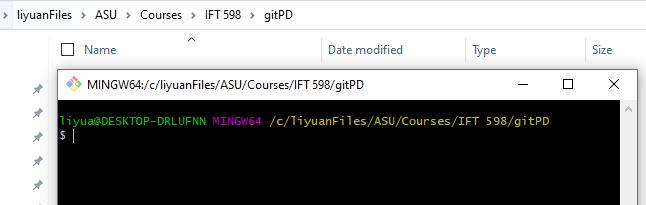
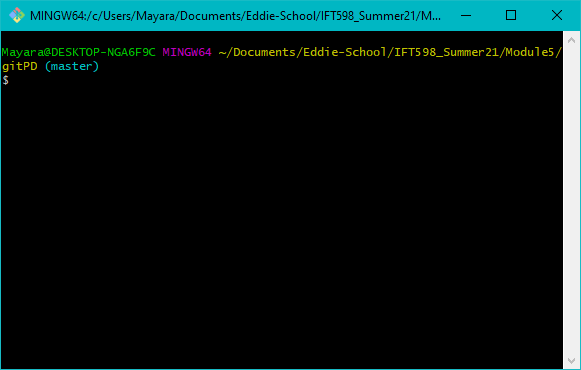
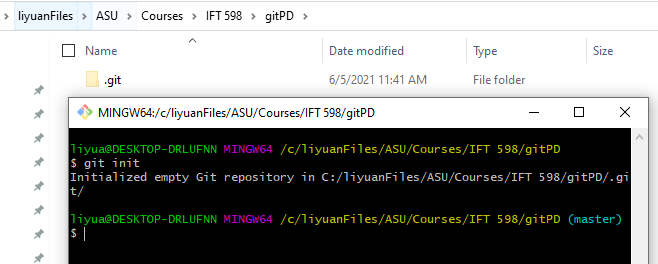
* 1. On your local disk, create a new directory called **gitPD** and change (cd) into it. (This will be the directory you work in unless otherwise specified.)

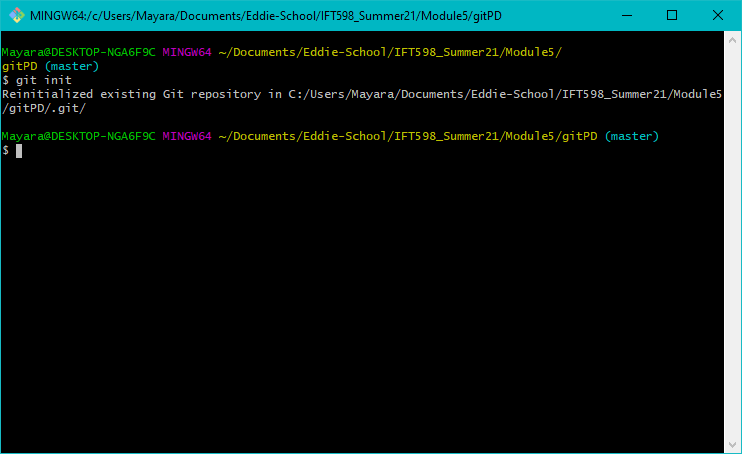




* 1. Initialize a new repository by running the following command: $ git init

This command creates a new git repository skeleton in a subdirectory named .git under the current directory—as indicated by the output message from the command. This means that you’re now able to start using other Git commands in the current directory.

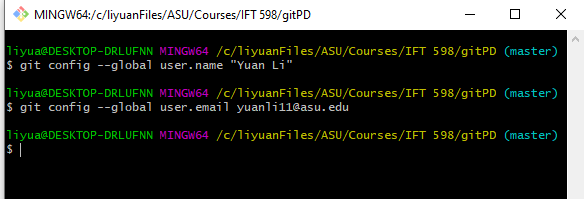


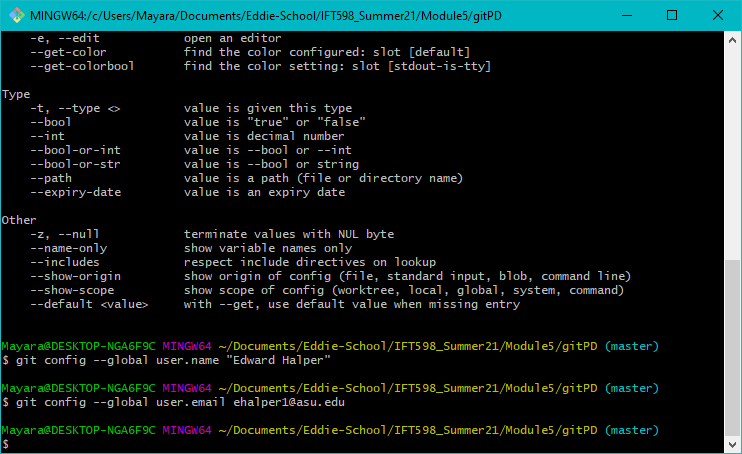


* 1. Tell Git who you are by setting your basic identification configuration settings with the following commands, substituting in your name and email address as the values for the configuration. (Note the double dashes preceding global as you are spelling out the option. Also, values only require quotes if they contain a space.)

$ git config --global user.name "First-name Last-name"

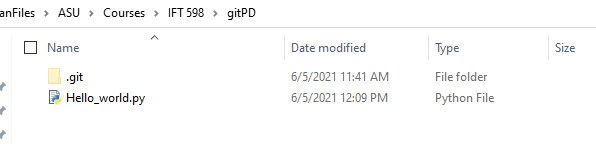
$ git config --global user.email [emailAddress@asu.edu](mailto:emailAddress@asu.edu)





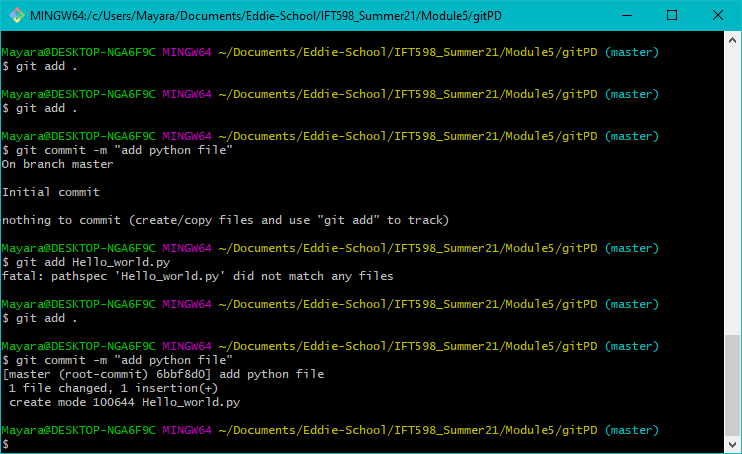
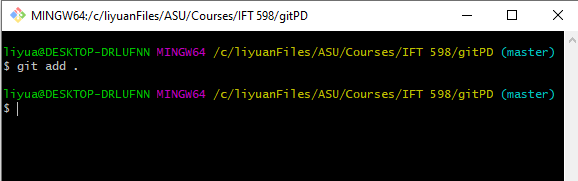
* 1. Now let’s get some content to put through the Git workflow. Create an example file

***Hello\_world.py*** (just a simple source code file to learn git or you can put your project source code here)



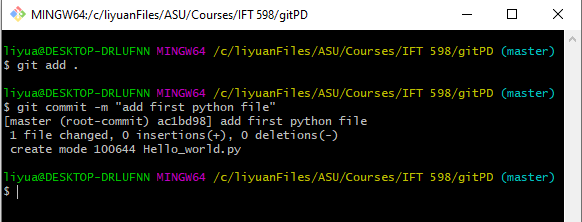
* 1. Stage the files with the add command. (If you prefer, you can add each separate file explicitly rather than using the “.” )

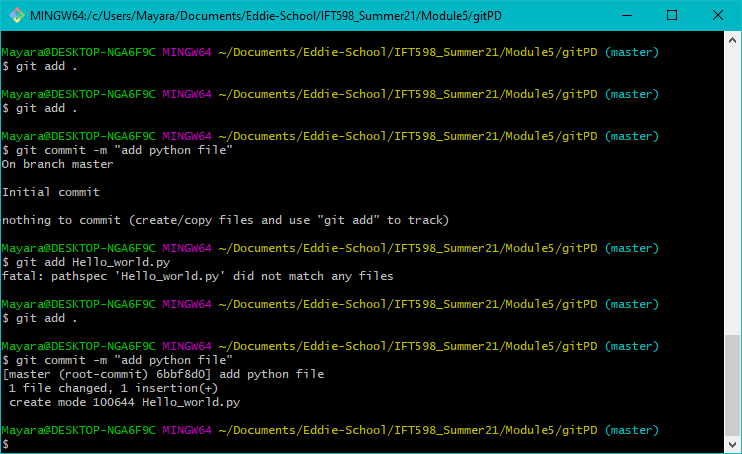
$ git add .



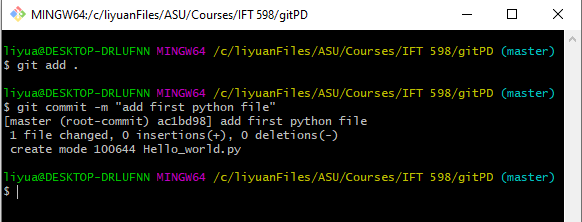
* 1. Commit the files using whatever comment you want.

$ git commit -m "comment string"

