UC Berkeley
CYPLAN255
Spring 2024

Getting Started with GitHub

1. Configuring Git

    1.1. Tell git who you are (replace “Mona Lisa” with your own name):

    ```
    git config --global user.name "Mona Lisa"
    ```

    1.2. Tell git what your e-mail address is (replace e-mail address with your own):

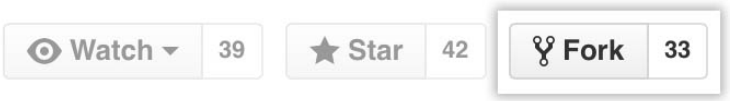
    ```
    git config --global user.email "mlisa@berkeley.edu"
    ```

    1.3. Double check that git got all that right

    ```
    git config --list
    ```

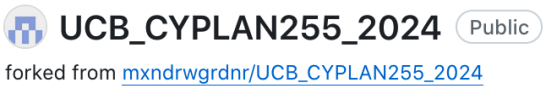
2. Forking the class repo

    2.1. Go to https://github.com/mxndrwgrdnr/UCB_CYPLAN255_2024
    2.2. On the top right of the page, click the button labeled “Fork”



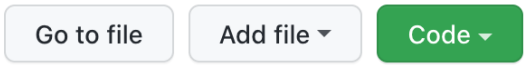
    2.3. Now, instead of seeing this at the top left of your screen

    you should see something like this:

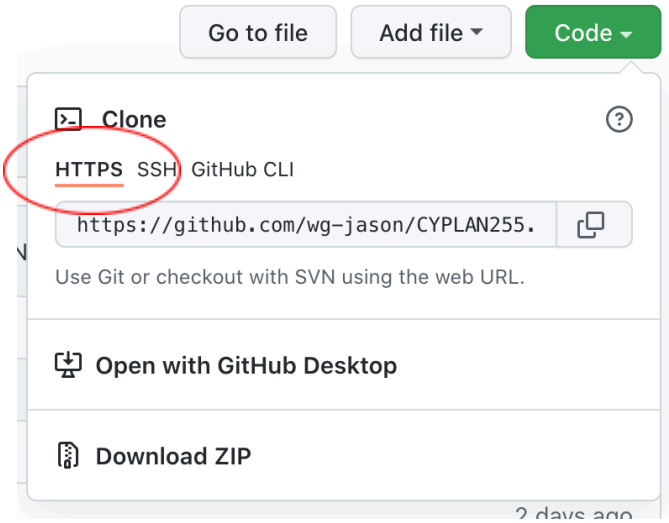


3. Cloning your fork

    3.1. From your fork, click the green button labeled “Code”



3.2. In the drop-down menu, make sure you have HTTPS selected



3.3. Click the copy button to copy the URL to your clipboard:

3.4. Open up a terminal/bash shell/command prompt and type:

```
pwd
```

This is the location on your local computer to which you are about to copy the repo. Take note. If this is not where you want the repo to be stored on your computer, either `cd` to the right location before continuing, or you can simply move the project (repo) directory afterwards.

3.5. Now in the same terminal type

```
git clone <paste the URL you copied here>
```

and hit `<Enter>`.

3.6. Navigate into the root directory of the repo

```
cd UCB_CYPLAN255_2024
```

3.6. To check that you did everything correctly, do

```
git remote -v
```

and you should see something that looks like this:

```
origin    <URL of your fork.git> (fetch)
origin    <URL of your fork.git> (push)
```

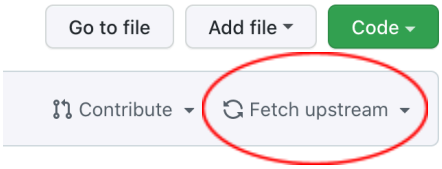
If instead of the URL of *your* fork, you see the URL of the main repo ("https://github.com/mxndrwgrdnr/UCB_CYPLAN255_2024.git"), then you have cloned the main repo rather than your fork. Delete the folder you're in, return to step 3.1, and try again with the right URL.

