Version Control, Git, and GitHub

Version control is a system designed to manage changes to files for a project. The basic functionality of a version control system includes:

- keeping track of changes
- synchronizing code between developers and users
- allowing developers to test changes without losing the original
- reverting back to an old version
- tagging specific versions

Git is one of many version control systems. It can be run on any number of platforms, including your personal computer.

GitHub is a cloud hosting service that makes use of the Git system for version control. Data is stored online through GitHub in repositories (or "repos") which are tracked with Git.

These two are different but complementary tools. You can have version control without Git, you can use Git without GitHub, and you can use GitHub without *really* using Git OR version control.

Install Git and make a GitHub account

Git is already installed on the CSEL coding environment we will be using for the semester. However, if you want to use it locally on your own machine you can install it following these instructions: https://git-scm.com/book/en/v2/Getting-Started-Installing-Git

You will also need to make a GitHub account, which you can do so at: https://github.com/

The Git/GitHub environments

When using Git and GitHub for version control you will be interacting with three different interrelated environments: 1) the remote repo, 2) the local repo, and 3) the local workspace.

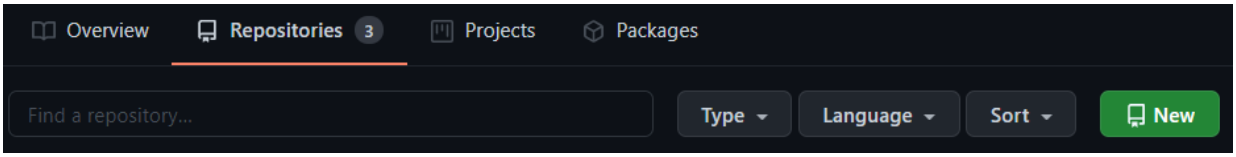
- **Remote repository:** The cloud-hosted database containing your code and its previously archived states. This environment can be updated and accessed by more than one person. It is stored and viewable through the GitHub website.
- **Local repository:** This is much like the remote repo, in that it contains your code and its previously archived states, but it exists only on your local computer and cannot be accessed by anyone but you.
- **Local workspace:** This consists of the actual files on your computer that you're editing, which are being tracked by Git.

Some definitions

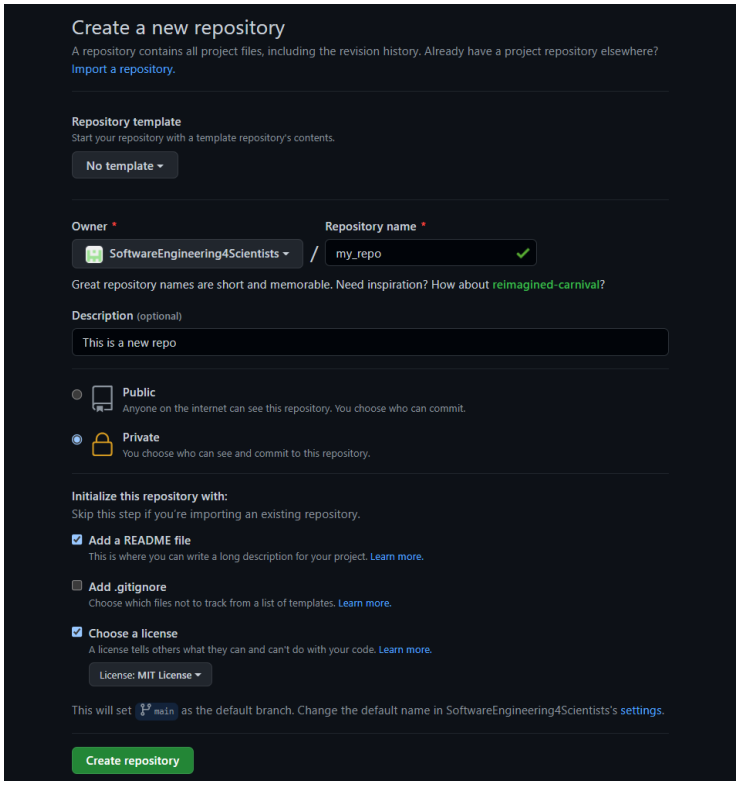
- **HEAD**: Git uses a pointer called HEAD to keep track of what you are doing. HEAD indicates which local branch you are currently on.
- **Downstream**: When you clone a repository, you are now located *downstream* from it
- **Upstream**: This refers to the repository you have cloned/checked-out from. When Github asks you to "set upstream", it is asking you to say where your work (which is downstream, should be pushed up to.
- **Origin:** Origin is used to refer to the repository you have originally cloned from, it is used by GitHub as a stand-in for the original URL you cloned from