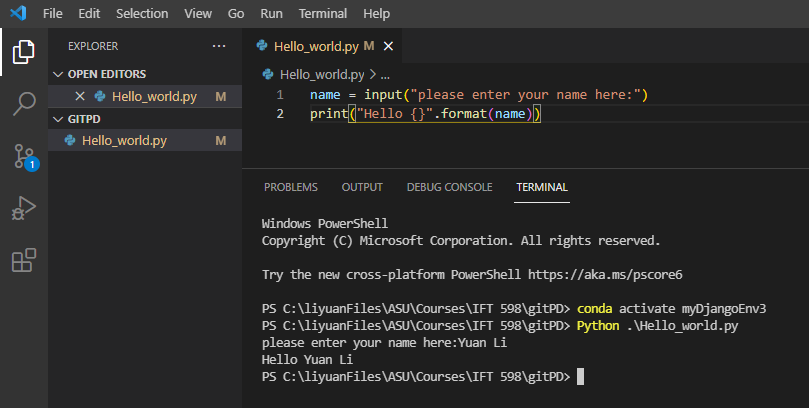


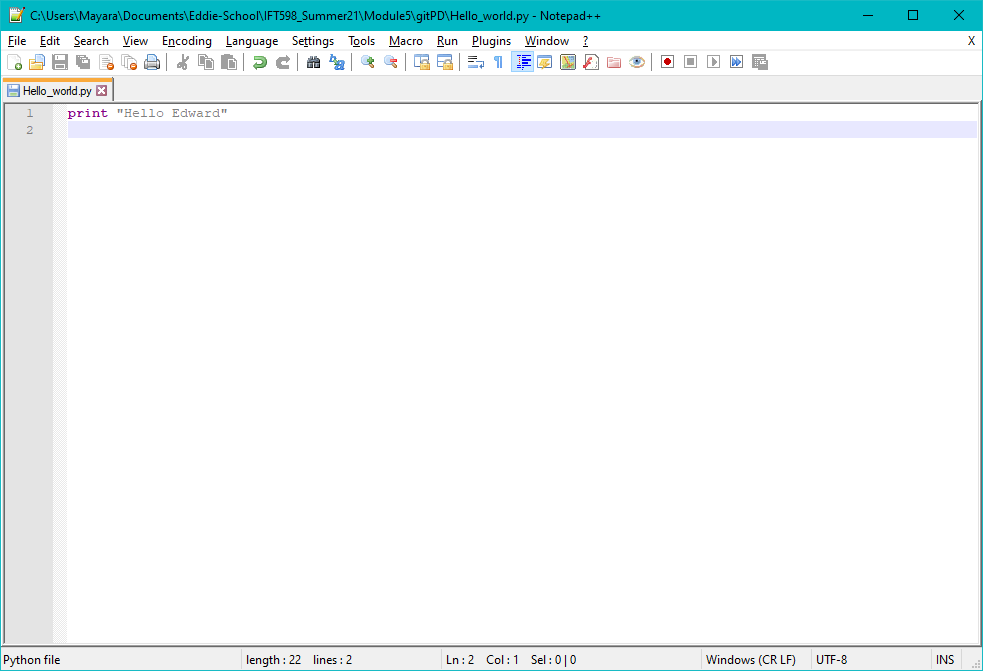


* 1. Notice the output you get. There is the branch name—the default branch—master, followed by an indicator that this is the first (root) commit and then the first few characters of the SHA1 for the commit.



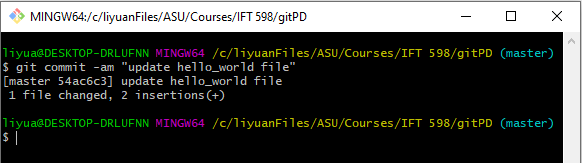
* 1. Edit the python file (Hello\_world.py) by making it print the message "Hello" + name instead of "Hello World"

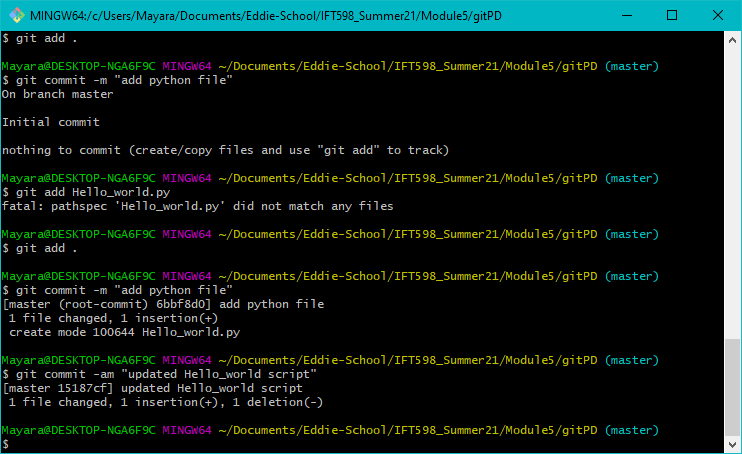




* 1. Stage and commit the file with the shortcut, using whatever text you want for the commit message.

$ git commit –am "comment string" ***(change the commit message to your comment)***

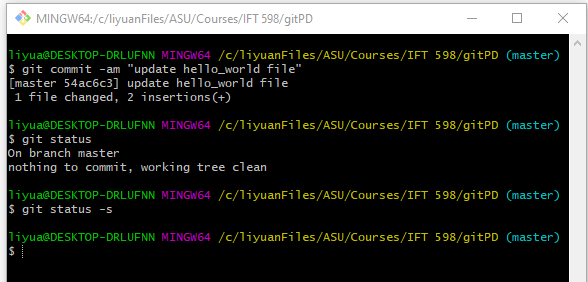


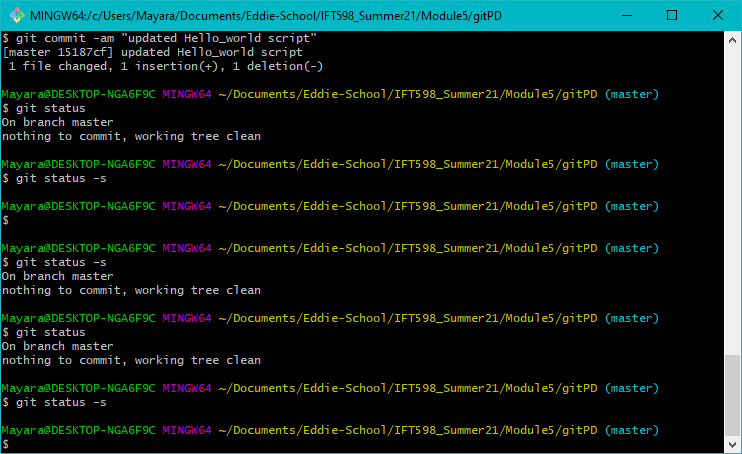


* 1. Run the status command or the short form to see how it looks when you have no changes to be staged or committed.

$ git status

$ git status –s



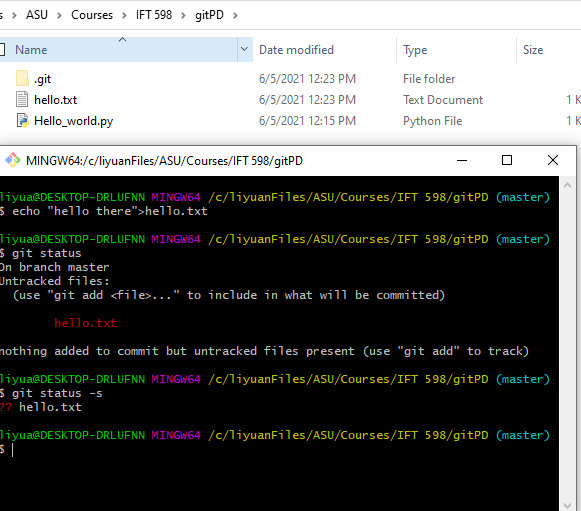


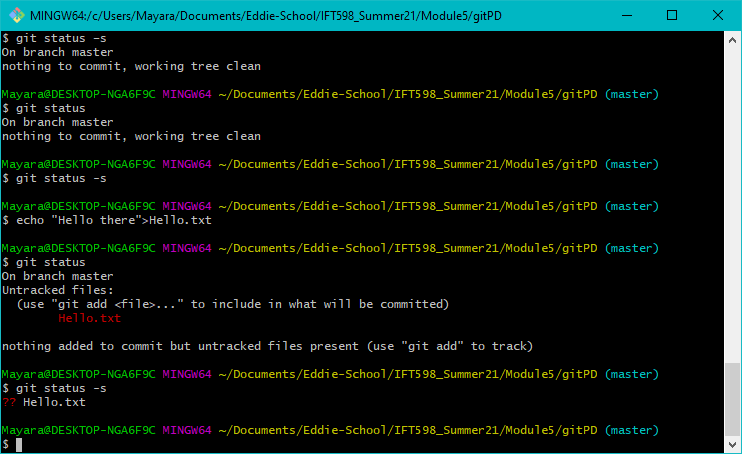
* 1. Create a new file and view the status.

$ echo “hello there” > hello.txt

$ git status

$ git status –s





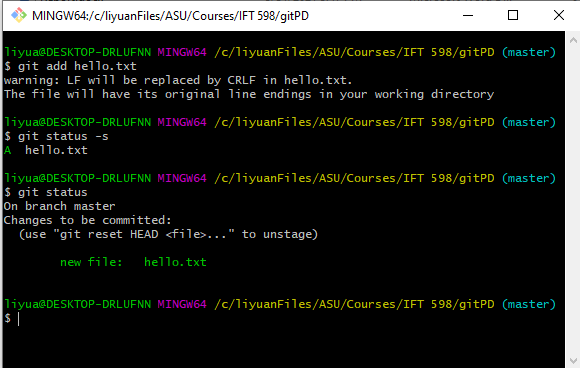
### Question 1: Is the file tracked or untracked?

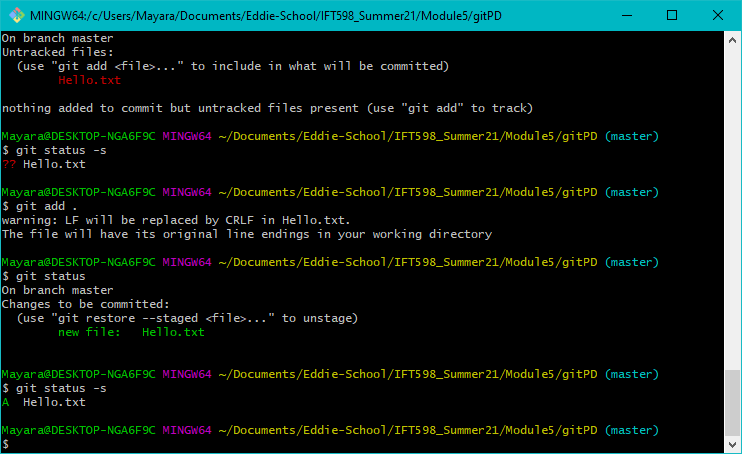
### No, it is not tracked. It is still in unstaged status.

* 1. Stage the file and check its status.

$ git add . (or git add hello.txt)

$ git status (git status –s if you want)





### Question 2:

**Is the file tracked or untracked?**

**Yes, it is tracked. It is in Staged status.**

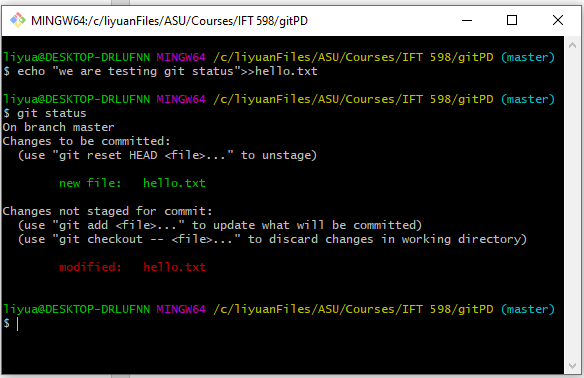
### What does Changes to be committed mean?

