your git commit message should be (think: a bullet list of additions after a single title line).

Before you commit code, you should execute all of these commands above. You do this for two reasons. First, you want to make sure that only code modifications that you want in the commit are in it. Each commit should be relatively focused, so it is important to catch this. Second, it is a good way to remind yourself what the commit contains so that you can better summarize it in the commit message. It is also useful to know that you can execute `diff` on ranges of commits to see what changed (see `git log` to see the list of commits).

By: Isabella Dayrit Spring 2026

- Creating your commit
    - **`git add <filename>`** - This will adds this specific file/ code changes to this file to the commit that you are preparing
    - **`git commit -a`** - commit your change, entering the commit message with the first line being a high-level summary (that you'll see when you later use `git log`), and the rest being a bullet list of the changes.
    - **`git push`** - to actually update github and "submit". You should push often. If your machine dies, this will ensure that your updates are on the github servers.

General Individual Git Advice:

- When working on your own branch, it's good to commit often and insert code in smaller increments to have more iterations of development that you can revert your branch back to if a future commit causes a bug.
- The repository that you're in when creating a branch is the version that will be cloned on your new branch. It's generally best to checkout into the main branch and create your branch from there to ensure your clone has all the important things that have been merged from main. This advice will make more sense later on in the cheatsheet.

Useful Git Commands for debugging:

- **`Git restore <filename>`** - This will discard this file's uncommitted changes and restore the code in the file back to the version that existed in the previous commit. This is helpful if you want to just simply restart and return to your previous state if the new code that you inserted produced a bug and you're unable to trace its origin, so you can revert back to the working version. This technique of debugging is also why it's good to insert your code in small increments so you can easily revert back to older versions without losing a lot of code.
- **`git restore .`** - This has a period and space after the restore. This will revert all the files in the repository entirely back to the version from the last commit

Git Branching and Working on a Team:

- **`git branch <branch-name>`** and **`git co <branch-name>`** -These two commands both will create a branch. Get used to using branches. Branches are useful for when you're trying to debug a problem, or implement a new feature. To see how they're useful, it is important to realize that it is not uncommon to get distracted, and have to redirect your attention to another feature or bug. Without

branches (and the ability to roll-back to before your work on the feature), you end up with a commit history of half-finished features intertwined with bug fixes and other features. This makes your PRs schizophrenic and difficult to review. So by default, you should *always* be developing on a non-mainline branch. When you go fix a bug, or work on a new feature, ask yourself if it deserves its own branch.

- `git push -u origin <branch name> -` This command is essential to pushing the branch that you just created on your local machine, to now be tracked by your remote repository. You will need to run this command each time you create a new branch in your local machine/repository.
- `git branch` -This command will list out all the branches that exist in your current github repository. If you don't see the branch that your team member just made, that means you need to git fetch, because your machine doesn't have the latest version of what branches exist in the github repository
- `git stash` -This command is an acknowledgment that we mess up with our branches. If you find that you're somewhat accidentally developing a new feature, or fixing a new bug, but have a bunch of unstaged changes on the current branch, you can `stash` them, create a new branch, `git stash pop` the changes into the new branch. I'm sure there are other ways to do this, but this example at the very least demonstrates how `stash` can be used to delay a current set of changes.
- `git checkout <branch-name>` -This will switch your local machine to now view the code in the branch name that you're checking out. If you don't see any of the code changes that your teammate made yet after checking out in the branch,
- `git fetch -` This will download all the commits and files from the local repository to your local repository but will not merge them into your current working branch. This is a much more cautious approach to retrieving the changes from the remote repository because it will allow you to inspect all the changes from the remote repository before integrating them into your own local repo. Although this is more meticulous, this provides more control over the integration process
- `git pull` -This is similar to git fetch, however this will immediately update your current local working branch with the fetched changes, potentially causing merge conflicts in your working files and is riskier if done without caution, as automatic merging can often lead to unexpected changes and issues. This should only really be used when you're certain that the remote changes won't overlap with your local uncommitted modifications.
- `git rebase` - enables you to update a branch to an updated master, and to "squash" multiple commits, and clean up your commit history. When the master progresses beyond the point where your branch split from it, you generally want

to fast-forward merge your changes onto the new master. `git rebase master` is your friend here. It is generally superior to `git merge` as it maintains a cleaner (linear) history. `git rebase -i` allows you to rewrite history by combining different commits. This is very useful as it enables you to have a set of commits that tell a story, and don't have a number of "in progress" commits. My suggestion is that you label your commits that you'll likely want to get rid of in the future with `TODO` as a header on the title line. More information about this can be found here and here.

When do merge conflicts happen?

- When two developers work on the same functions in the same file and one person pushed their version to the main repository, and then the second person tries pushing to main repository too without first integrating their changes to that that have been made to that file in main.
    - Think of the following sequence:
        - A repository has `x = 0` as one of its statements.
        - Two developers clone that repo.
        - One developer changes it to `x = 1`, makes a commit, and pushes to the initial repo.
        - The second developer (who still sees `x = 0` since that is what they cloned) sets `x = 2`, and commits.
        - What do you think the repository should contain? Two developers have made **conflicting changes** to the code!
        - When the second developer tries to push, they will get a merge conflict and see that `main` has `x = 1`, while there version has `x = 2`, and it is their job to update the code to choose which variant is correct.
        - Once they do, they commit the update that merges the update, and push the resolution.
    - Merge conflicts are weird and complex, but one safe cautious way of avoiding them for now as you're unfamiliar with git is to ensure you and your partners are working on two separate parts of files if you're inserting code at the same time and you don't want to worry about unexpected changes. This means that you and your partner wouldn't have overlapping changes when pushing to main.
- Good tutorial video about managing merge conflicts posted by VS Code:
    - https://www.youtube.com/watch?v=HosPml1qkrg&t=180s

Github Concept Explanations Sources:

- **Git Intro:**
  - **https://github.com/gwu-cs-os/resources/blob/master/gitprimer.md**
    - This link explains git terminology (what a branch is, what a repository is, what it means to clone a repository, and essentially the meaning behind commonly used git commands)
- **Github best practices / rough guidelines on how to use git**
  - **https://github.com/gwu-cs-os/resources/blob/master/github.md**
    - Advice more specific to github (which uses git)