Creating a repository

On your GitHub page if you select the "Repositories" tab, it will bring you to the list of your repos. At the top right you'll see a green button "New", which will bring up the "Create a new repository" page.



To create your new repository, specify a name, whether or not it's public or private, and whether or not to include a README and a license.



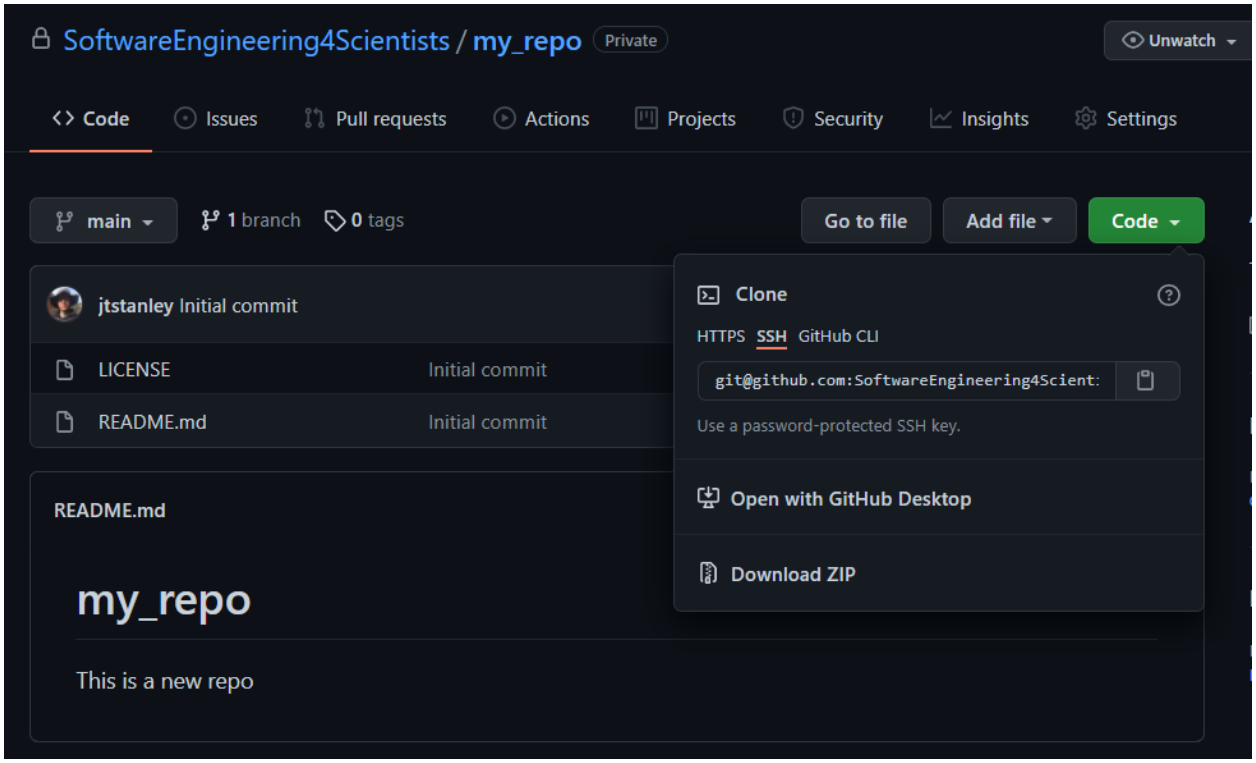
It is recommended that you include a README. This file is a markdown file which will be displayed on your repo's page, below all the files. These README are useful for providing information about the contents of your software and how it can be run.