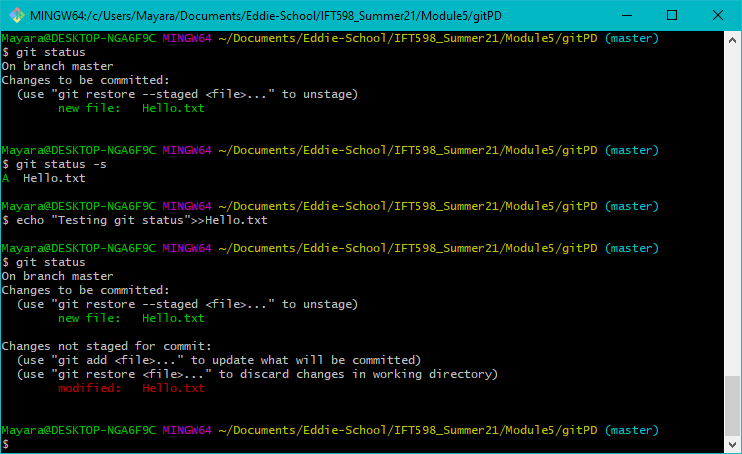
### Git is suggesting commit the content in staging area to next level which is local repository.

* 1. Edit the file *hello.txt* in your working directory and check the status.

$ echo “We are testing git status” >> hello.txt

$ git status





### Questions 3:

* + 1. **Why do you see the file listed twice?**

**One is in the staged area. The other one is still in unstaged area .**

### Where is the version that’s listed as Changes to be committed (in the working directory, staging area, or local repository)?

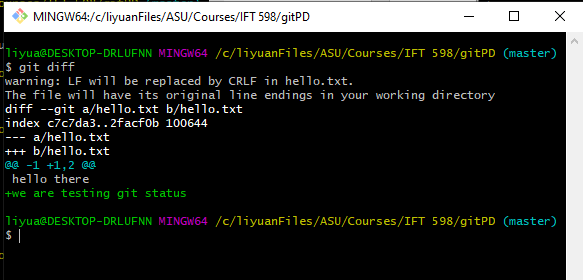
### In staging area.

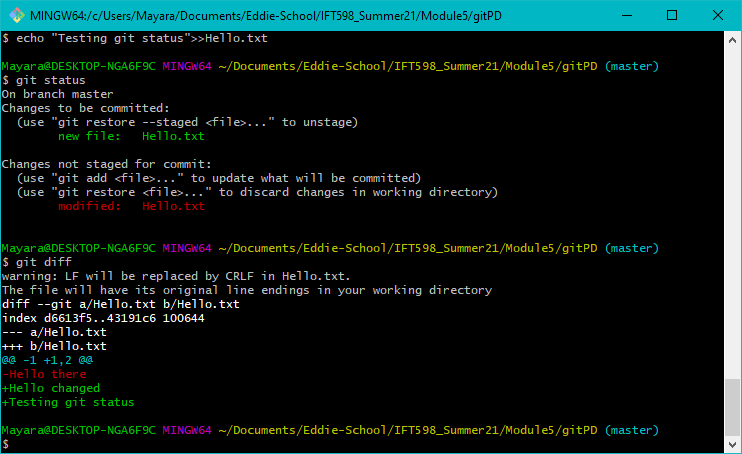
* + 1. **Where is the version that’s listed as Changes not staged for commit (in the working directory, staging area, or local repository)?**

**In working directory.**

* 1. Do a diff between the version in the working directory and the version in the staging area.

$ git diff

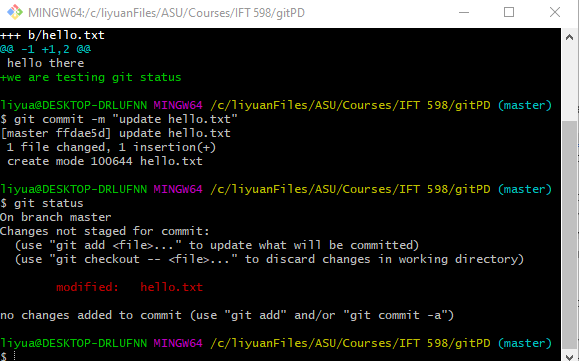


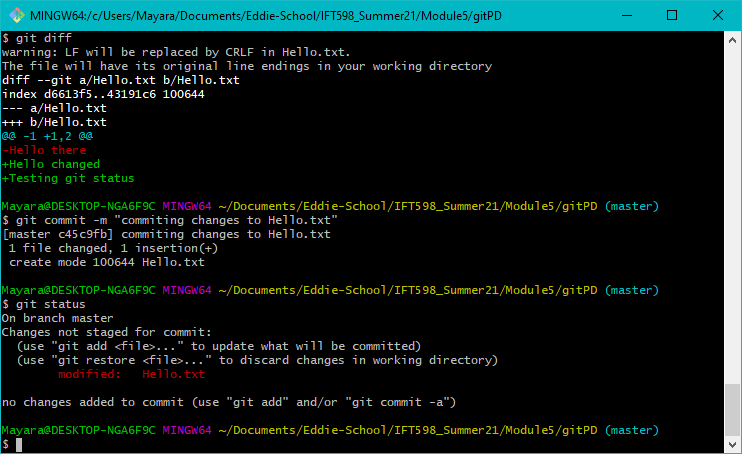


* 1. Go ahead and commit and do another status check.

$ git commit –m "comment" (***(change the commit message to your comment)***

$ git status





### Question 4:

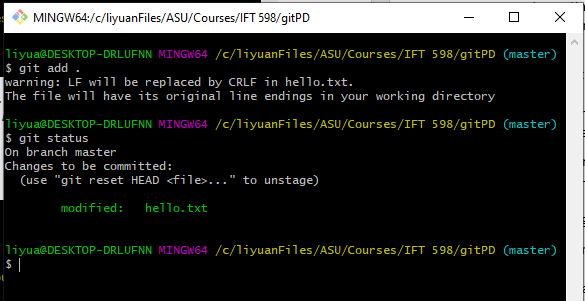
**Which version did you commit: the one in the staging area or the one in the working directory? (Hint: Which one is left [shows up in the status]? Note the changes not staged for commit part of the status message.)**

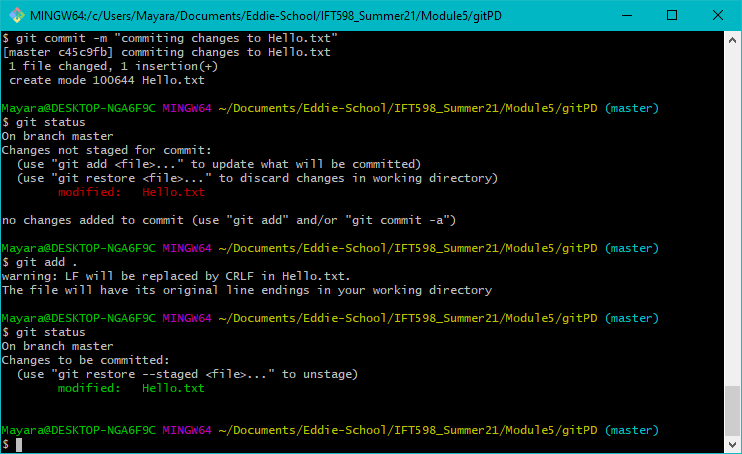
**The one in the staging area**

* 1. Stage the modified file you have in your working directory and do a status check.

$ git add .

$ git status

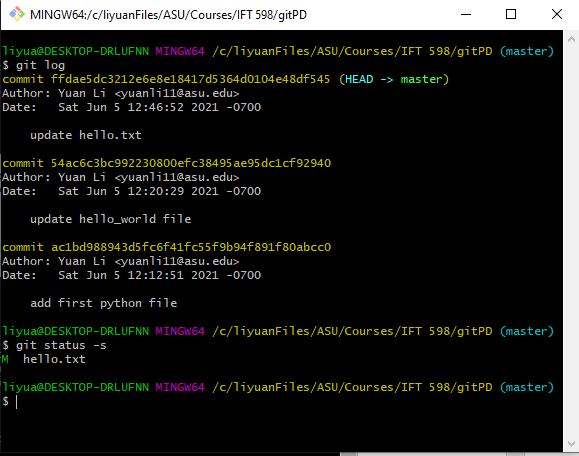


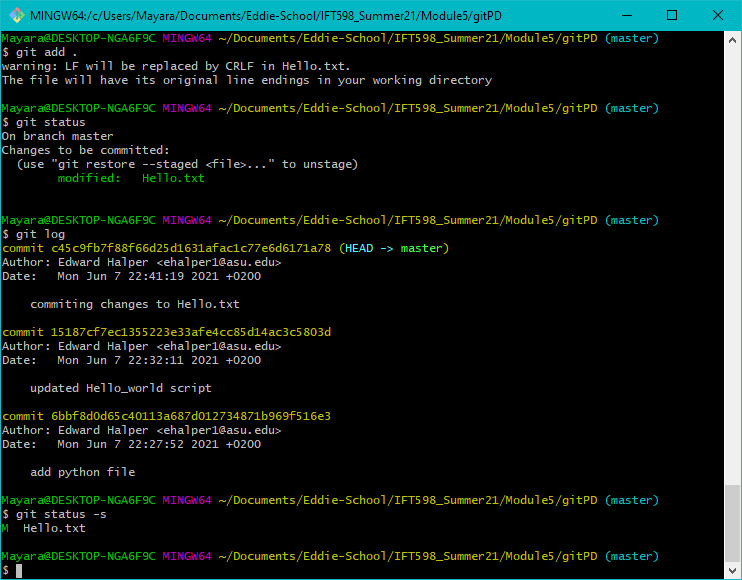


* 1. Look at the history you have so far in your small repository. To do this, run the log command. (In some terminals, your history may be longer than the screen and so you will need to press a key to continue. If you are paging through the log output on a Unix terminal and want to end the listing, press the q key.)

$ git log

$ git status –s

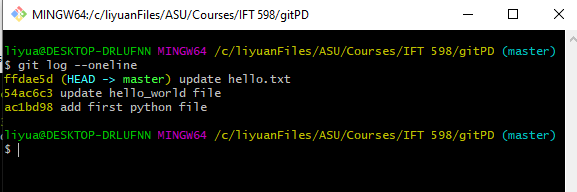


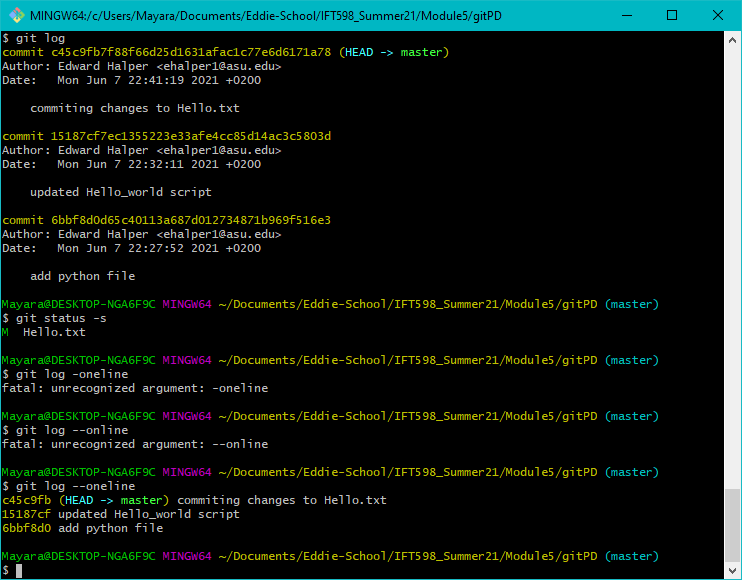


* 1. Often when looking at Git history information, users only want to see the first line of each entry, the subject line. This is why it is important to make that first line meaningful when using Git.

