# Git for 61C



Writing a Git guide is a bit like reinventing the wheel; there are dozens of great Git resources available to you on the internet. Here are some I endorse:

- 61B's Git tutorial, by Sarah Kim and Josh Hug; I did this as a student, and it has been incredibly useful; it starts out from the very beginning and takes you through everything you need to know, and I couldn't recommend it highly enough.
- A git cheat sheet from Github with all the basic commands in one page.
- A git cheat sheet that has a great diagram that I highly recommend you look at.
- The Atlassian git tutorials are very complete, written to be fairly accessible to beginners, and are full of great diagrams. If your goal is to understand what a command is doing, this should be your first stop.

Please don't hesitate to ask conceptual questions about Git. When I was a student in 61C it seemed like everyone was a Git/command line wizard but me, but as a TA, I've discovered this is absolutely not the case.

And if you have any suggestions for this guide, definitely let us know!

## Part 1: Getting Started with detailed explanations

### Step 1: Download Git

- Before working with git, you'll need to download it! Git is available to download for Mac, Windows, and Linux.
    - Git is a program that tracks changes in your files and allows you to coordinate multiple people changing the same files

### Step 2: Start tracking with Git

- Git only tracks changes in a directory when you "initialize" that directory as a git repository
    - Option 1: you've already created a directory on your local machine for the files you want to track with git
        - Command: `git init`
        - What does it do? In whatever directory you're in, creates a directory called ".git" that stores all the information Git needs to track changes
            - Inside .git directory, a file called .gitignore, specifying files and file types that git will ignore (if you do "git add -A" these files will *not* be added)
            - Often this includes executable files, which are specific to the architecture of the computer it's compiled on
    - Option 2: don't have a directory on your local machine yet
        - Command: `git init [directory name]`
        - What does it do? Creates an empty directory with the name you specify, and a .git directory within it

### Step 3: Add the repository with the files you want to start out with as a remote

- A remote may exist on a website like Github or Bitbucket, or it may be a repository on a machine that you have SSH access to
    - Command: `git remote add [name of remote] [URL]`
        - Usually: `git remote add origin [URL]`, or `git remote add starter [URL]`

### Step 4: Get files from the remote

- If you want to work on your local machine, this is the step where you actually transfer the files
    - Command: `git pull [remote name] [branch]`
        - Usually: `git pull origin master`
    - What does it do? Pulling is basically equivalent to two steps, which you can do separately:
        - `git fetch --all` to pull all changes on all branches from the remote

        - `git merge [remote name]/[current_branch]`, usually `git merge origin/master` which attempts to combine the changes from the remote repository with the current state of the files in your repository

### Step 2B: git clone

- git clone does many of the previous steps all-in-one
    - Creates a copy on your local machine of a repository that exists elsewhere
    - `git clone [URL] [directory name]`
    - What does git clone do?
        - `git init [directory name]`
        - `cd [repo name]`
        - `git remote add origin [URL]`
        - `git pull origin master`

<u>The holy trinity:</u>

### git add

- Adds specified files to "staging area" so git tracks changes in them. Some options:
    - `git add -A`
        - Add all files
    - `git add *.c`
        - Add all .c files
    - `git add foo.py`
        - Add a particular file called foo.py

### git commit

- Packages changes for tracked files into a "commit", so it is officially in the history of your local repository
    - `git commit -m "Message"`
        - Create a commit with the specified message
    - `git commit -a -m "Message"`
        - Does `git add` for all files and creates a commit with the specified message

### git push

- Sends all commits to a remote repository
    - `git push`
        - Pushes to the default upstream and branch. Usually, default upstream is origin, but you can change this using the "--set-upstream" flag
    - `git push origin master`
    - `git push [remote] [branch]`

## Part 2: Getting Started, with short descriptions

**I have the files I want on Github, and I want them on a machine I'm working on**

```
git init [project directory]
cd [project directory]
git remote add origin [remote URL].git
git pull origin master
```

--OR--

```
git clone [URL]
cd [remote repository name] // You *don't* chose name of directory
```

--OR--

```
git clone [URL] [directory name]
cd [directory name] // You *do* choose name of directory
```

**I have Git set up, I want to make some changes and push them so that they're reflected on Github**

```
git add [files changed]
git commit -m "Commit message"
git push origin master
```

**I want to get the starter files for class, make some changes, and push them to my personal Github**

```
git init [project directory]
cd [project directory]
git remote add starter [starter URL].git
git pull starter master
git add -A
git commit -m "Starter files"
git remote add origin [origin URL].git
git push origin master
```

## Part 3: Slightly more advanced Git

### Going back in time: revert and reset

- git revert
  - Syntax: `git revert [commit hash for BAD commit]`
    - How do you get the commit hash? Use `git log`
    - If you want to revert to HEAD, you can use `git revert HEAD`
  - What does it do? Takes all the changes you've made since a specified commit, inverts them, and creates a new commit, and moves the HEAD pointer to that commit
  - This is a somewhat safer option than `git reset`, and maintains your "bad changes" in case you find you'd like to go back to them
- git reset
  - Syntax: `git reset [commit to reset repository to]`
  - What does it do? Moves the HEAD pointer to the specified commit
    - If you use the `--hard` flag:
      - Removes changes from working directory & staging area
      - This is dangerous! You will "orphan" commits that are chronologically after the one you reset to. You can still get to them using `git reflog` in the short term, but in the long term, they will be removed by the Git garbage collector
    - If you use the `--soft` flag:
      - Does *not* remove changes from working directory & staging area