The license is also important because it is required for a piece of software to be truly "open source." For an explainer on the choice of licence, see here.

Once you have specified all the above information, click the green button "Create repository"

Clone the repository

Now that you've created a *remote* repository on GitHub, you will have to run the "clone" command to create a copy of it *locally* on your computer. On the repo page click the green "Code" button which will bring up the clone menu. Copy the appropriate link.



Run `git clone` with the copied link. This will download the contents of the repo and create the local repo in the current directory in a folder with the name of the repo.

```
jovyan@jupyter-jast1849:~$ git clone
git@github.com:SoftwareEngineering4Scientists/my_repo.git
Cloning into 'my_repo'...
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 0), reused 0 (delta 0), pack-reused 0
Receiving objects: 100% (4/4), done.
jovyan@jupyter-jast1849:~$
jovyan@jupyter-jast1849:~$ ll my_repo/
total 12K
drwxr-xr-x 8 jovyan users 4.0K Aug 26 04:18 .git
-rw-r--r-- 1 jovyan users 1.1K Aug 26 04:18 LICENSE
-rw-r--r-- 1 jovyan users   29 Aug 26 04:18 README.md
jovyan@jupyter-jast1849:~$
```

As you can see the LICENSE and README.md files that were created when the remote repo was initialized are downloaded to this directory. Additionally there is a directory `.git`. It is this directory that makes this a "local repo." It contains all the archived records of the

history of the files contained in this folder. In general, there is no reason why you need to manually interact with the contents of the directory. Just be aware that the files you are tracking must stay in the same directory as `.git`.

To confirm the local repo is sourcing from the correct place you can use the `remote` command as follows:

```
jovyan@jupyter-jast1849:~/my_repo$ git remote -v
origin  git@github.com:SoftwareEngineering4Scientists/my_repo.git (fetch)
origin  git@github.com:SoftwareEngineering4Scientists/my_repo.git (push)
jovyan@jupyter-jast1849:~/my_repo$
```

Here we see the "`push`" location is the same link we copied from the remote repo and it is labeled "origin."---the default name GitHub gives to the original source repo.

Editing files: git add and commit

The files contained in the repo directory exist in your local workspace. In order to edit those files or add new files and have those changes be recorded in the local repository (i.e. as updates to the contents of the `.git` directory) you must use `git add` and `commit`. These are the two most common commands you will run. The three steps are:

1. Edit the file
2. Run: "`git add <filename>`"
3. Run: "`git commit -m "<commit message>`"

For example, say we want to create a new file in the above repo called new_file.txt, include some information in it, and then commit it to the local repo.

```
jovyan@jupyter-jast1849:~/my_repo$ vim new_file.txt
jovyan@jupyter-jast1849:~/my_repo$ cat new_file.txt
The new file for my_repo.
jovyan@jupyter-jast1849:~/my_repo$ git add new_file.txt
jovyan@jupyter-jast1849:~/my_repo$ git commit -m "Adding a new file to the
repo."
[main 8e66e3a] Adding a new file to the repo.
 1 file changed, 1 insertion(+)
 create mode 100644 new_file.txt
jovyan@jupyter-jast1849:~/my_repo$
```

Here we see the new file was committed to the repo on the main branch. In order to see the history of the commits you can use the `log` command.

Commit best practice: Typically, commits should consist of only a small change to the code and not include more than a couple files. The commit message should be informative but does not have to be very long. For example, a commit could consist of a single new function in your library or the addition of some documentation or the change of a parameter name.

Git pull and push

Once your local repo has been sufficiently added to and you'd like to make all your changes available on GitHub you will need to use the `push` command. This command will push all commits that are ahead of those on the remote repository (that have been created after the most recent commits on remote).

```
$ git push origin <branchname>
```

(Note: <branchname> will be addressed in the next section)

However, in the event that there are updates on the remote repository (origin) that are ahead of the current state of your local repository (i.e. there are commits, possibly made by another developer, that are not in your local commit record) then you will need to run the pull command, to incorporate these into your local repo.

```
$ git pull
```

(Note: this is much like `clone` except that `clone` is only meant to be run when creating a new repo)