To see only the first line of each log message, you can use the --oneline option:

$ git log –oneline

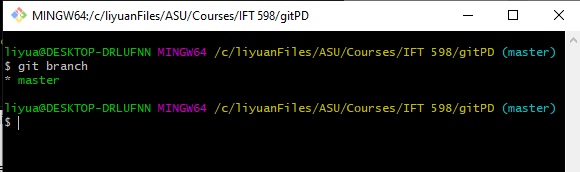


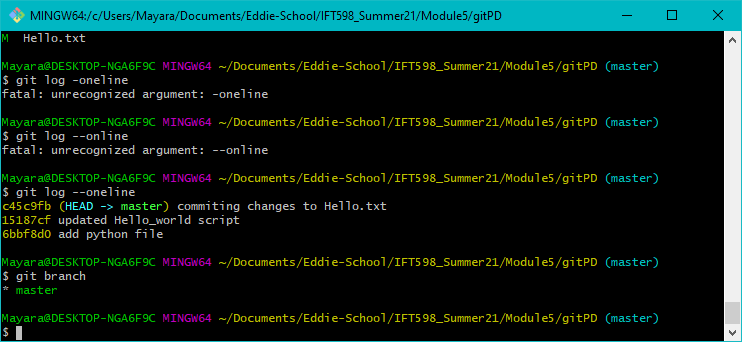


* 1. Use the git branch command to look at what branches you currently have.

$ git branch

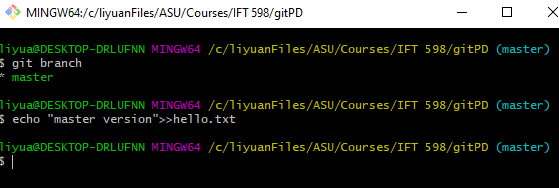
You see a line that says “\* master”. This indicates that there is currently only one branch in your repository: master. The asterisk (\*) next to it indicates that it is the current branch (the one you've switched to and are currently working in). If your terminal prompt is configured to show the current branch, it also says “master”.

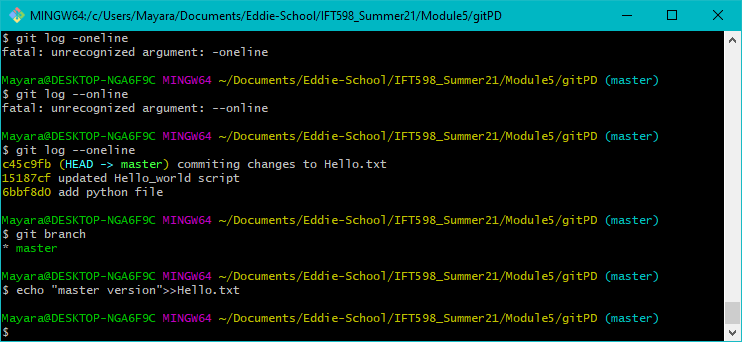




* 1. Before you work with a new branch, you need to update the files in the master branch to indicate that these are the versions on master so it will be easier to see which version you have later. To do this, you can use a similar version of the same way you have been creating and updating other files. Run the following command for each file.

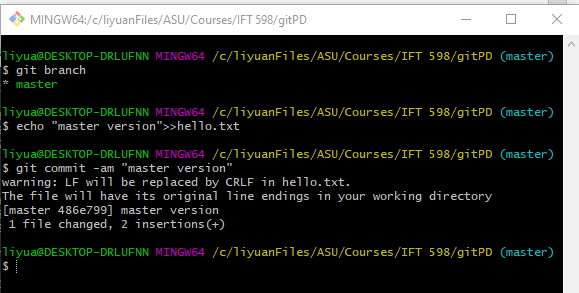
$ echo "master version" >> hello.txt

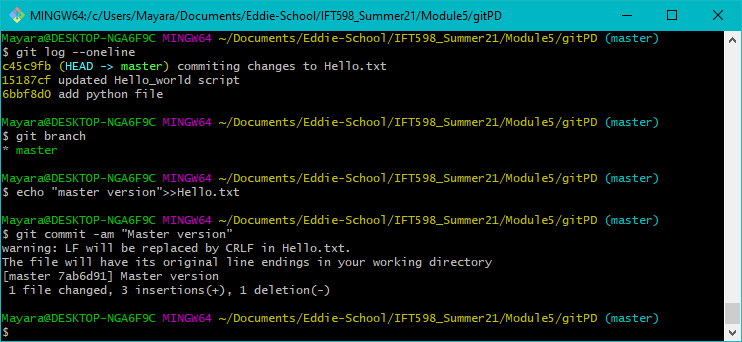




* 1. Stage and commit the updated files. Because these are files that Git already knows about, you can use the following shortcut command:

$ git commit -am "master version"

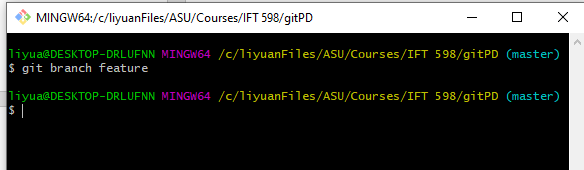


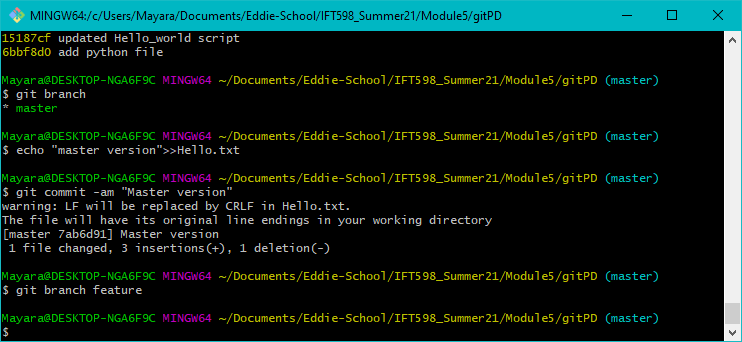


* 1. You have a new feature to work on, so you now create a feature branch with the name feature. Run the following command:

$ git branch feature

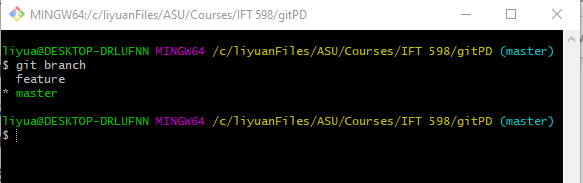
Notice that this command creates the branch, but does not switch to it.

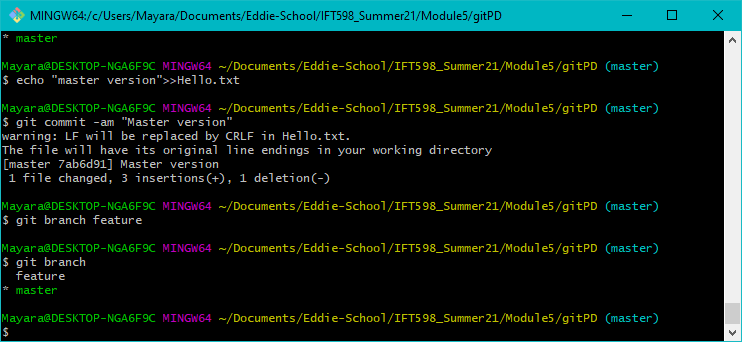




* 1. You can now check what branches you have and which is your current branch.

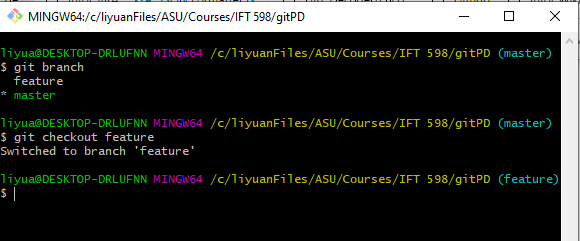
$ git branch

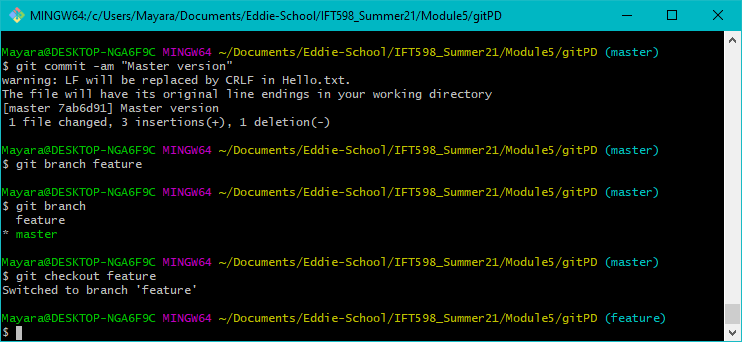




* 1. You can now see your new branch listed. Change into the feature branch to do some work:

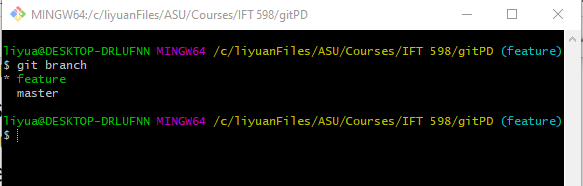
$ git checkout feature

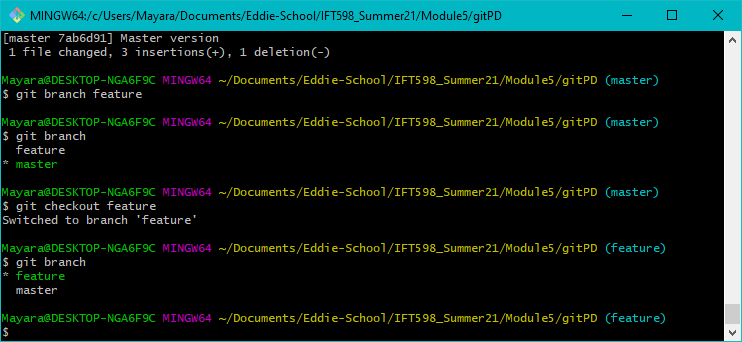




* 1. To verify that you're on the feature branch, run the following command, and observe that the asterisk (\*) is next to that branch:

$ git branch

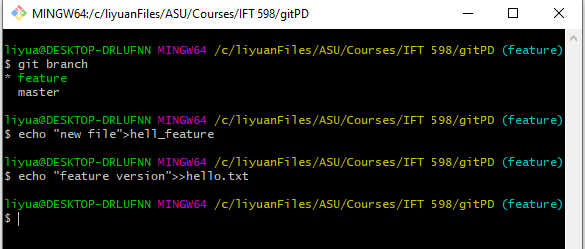


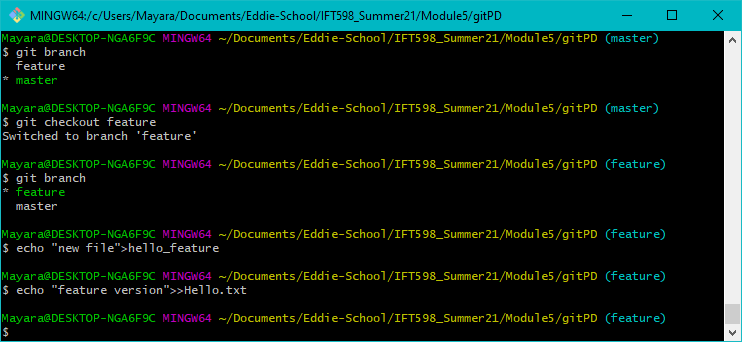


* 1. create a new file and then update the files in the feature branch to indicate that they are the feature branch version.