4. Syncing your fork

4.1. *FROM THE UI:*

The GitHub UI provides us with a convenient method for syncing one branch of your fork with the corresponding branch on the original "upstream" repo.

4.1.1. Navigate to your GitHub fork in a browser and click the button labeled "Fetch upstream" under the green "Code" button



and then click "fetch and merge" when prompted. Do this as often as you like.

4.1.2. Now that your *remote* is synced, you need to copy those changes to your *local* copy. To do that, all you need to do is

```
git pull
```

*4.1.3. NOTE: fetching and merging from the UI is meant to sync two specific branches. This means **it will not get any new branches which have been created upstream**. If you need to get new branches (you will) you'll have to use the command-line instructions.*

4.2. *FROM THE COMMAND-LINE:*

To sync your fork from the command-line, you'll first have to add the upstream repo as a new remote.

```
git remote add upstream
https://github.com/mxndrwgrdnr/UCB_CYPLAN255_2024.git
```

Now if you do `git remote -v`, you'll see you are now tracking two different remotes. This is good. Now you'll want to make sure you're on the default branch:

```
git checkout main
```

And then fetch all the changes from the upstream repo:

```
git fetch upstream
```

At this point, you haven't merged any changes, you've just told your local Git to download the changes and hold them in a kind of staging area as unmerged commits. Once you're ready to merge them, go ahead:

```
git merge main
```

The command above tells Git to merge all *unmerged* commits in the main branch to the branch you currently have checked out, which, if you've been following these instructions, should be "main". Now your *local* main is up-to-date (synced) with the upstream *remote* main. If any new branches have been created in the upstream remote, you'll now have access to them from your own local repo. The fetch and merge effectively created those new branches for you when you synced with the upstream remote. Give it a try:

```
git checkout hw0_submissions
```

and then confirm you're on the right branch with

```
git branch
```

5. Submitting assignments

5.1. As mentioned in the assignment README, assignments should be submitted on the branch which corresponds to the assignment. For example, Assignment 1 will be submitted via the "hw1_submissions" branch, and so on. To change branches do

```
git checkout hw0_submissions
```

5.1.1. If you get an error telling you that no such file or branch exists, then either I haven't created the "hw0_submissions" branch yet on the upstream repo, or you have not properly synced your fork. It's most likely the latter. Revisit Step 4.2 above to try syncing again.

5.2. Remember, do not do your work in a file with the same name as a file that you find in the repo. You may use existing files as a template by creating a copy,

renaming that copy with a unique identifier (first and last name, or last 4 digits of your student ID). For example:

```
cp assignment_0.txt assignment_0_max_gardner.txt
```

Or just create a brand new file and do your work there.

5.3. Before committing your changes, make sure you're on the right branch. For homework-related commits, there will be a branch which corresponds to each assignment. For example, to switch the the branch for assignment 1, do

```
git checkout hw1_submissions
```

and then confirm you're on the right branch with

```
git branch
```

If Git can't find that branch, repeat Step 4 for syncing your fork.

5.4. When you are ready to push your changes from local to remote, you must first tell Git which files contain the changes you want to push:

```
git add assignment_0_max_gardner.txt
```

5.5. Then you are ready to *commit* those changes. No going back now!

```
git commit -m "<mandatory descriptive message goes here>"
```

5.6. OK so we're about to push the changes, but we have a few options for how to do that. The easiest, foolproof way to do it is to be as explicit as possible by telling Git exactly where you want to push to. by specifying the name of the remote and the branch on the remote where you want your commits to go:

```
git push origin hw0_submissions
```

This is useful because sometimes your local copy might be tracking multiple remotes. In fact, if you followed step 5.1.1.1 that's exactly what you did (try `git remote -v` and you'll see what I mean). But if your Git project has multiple remotes to choose from, it might not know which one you want to push to, and it might actually push to the wrong one by mistake if you're not careful. The other way to avoid pushing to the wrong remote is to *set the default*:

```
git branch --set-upstream-to origin/hw0_submissions
```

and now you can simply do

```
git push
```

and Git will know exactly where to push to.