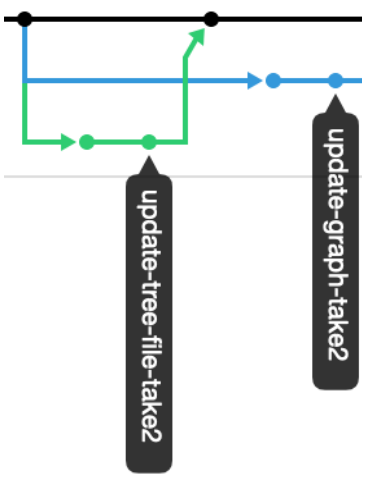
Branches

Next to commits, branches are the single most important feature of Git and GitHub and they become increasingly important the more developers you have working on a project.

```
jovyan@jupyter-jast1849:~/my_repo$ git checkout -b create_new_file
Switched to a new branch 'create_new_file'
jovyan@jupyter-jast1849:~/my_repo$ git branch
* create_new_file
  main
jovyan@jupyter-jast1849:~/my_repo$
```

Here you can see we've created a new branch called `create_new_file` and the branch command tells us the only other branch is the `main`. The "*" indicates which branch we are currently "checking out" and therefore any add/commits made to the repo will be committed to that branch.

If you go to `Insights -> Network` on Github, you can see the graph of commits/branches/merges. **Dots** represent individual commits, **arrows** represent individual branches, arrows that connect back to another line indicate the branch was **merged**. The black line indicates the **`main`** branch. `main` is the definitive branch into which all other development branches (green and blue) should be merged. All development work should be done on separate branches and only merged into `main` once the intended feature is complete. Generally, a branch will have many incremental commits along it.



Standard workflow

1.) Make sure you are on the main branch, when you run `git branch`, this branch will be highlighted with a star.

```
jovyan@jupyter-jast1849:~/my_repo$ git branch
* main
jovyan@jupyter-jast1849:~/my_repo$
```

2.) Create and check out a new branch

```
jovyan@jupyter-jast1849:~/my_repo$ git checkout -b new_branch
Switched to a new branch 'new_branch'
jovyan@jupyter-jast1849:~/my_repo$
```

Note: `checkout -b` command both creates a new branch and checks it out. It is the same as the following:

```
jovyan@jupyter-jast1849:~/my_repo$ git branch new_branch
jovyan@jupyter-jast1849:~/my_repo$ git checkout new_branch
Switched to a new branch 'new_branch'
```

3.) Perform your code edits and commits as usual (example below).

```
jovyan@jupyter-jast1849:~/my_repo$ git add new_file.txt
jovyan@jupyter-jast1849:~/my_repo$ git commit -m "Added a new text
file."
[new_branch 1230ded] Added a new text file.
 1 file changed, 1 insertion(+)
 create mode 100644 new_file.txt
jovyan@jupyter-jast1849:~/my_repo$
```

4.) Push your local changes to the remote branch. If you have not set an upstream remote branch, you will get the following error.

```
jovyan@jupyter-jast1849:~/my_repo$ git push
fatal: The current branch new_branch has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin new_branch

jovyan@jupyter-jast1849:~/my_repo$
```

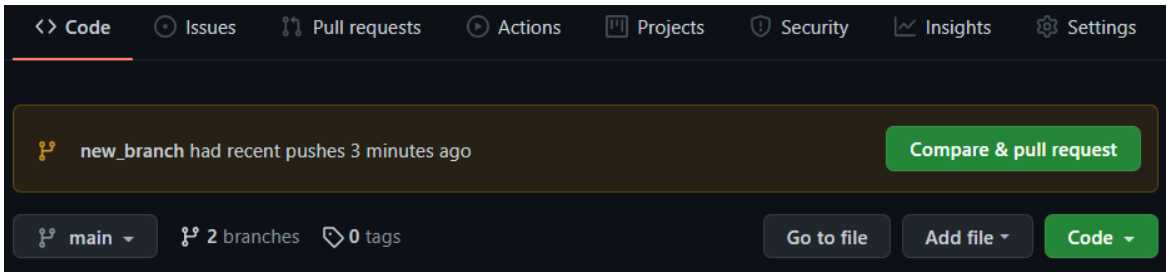
To create a branch in your remote repository with the same name as your local branch, you use `--set-upstream origin <branchname>`

```
jovyan@jupyter-jast1849:~/my_repo$ git push --set-upstream origin
new_branch
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 16 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 325 bytes | 108.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0)
```