### Branching and git checkout

- You may want to use a branch if...
  - ...you want to make experimental changes
  - ...you and a partner are working on separate tasks and you don't want to interfere with each other for testing purposes
  - ...you want to pull changes from Github that you don't want to mix up with local changes
- The default branch is master
- Figure out what branch you're currently on using `git status`
- git branch
  - `git branch`
    - List all branches
  - `git branch [branch name]`
    - Make a new branch with specified name
    - Unlike checkout, does not move you onto this branch
  - `git branch -d [branch name]`
    - Delete the specified branch

        - Use -D (capital “D”) if you want to delete unmerged changes; I recommend -d for safety
- git checkout
    - Make a new branch and check it out, from HEAD:
        - `git checkout -b [new branch name]`
    - Make a new branch and check it out, from existing branch:
        - `git checkout -b [new branch name] [existing branch name]`
    - Switch onto a different branch:
        - `git checkout [branch name]`
    - Switch onto a branch from a remote repository
        - `git fetch --all`
        - `git checkout [remote branch name] origin/[remote branch name]`

## Part 4: The 8 most common errors I've seen as a 61C TA

1. `'git' is not recognized as an internal or external command, operable program or batch file.`
    - First: did you download git?
    - Second: did you close and re-open your command prompt, if you downloaded git recently?
    - Third: is git on your path? Two fixes:
        - Are you on Windows, and trying to use the command prompt?
            - Use Git bash instead
            - Re-download Git and select the option to use git from command prompt
        - Add git to your path
2. `fatal: not a git repository (or any of the parent directories): .git`
    - First: are you in the right directory? Do a double check
    - Second: you're in the right repository. You may need to initialize this repository as a git repository. Do this with `git init`
3. `fatal: unable to access '[URL for git repo]': The requested URL returned error: 403` when trying to push (or any sort of "`fatal: unable to access`" error)
    - If you're using a URL: check the spelling of the URL for the repository you're trying to push to
    - If you're using a named remote:
        - Use `git remote -v` to list all of your remote repositories
        - Check the URL for the one you're trying to push to
        - 99% of the time, this is what happened: you have origin set to our starter code, which students have read access to but not write access.
            - The best solution is for you to reset the URL for origin to your personal repository, with these steps:
                - `git remote set-url origin [URL]`
            - If for some reason you want to remove a remote repository, you can use this command:
                - `git remote rm origin`
4. I want to transfer something from my local machine to the hive machine; I'm thinking about using scp, but I don't want to mess up my Git history
    - If you have the same git repository on both machines, you can use this all-in-one command:
        - `git push cs61c-XXX@hive30.cs.berkeley.edu:~/projectY.git master`
    - Or, if you want to add the hive machine as a remote repository you can push to:

        - `git remote add hive30 cs61c-XXX@hive30.cs.berkeley.edu:~/projectY.git`
        - `git push hive30 master`
5. Help, I tagged the wrong commit for submission of my project!
    - Step 1: Remove the incorrect tag locally
        - `git tag -d [tag]`
        - e.g. `git tag -d proj1-2-sub`
    - Step 2: Push to remove the incorrect tag from your Github
        - `git push origin :refs/tags/[tag]`
        - e.g. `git push origin :refs/tags/proj1-2-sub`
    - Step 3: Re-tag the correct commit locally
        - `git tag -a [tag] [first 7 digits of commit hash]`
        - e.g. `git tag -a proj1-2-sub 9ebd89c`
    - Step 4: Push this new, correct tag to Github
        - `git push origin :refs/tags/[tag]`
        - e.g. `git push origin :refs/tags/proj1-2-sub`
6. The horror! A merge conflict
    - Resolving a merge conflict is a lot of work, what are some tools to make it easier?
        - If you're working locally and can use an IDE:
            - There's a great interface for resolving merge conflicts in Visual Studio Code--it really minimizes the work you have to do. VSCode is available for Linux, Windows, and Mac
        - If you're working over SSH:
            - You can use git rebase interactively to solve merge conflicts from the command line.
7. I want my local repository to match my remote exactly
    - Be careful using `git reset --hard`!! Something to be safe--save your local changes on a branch *before* you do this, so that you still have access to them if something unexpected happens
        - `git add -A` Add all your files
        - `git commit -m "Saving local changes before hard resetting"` Commit your changes to those files
        - `git branch saving_local` Create a new branch called "saving_local" for your local changes
    - Get everything from your remote
        - `git fetch [remote name]`, probably `git fetch origin`
    - Reset to what you got from your remote
        - `git reset --hard [remote name]/[branch]`, probably `git reset --hard origin/master`
8. I want my remote repository to match my local exactly
    - Be careful using `git push -f`! It overwrites your remote irrevocably.

- The command: `git push -f [remote] [branch]`, so probably `git push -f origin master`