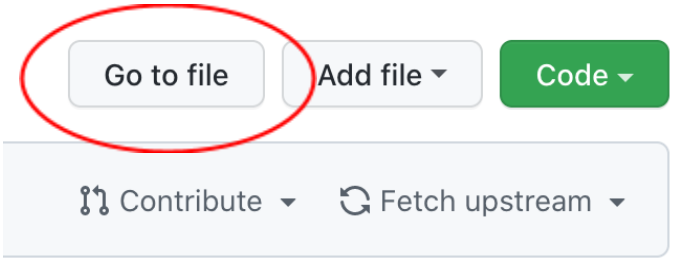
5.7. At this point, if you cloned your repo with HTTPS, Git should prompt you to enter your github username and authentication token in the terminal. Go ahead and do that.

5.7.1. If you find it cumbersome to have to enter your username and access token every time you want to make a commit, there are other authentication options that you can try, although they can be a bit trickier to set up. See the official GitHub docs here for more details.

5.7.1.1. GitHub no longer allows password authentication. Instead, you must use a "personal access token". See the official GitHub docs here for instructions on how to generate this token. NOTE: Make sure you follow the instructions for CLASSIC token, not fine-grained.

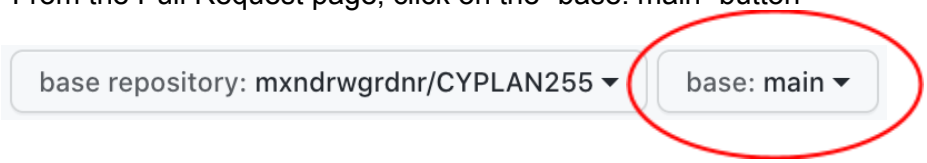
5.8. Once you've pushed, go back to the browser and look at your fork. You should see your commit message at the top of the page along with a note that says changes were pushed "a few seconds ago".

5.9. When you're ready, click the button labeled "Contribute", just to the left of the "Fetch upstream" button you used to sync your fork:



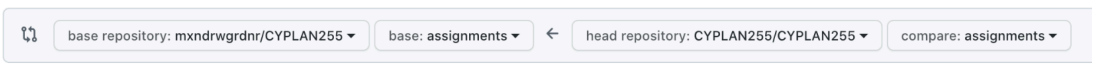
and click "Open Pull Request" when prompted.

5.10. From the Pull Request page, click on the "base: main" button

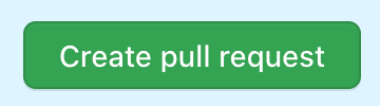


and change it to "base: hw0_submissions".

5.11. Then do the same thing for the "head" repository (your fork), by clicking on the "compare: main" button, and selecting the "hw0_submissions" branch. You should now see something like this:



5.12. When you're ready, click the green "Create pull request" button:



5.13. Lastly, give your pull request a descriptive title, and a helpful comment or two. The title does NOT need to be unique. Then click the green button once more and give yourself a pat on the back :)

APPENDIX

A. Handling Merge Conflicts

Before reading on, please read about merge conflicts on the official GitHub docs here.