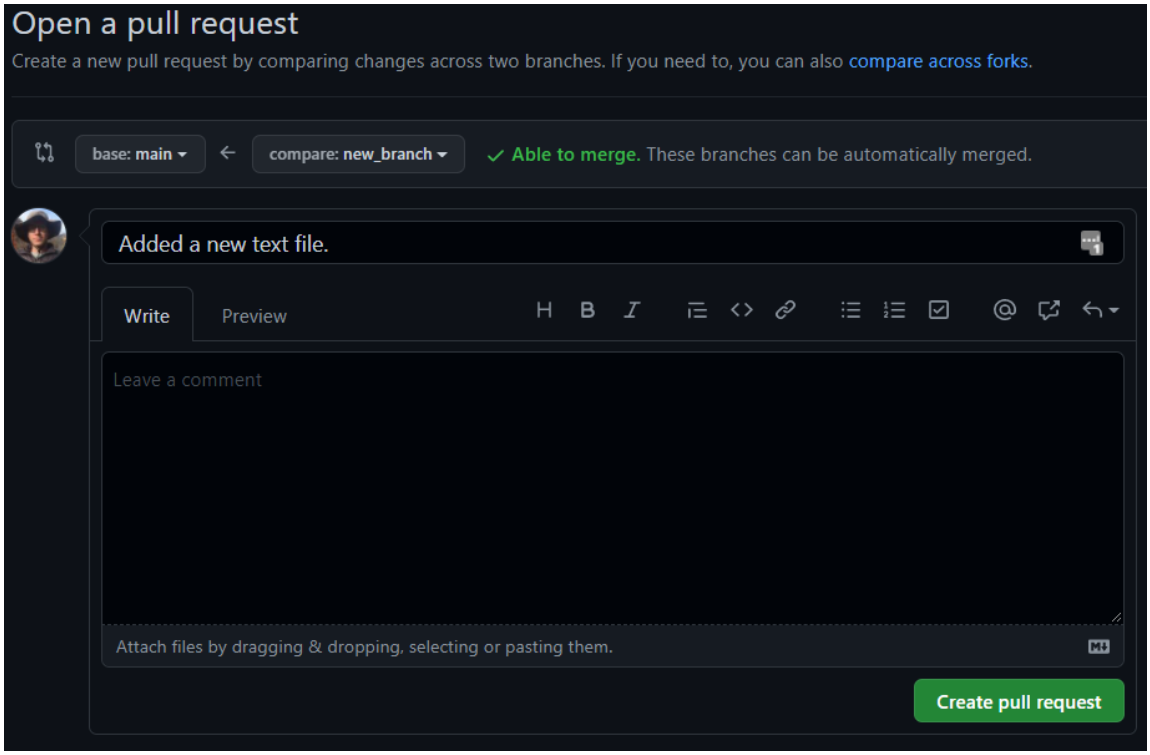
```
remote:
remote: Create a pull request for 'new_branch' on GitHub by visiting:
remote:
https://github.com/SoftwareEngineering4Scientists/my_repo/pull/new/new_b
ranch
remote:
To github.com:SoftwareEngineering4Scientists/my_repo.git
 * [new branch]      new_branch -> new_branch
Branch 'new_branch' set up to track remote branch 'new_branch' from
'origin'.
jovyan@jupyter-jast1849:~/my_repo$
```

5.) You can now create other changes, commits, pushes to the remote, etc. You can now just use `git push`. You don't need to set upstream again.