$ echo "new file" > hello\_feature

$ echo "feature version" >> hello.txt

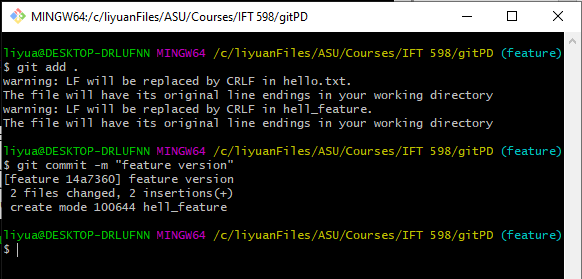


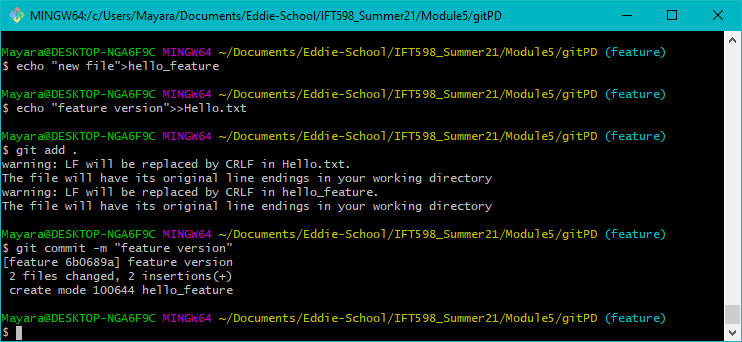


* 1. When you're done, stage and commit your changes.

$ git add .

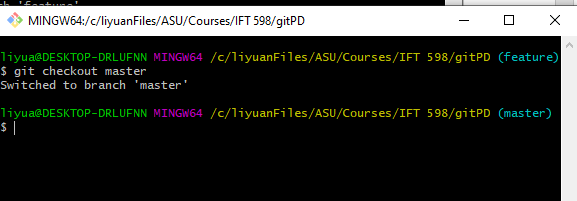
$ git commit -m "feature version"

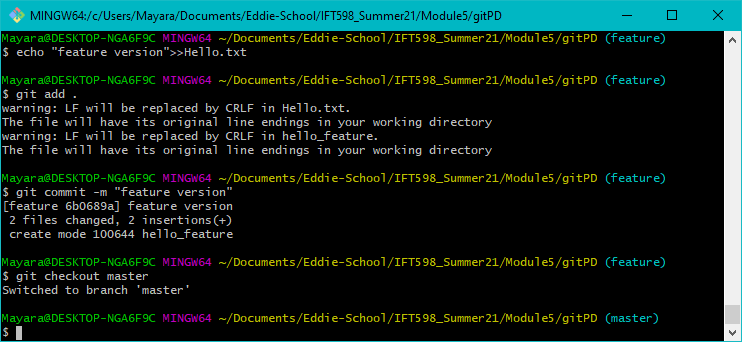




* 1. switch back to the master branch.

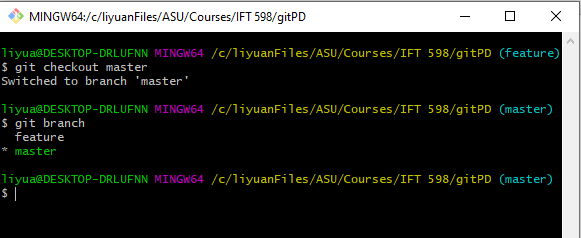
$ git checkout master

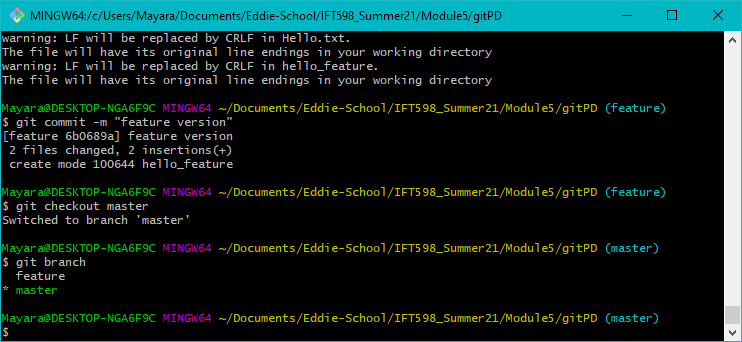




* 1. Verify that you're on the correct branch.

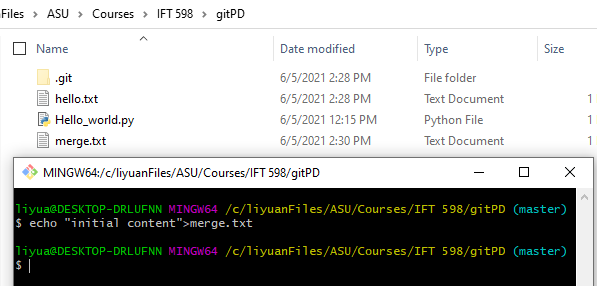
$ git branch

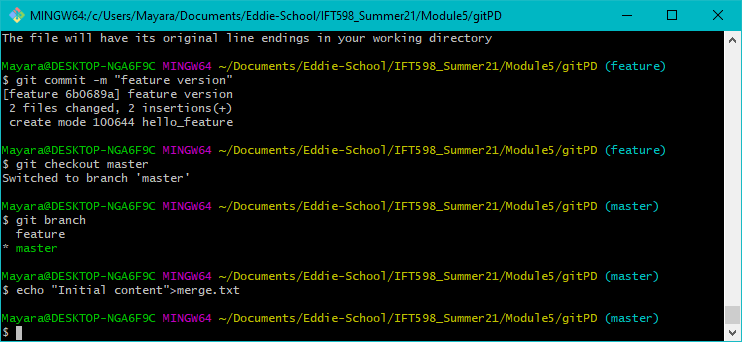




* 1. Create a new one-line file.

$ echo "Initial content" > merge.txt

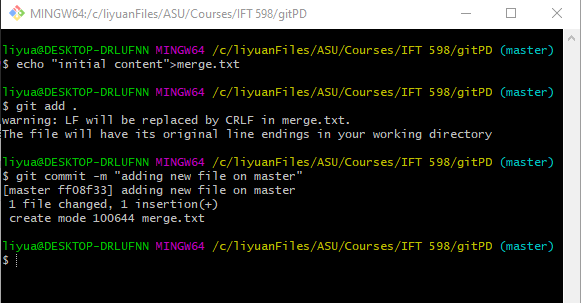


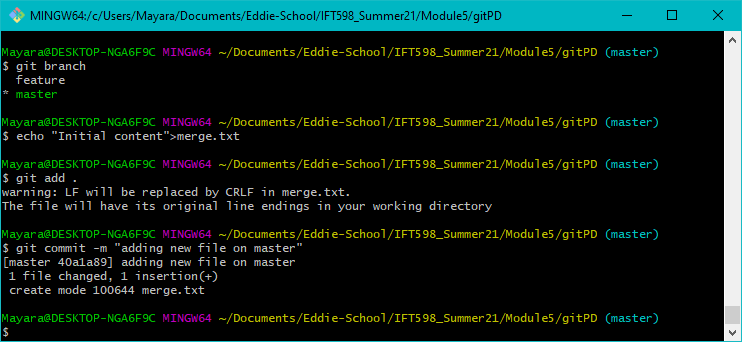


* 1. Stage and commit the file on the master branch.

$ git add .

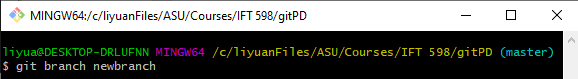
$ git commit -m "adding new file on master"

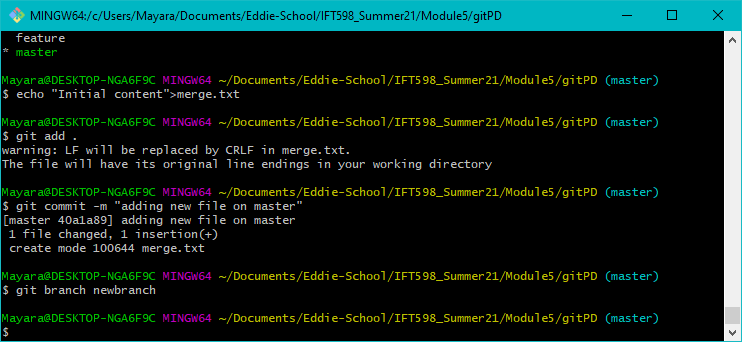




* 1. Create a new branch, but don't switch to it yet. (You can use whatever branch name you want.)

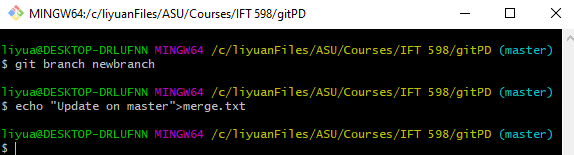
$ git branch newbranch

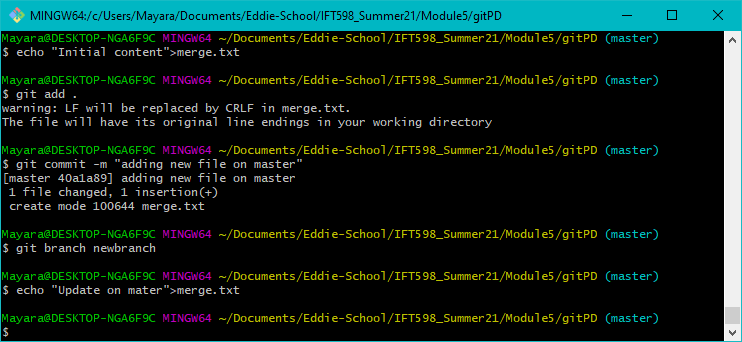




* 1. Change the same line in the new file (still on the master branch).