Git is usually pretty good about automatically merging different versions of files together, but sometimes it needs help. These are called merge "conflicts", and sometimes they are minor enough that you can fix them directly in the GitHub UI, but other times you'll need to use the command-line. Which is the best approach will depend on the type of conflict that has occured. Below I describe just *some* of the ways to resolve the kinds of conflicts that you are most likely to encounter throughout the course of the semester

I. *PULLING FROM REMOTE*

Whenever you use the `git pull` command to copy changes from a remote repo to your local version, Git is actually doing a **fetch** and a **merge** behind the scenes. Here are two reasons you might encounter merge conflicts during this process:

a. As mentioned above, Git is usually pretty good about merging different versions of files on its own. But remember, Git doesn't really know about anything you've done until you **commit** your work. This means that if you have made changes to a local copy of a file, but have not yet *committed* them, then Git can't incorporate them into the merge step of the pull. As a result, even though Git technically *can* do this merge, it will instead throw an error to let you know that the merge would have overwritten your local changes:

```
error: Your local changes to the following files
would be overwritten by merge:
my_directory/my_file.foo
Please, commit your changes or stash them before you
can merge.
```

Git has also provided us with a couple suggestions for how to proceed. Which we choose depends on whether or not we want to preserve the local changes. Let's go through each scenario:

i. If you don't care about overwriting your local changes, the simplest thing to do is `git stash`. This will literally stash your changes away in a temporary location to allow you to avoid triggering the previous error message when you try the git pull again. NOTE: Git normally keeps your stashed changes around for ~90 days, so it is possible to recover them if later on you realize that you actually needed them. Otherwise, just `git pull` and forget about it.

ii. If you do care about saving your changes, now is a good time to say "Thank you Git! You have spared me from the unthinkable horror of deleting important work that I spent hours on!". Then, simply add and commit the changes and re-try the pull. You might not be out of the woods yet, as the changes from remote could still conflict with the changes you just committed, but at least you won't have to worry about losing your work!

iii. What if you want to preserve your changes, but you also don't want to commit them? Maybe you have code in which you've temporarily hardcoded a password, and you don't want that password committed to the Git history for all eternity. Here you can do (in order):

```
git stash;
git pull;
git stash pop;
```

The first two lines are exactly the steps you would take if you didn't care about preserving your changes. But here we are just using them to avoid the error and successfully complete the pull. Once the pull is complete, the third command will take the changes you stashed and slap them on top of the changes you just pulled from the remote. This might still raise merge conflicts which will need to be resolved, but at least you don't have to worry about losing your work! And you haven't committed anything!

iv. What if you're not sure about whether your local, uncommitted changes are important or not? You can use

```
git diff <filename>
```

in your terminal, and Git will show you exactly what changes you have made to the file in question. Then you can decide for yourself whether to stash or commit.

b. Sometimes there are merges which are simply too complicated for Git to resolve on its own. When this happens, Git will temporarily create a version of the file in question containing *both sets of changes* (remote and local). Obviously you will encounter issues if you try to run this mutant file. From here you typically have two options:

   i. What Git expects you to do is to open the mutant file in a text editor and manually resolve the conflicts, which it has highlighted for you, line by line. If the file in question is a normal, human-readable file that can be opened in a text editor and made sense of (e.g. .txt or .py) then this is the way to go. It's actually quite painless, and the official GitHub documentation has an excellent guide to walk you through this process here.

   ii. If the file in question is something messier, like (you guessed it) a Jupyter Notebook, then opening the mutant file in a text editor isn't going to be very helpful – if you don't believe me give it a try and you'll see what I mean. Typically, the easiest thing to do here is to pick one version of the file and tell Git to keep all of the changes from that version:
   - To keep all of the changes from remote do

     ```
     git checkout --theirs <filename>
     ```

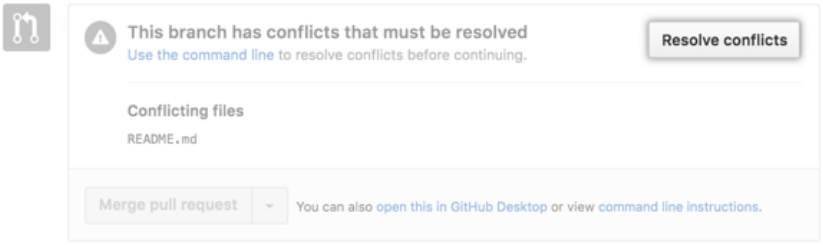
   - To keep all of your local changes do

     ```
     git checkout --ours <filename>
     ```

   NOTE: Third party tools do exist that allow you to merge Notebook conflicts just like you would any other file, but I've never used them and so I can't offer an informed opinion. For what it's worth nbdime looks pretty cool.