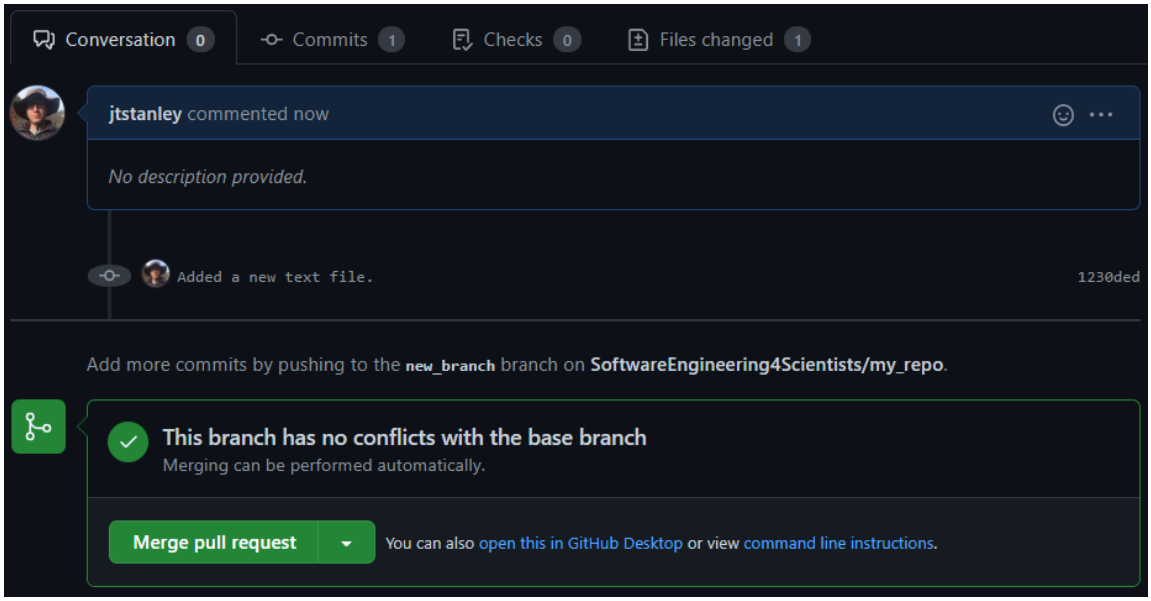
6.) After you are finished working on the branch, run a final push. Switch to the GitHub interface and navigate to the repo to create your pull request and merge it to main. See the following walkthrough:
First, see that you have a new "Compare & pull request" and click the button.



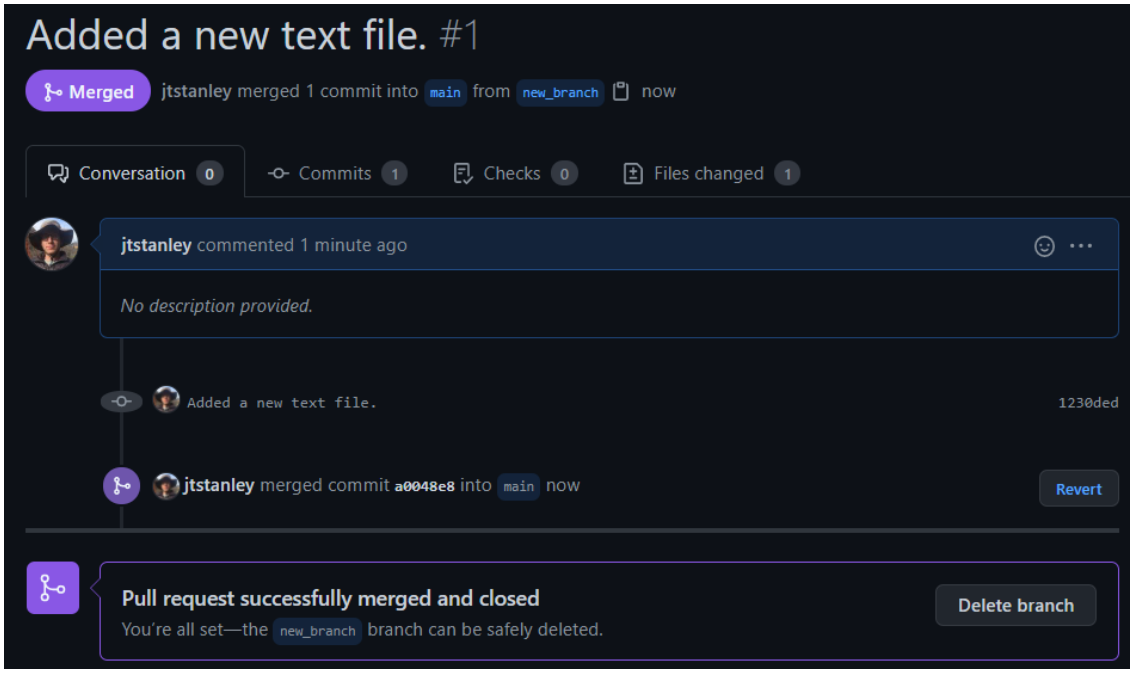
Second, click "Create pull request."