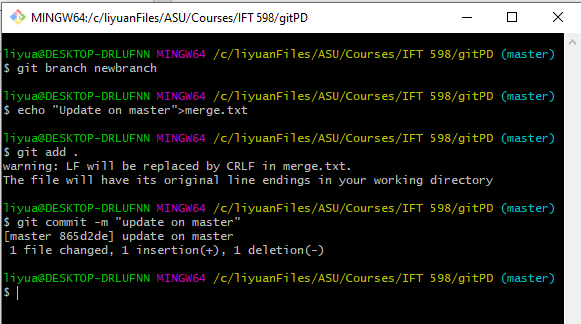
$ echo "Update on master" > merge.txt

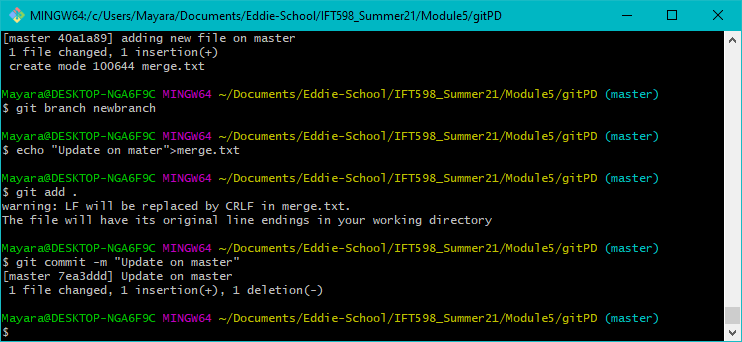




* 1. Stage and commit that change (still on the master branch).

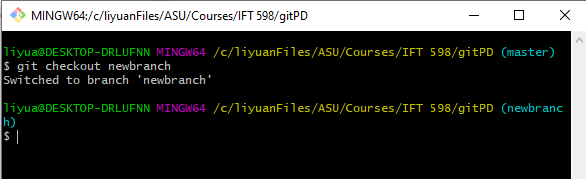
$ git add .

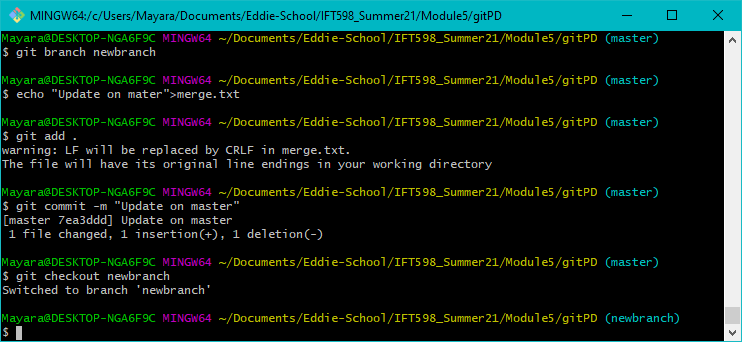
$ git commit -m "update on master"



* 1. Switch to your new branch.

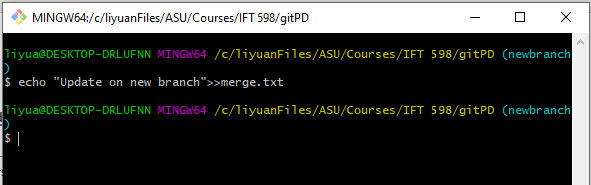
$ git checkout newbranch

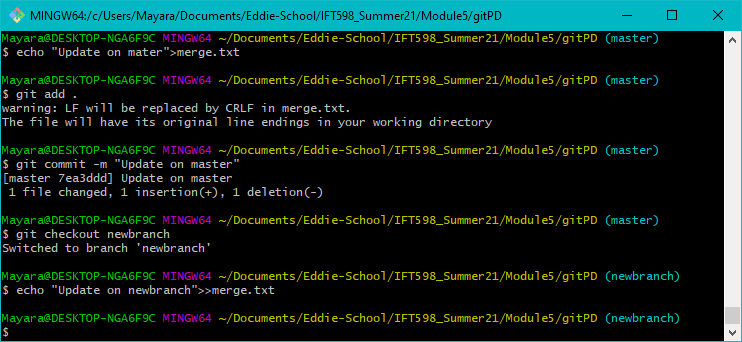




* 1. On the new branch, make a change to the same line of the same file.

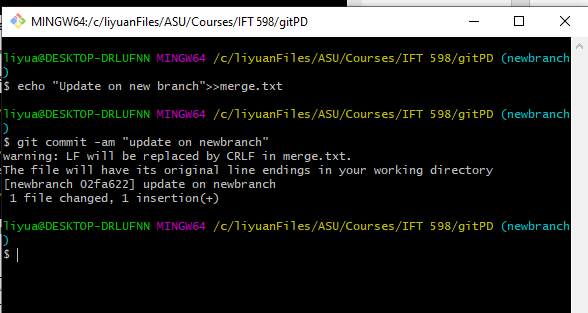
$ echo "Update on newbranch" >> merge.txt

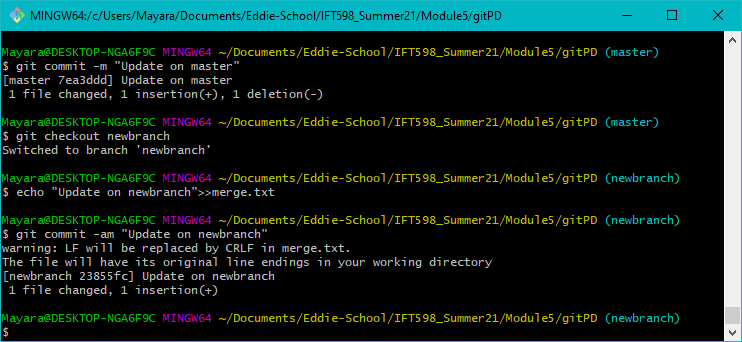




* 1. Stage and commit the file with the change on the new branch.

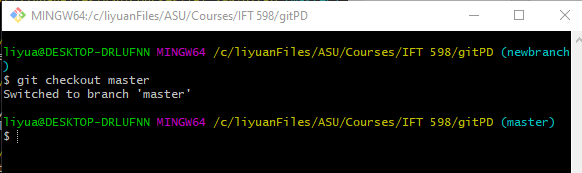
$ git commit -am "update on newbranch"

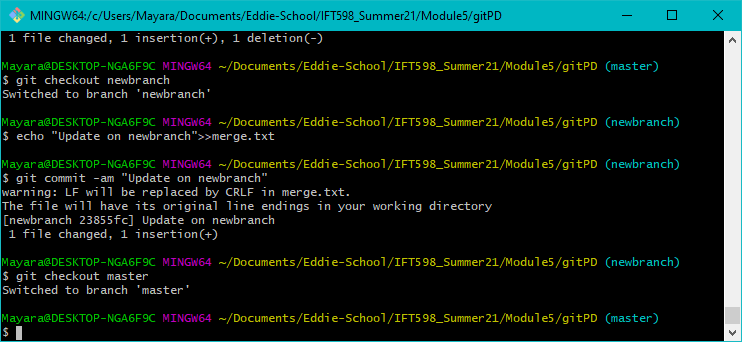




* 1. Switch back to the master branch

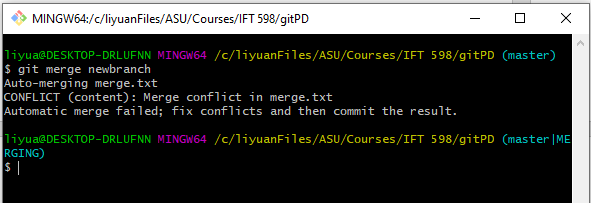
$ git checkout master

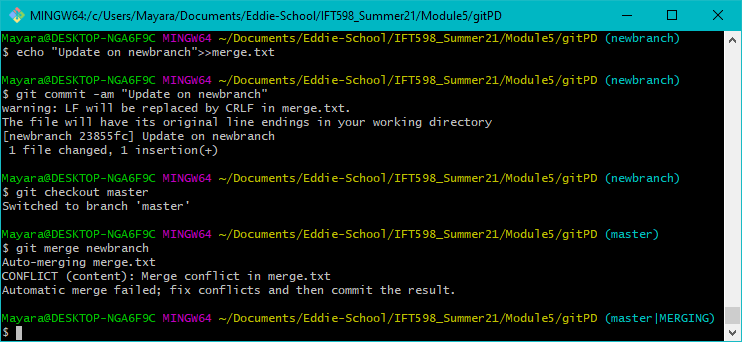




* 1. Merge your new branch back into the master branch. (Git attempts to merge the new branch into the master branch.) You will end up with a merge conflict after this.

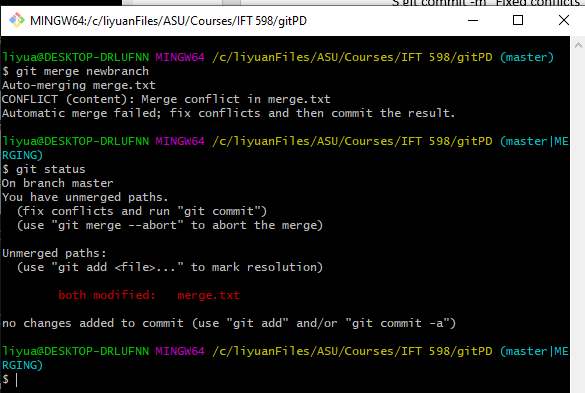
$ git merge newbranch

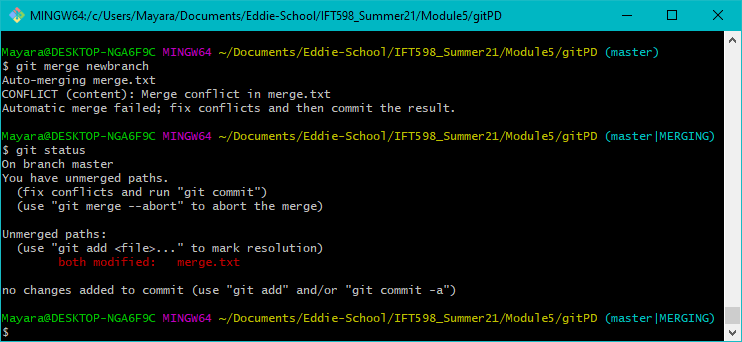




* 1. Check the status of your files in Git. Note the information that Git provides to you about the conflict.