II. *SYNCING A FORK FROM THE GITHUB UI*

The GitHub UI gives you a nice way to sync your fork with the upstream repo directly from a browser (see Step 4 on Page 3 of this guide). Unfortunately, the convenience of this approach disappears as soon as merge conflicts show up. If merge conflicts are found, you'll see this message:



From here there are two possibilities for how to proceed:

a. If the file(s) in question are human-readable and capable of being opened in a text editor (e.g. .txt, .py, .md), GitHub will allow you to resolve the conflicts directly in the browser. Please see the official GitHub docs here for a step-by-step guide to completing this process.

b. If the "Resolve conflicts" button is deactivated, then GitHub has determined that your conflicts are too complex to be resolved in the browser. You'll have to use the command line instead. Refer to the official GitHub docs here for a step-by-step guide for syncing your local fork with an upstream repository. When you encounter the merge conflicts (and you will) refer to Section 1 of Appendix A above for how to resolve them.

B. Keeping your forks clean

Some tips we've learned so far this semester from resolving merge conflicts and submitting PRs:

1. Don't commit any changes to "main". This guarantees you'll always be able to fetch the latest changes from upstream into your local copy without any merge conflicts. And merging changes from upstream is the only thing you'll use "main" for. You'll never submit a pull request from origin/main or to upstream/main.

2. When you want to sync your fork (e.g. at the beginning of each class), just do

```
git checkout main
git fetch upstream
git merge main
```

You can make whatever changes you want to the files in main but do not commit these changes to main. See Tip #4 below for ways to avoid this.

3. Always commit your assignments to the assignment-specific branch (e.g. "hw1_subsmissions". But don't create a "hw1_submissions" branch manually. Always grab it from the main class repo by fetching from upstream. See Section 4.2 of this guide for how to do this.

4. Options for saving changes you make to the lecture (demo) notebooks:

1. Use `cp` to create a copy of the notebook and give that copy a unique name (e.g. `cp notebook_01.ipynb notebook_01_notes.ipynb`). This file will be safe from any subsequent merges from upstream because upstream/main will never have a file named like yours, so it will have nothing to overwrite. You can also add, commit, and push these new files from your local copy to the origin in order to sync your remote.
2. Use `mkdir` to create a new directory in your repo and call it anything you want (e.g. `mkdir maxs_notes`). Then use `cp` to copy the notebooks with your changes into your new directory. Files in this directory will be safe from any subsequent merges from upstream because upstream/main will never have this directory, so none of its changes could ever overwrite these files. You can also add, commit, and push these new files from your local copy to the origin in order to sync your remote.
3. Create a separate branch for storing your changes. For example, you could do `git checkout -b my_notes`, and then commit your changes there.

   If you've already made changes to a file on "main", and you want to preserve those changes, you have two options:

   i. If you have not yet committed those changes, all you have to do is checkout the right branch and commit your changes there.
   ii. If you have already committed those changes, you'll have to first checkout the right branch (e.g. `git checkout my_notes)`, and then do

   `git checkout main -- <path to file>`

   This will have grabbed the copy of your file (with the changes) on "main" and pasted into your special branch for note storage. Now next time you go to checkout main and fetch changes from upstream, you'll encounter some merge conflicts either warning you that your local changes will be overwritten or just telling you that there are conflicts to resolve. If the former, just stash your changes (see Appendix A.I.a.i above). If the latter, you can use `git checkout --theirs <filename>` (see Appendix A.I.b.ii above)