Third, fix your conflicts, if they exist, and then click “Merge pull request.”



Finally, confirm your merge.



7.) Now, return to your terminal, and switch to the `main` branch.

```
jovyan@jupyter-jast1849:~/my_repo$ git checkout main
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
jovyan@jupyter-jast1849:~/my_repo$
```

8.) The local `main` is not up-to-date despite what this message says. You need to create a pull request locally and this will show the most recent updates on the remote repo. The updates from your new branch are then fast-forwarded to your `main` branch.

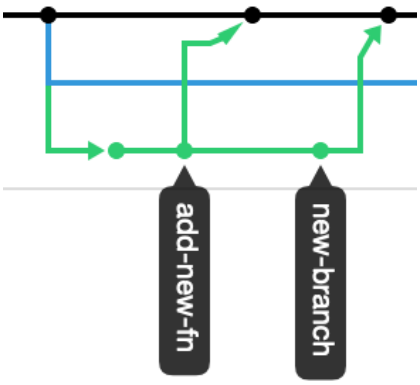
```
jovyan@jupyter-jast1849:~/my_repo$ git pull
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 647 bytes | 161.00 KiB/s, done.
From github.com:SoftwareEngineering4Scientists/my_repo
   2f152fe..a0048e8  main       -> origin/main
Updating 2f152fe..a0048e8
Fast-forward
 new_file.txt | 1 +
 1 file changed, 1 insertion(+)
 create mode 100644 new_file.txt
jovyan@jupyter-jast1849:~/my_repo$
```

9.) Finally, to start new edits you can start again from step 2. At this point you can also delete the branch `new_branch` both locally and remotely if you no longer need it.

Common Problems

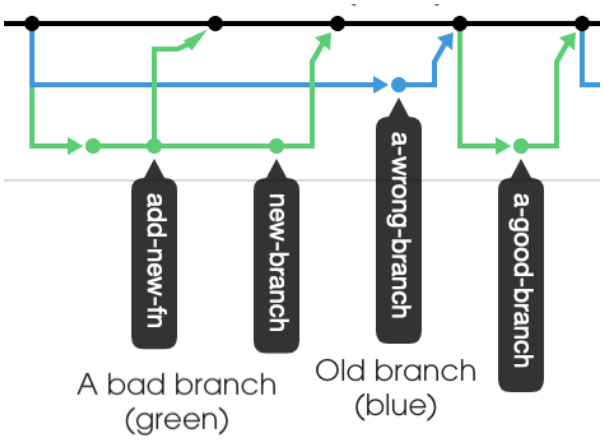
If you go to `Insights -> Network` on GitHub, you can see the graph of commits/pushes/branches. If you see any of the following, you are not following the above protocol and may lose points on your homeworks.

1. **Problem**: You checked out a branch from another branch, not `main` (black)
   What it looks like: An arrow from a dot on a branch, shown here on the green branch(s)



   Why did this happen?: You did not switch to `main` before checking out a branch. (Step 7 & 8 in the Standard Workflow were missed)

2. **Problem**: Say the blue branch was created first and merged with `main` (black) before green was created. However, your new branch (green) started from the very beginning of your history, not where you left off with the old branch (blue).
   What it looks like: Your green & blue branches both start from the beginning of `main`, instead of green branch starting from where blue branch merged



Why did this happen? You did not do the pull request from `main` after pushing the blue branch. Step 8 in the Standard Workflow above.