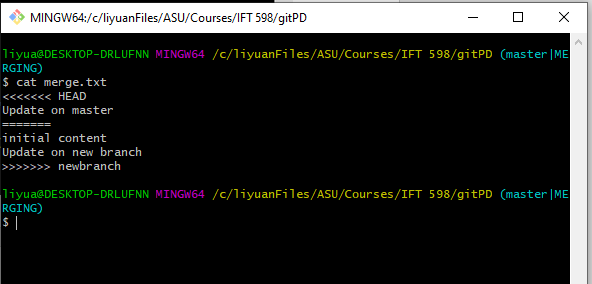
$ git status

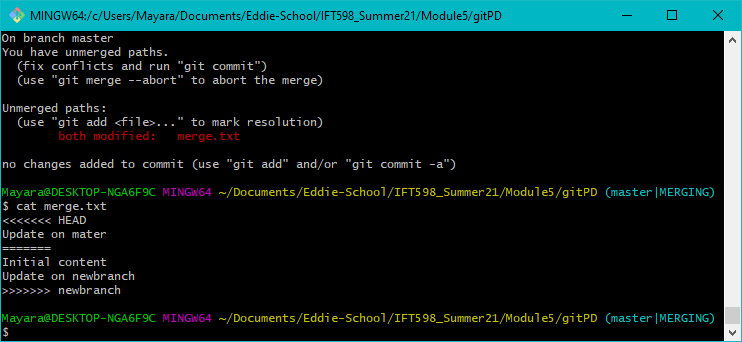




* 1. Look at the local file and note the conflict markers

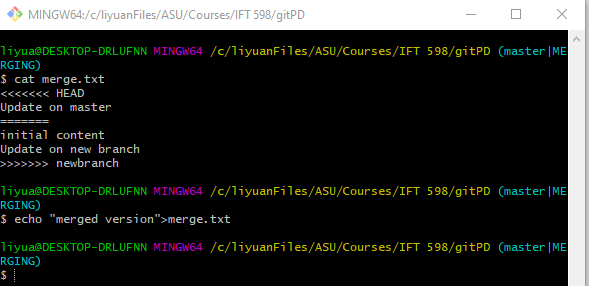
$ cat merge.txt

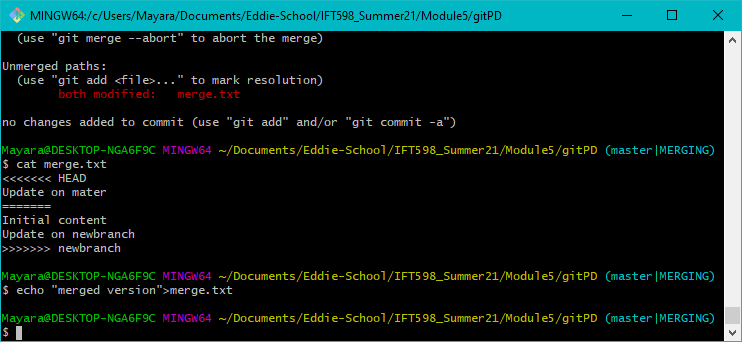




* 1. Resolve the conflict in the file in the working directory. (For simplicity, you can just write over it to simulate that the conflict has been resolved.)

$ echo "merged version" > merge.txt

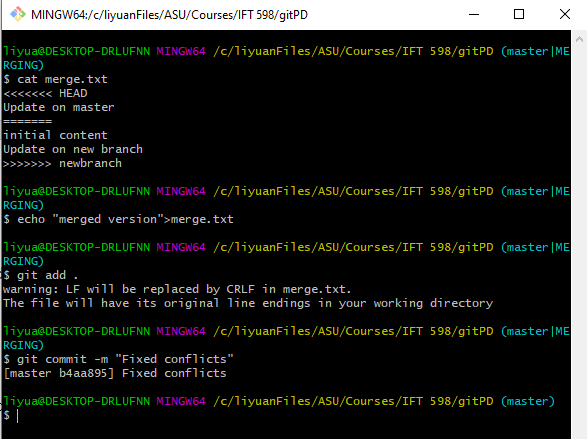


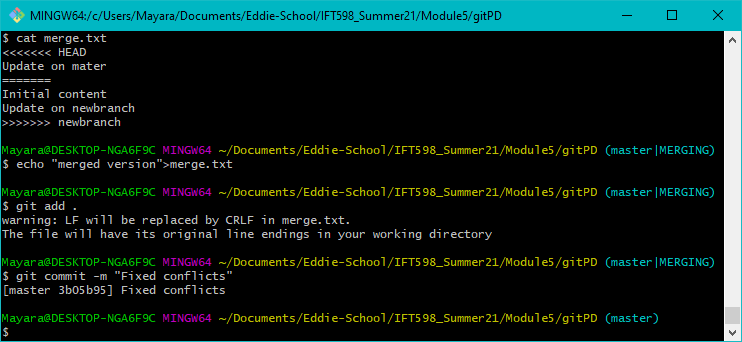


* 1. Stage and commit the fixed file. Note that this has to be done as two separate steps since this was the resolution to a merge conflict.

$ git add .

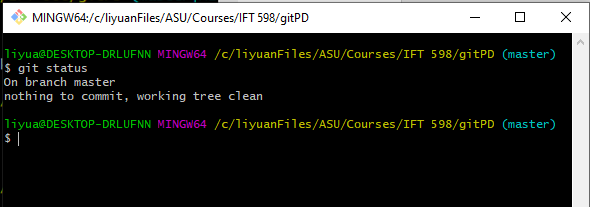
$ git commit -m "Fixed conflicts"

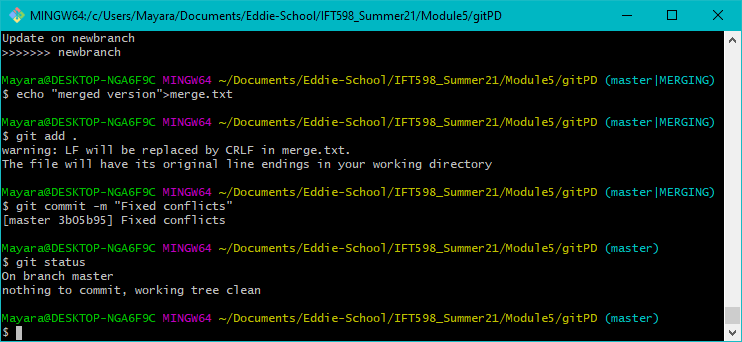




* 1. Check the status to make sure the merge issue is resolved.

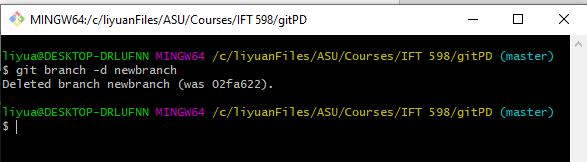
$ git status

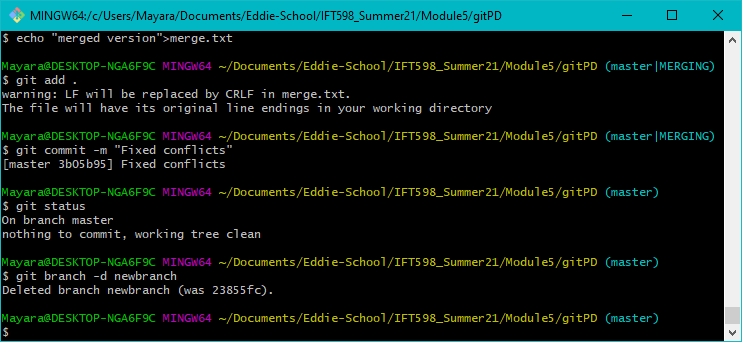




* 1. You're done with your new branch, so delete the branch.

$ git branch -d newbranch





* 1. To remove a commited file, use the rm command

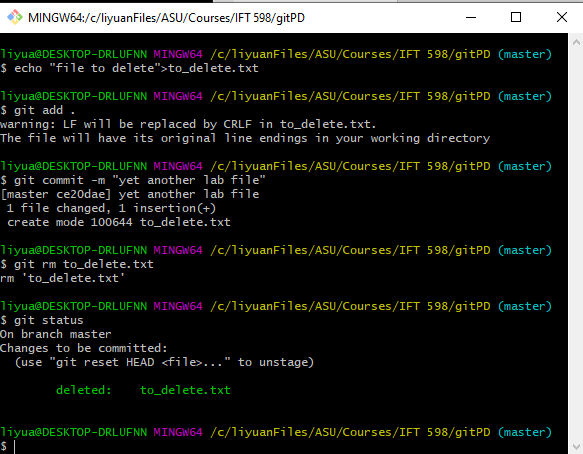
$ echo “file to delete” > to\_delete.txt

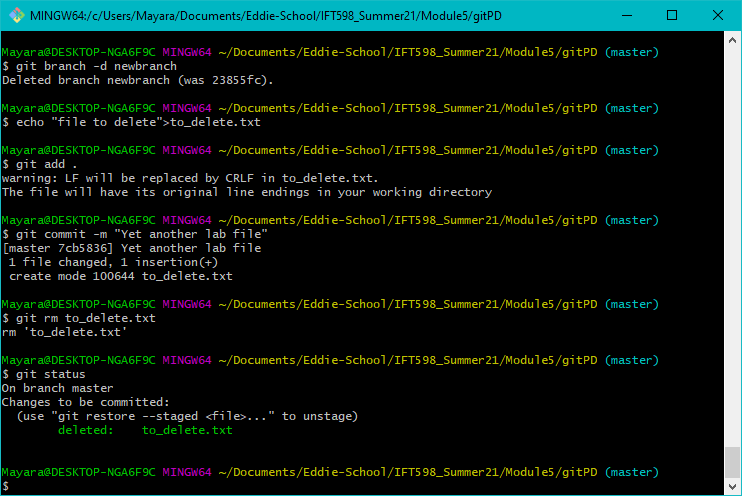
$ git add .

$ git commit -m "yet another lab file"

$ git rm to\_delete.txt

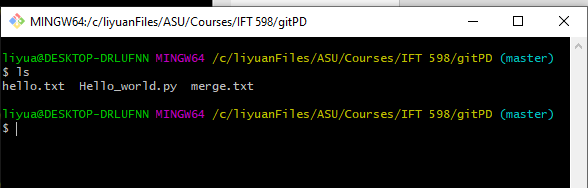
$ git status

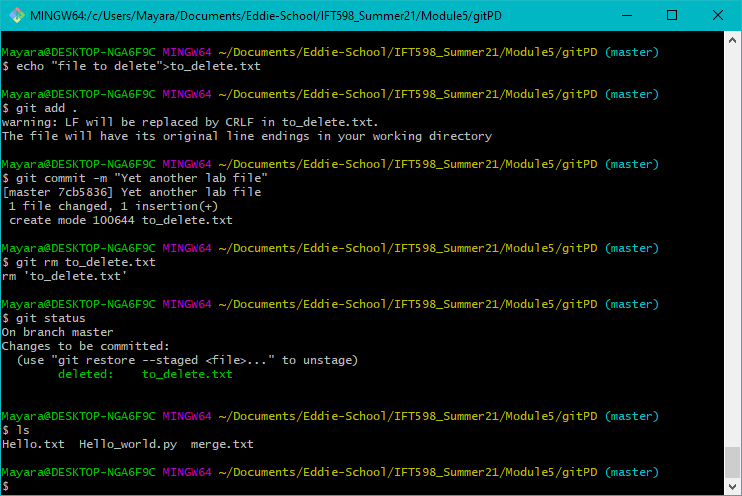




Run the ls command to find out whether the local file is still there.

$ ls



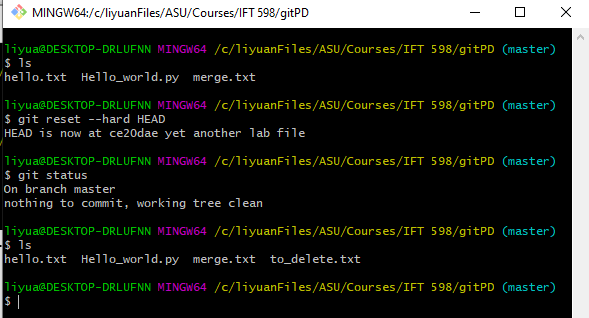


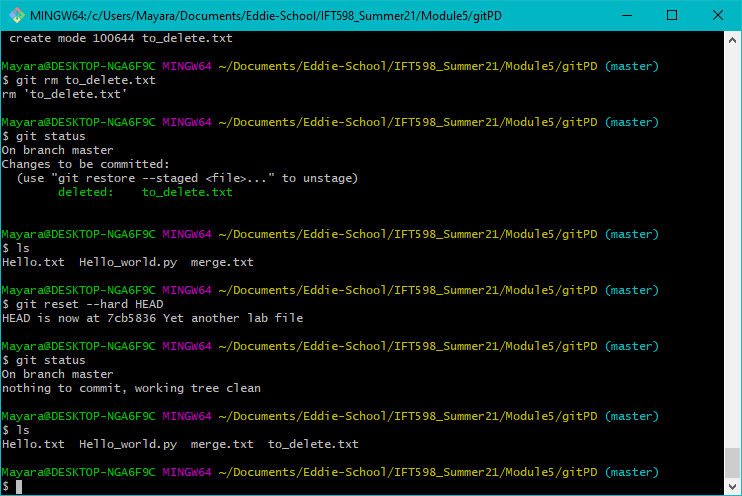
* 1. If You now change your mind, and you want the file back. Use the reset command to do that.

$ git reset --hard HEAD

$ git status

$ ls



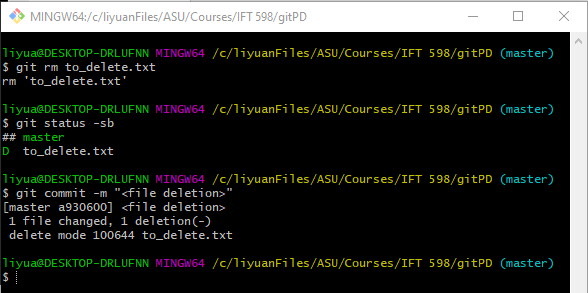


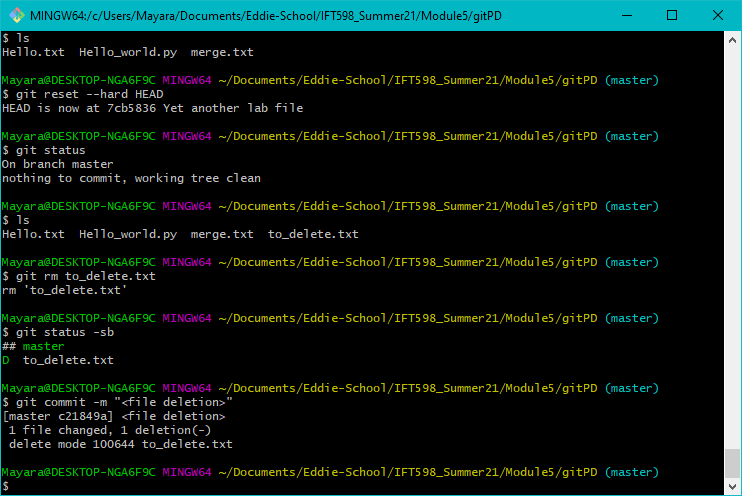
* 1. Delete and commit to the deletion this time.

$ git rm to\_delete.txt

$ git status –sb # Check the status; the file should be staged for deletion.

$ git commit -m "<file deletion>"



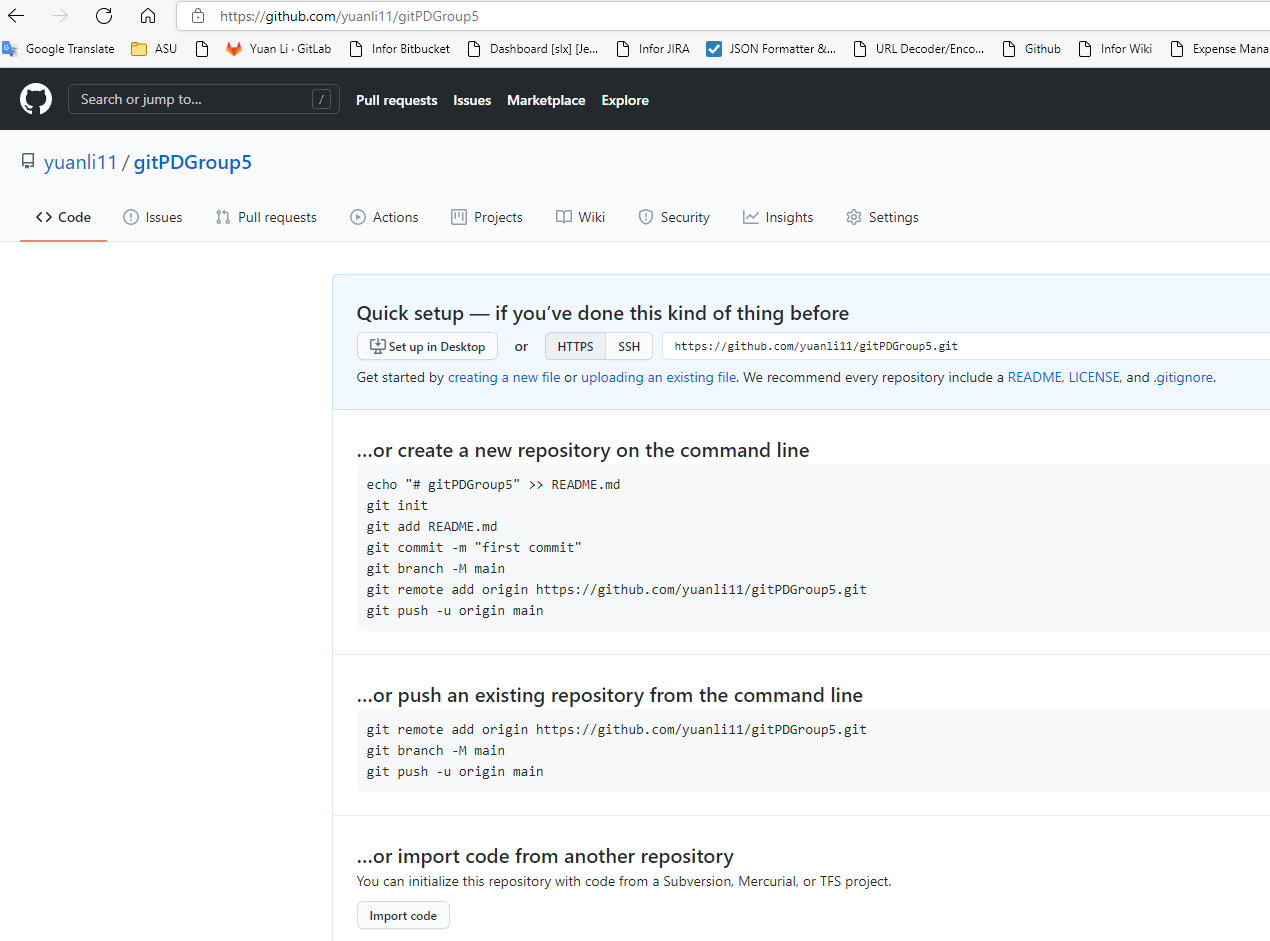


### Setting Up a GitHub Account and pushing to a Repository

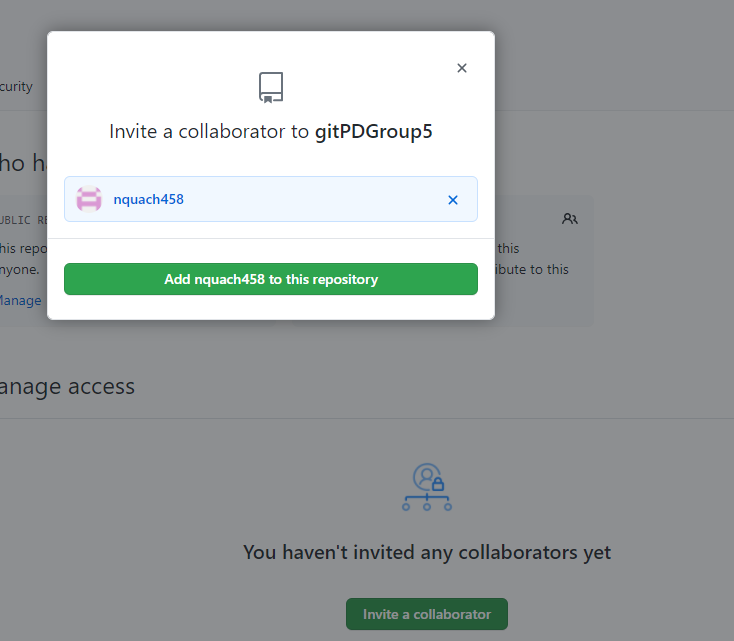
In this section, you'll get some practice with remotes by setting up a GitHub account, forking a repository, and cloning it down to your system to work with.

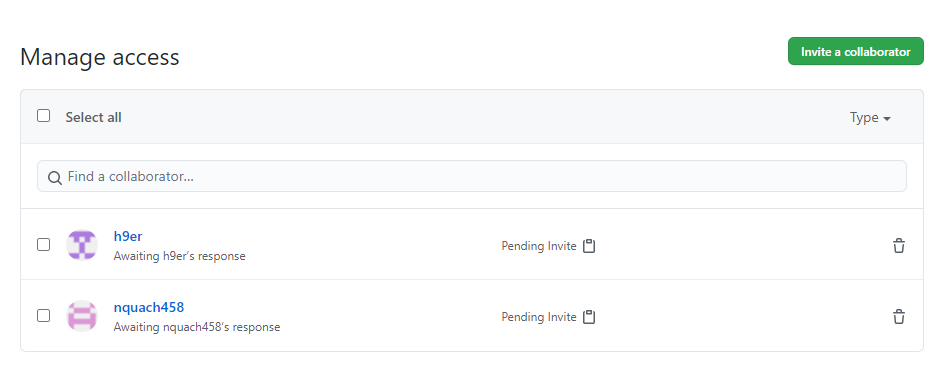
Steps

1. Go to [https://github.com.](https://github.com/)
2. Fill in a username, your email address, and create a strong password
3. Click the Sign up for GitHub button.
4. Accept the defaults on the next screen and click the Continue button.
5. Follow the instructions to verify your email address. Then click the Start a project button.
6. Start a new project by clicking on “New” repository then provide a project name.. Let’s call it ***gitPD<lastName>.*** <lastName> or <TeamID> is basically your Group Identifier. Note that the URL to your project will be [https://github.com/<user>/gitPD<TeamID](https://github.com/%3cuser%3e/japigit_%3cTeamID)>.



1. Add your teammate and your instructor (username: nquach458)

Push the files to the remote repository  




Below image is from Edward Halper’s computer as a collaborator.

