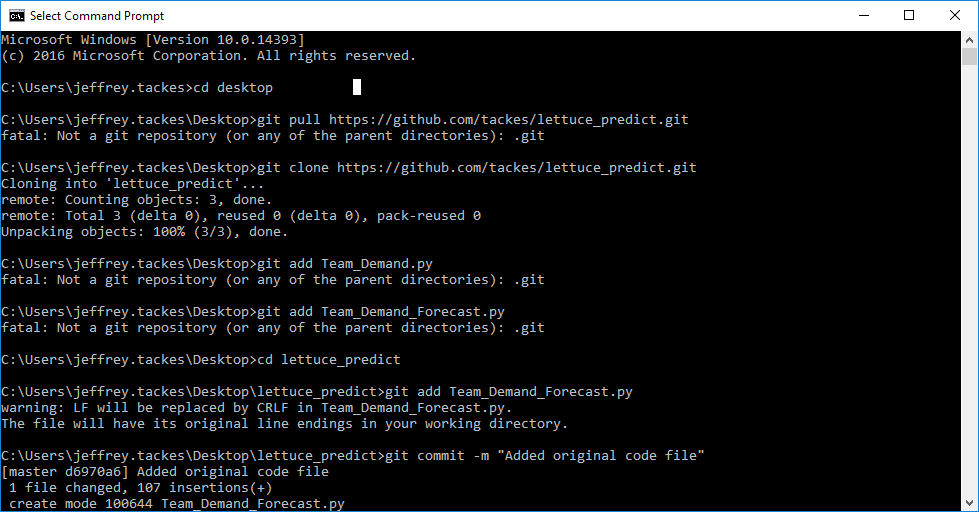
**GITHUB Quick Tutorial**

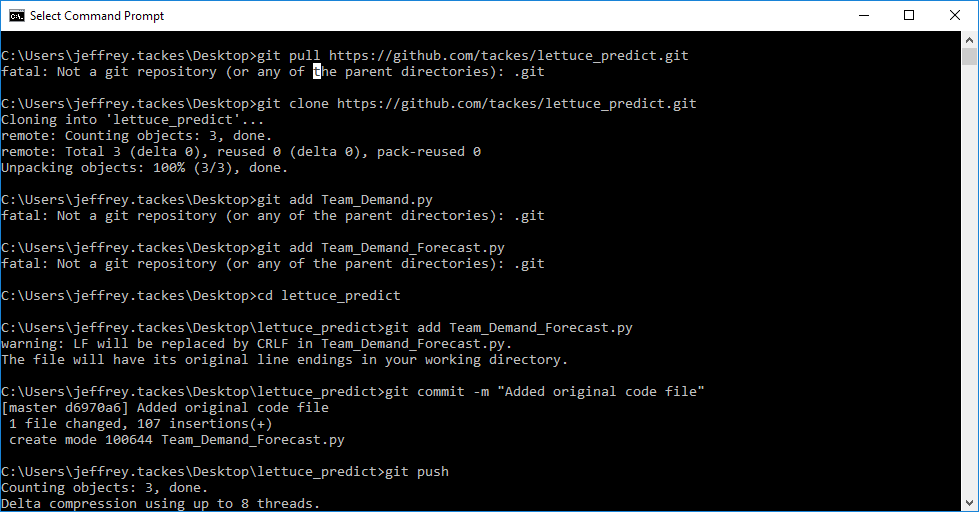
Github Repo - <https://github.com/tackes/lettuce_predict>

1. **FIRST TIME SETUP**
2. Download the desktop client <https://desktop.github.com>
3. On Windows, open your command prompt. On Mac, open the Terminal.
4. CLONE the github repository
   1. First navigate to where you want to store the repository on your computer. Navigate to that folder by typing ***CD Desktop*** (or equivalent folder or series of folders)



* 1. Type ***git clone*** [***https://github.com/tackes/lettuce\_predict.git***](https://github.com/tackes/lettuce_predict.git)

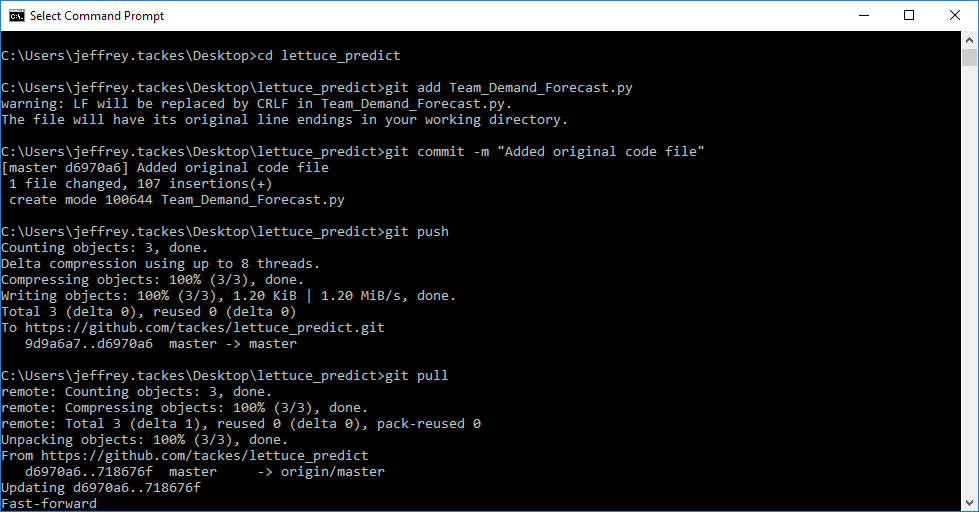
This will close the active repo to your computer and all its components.



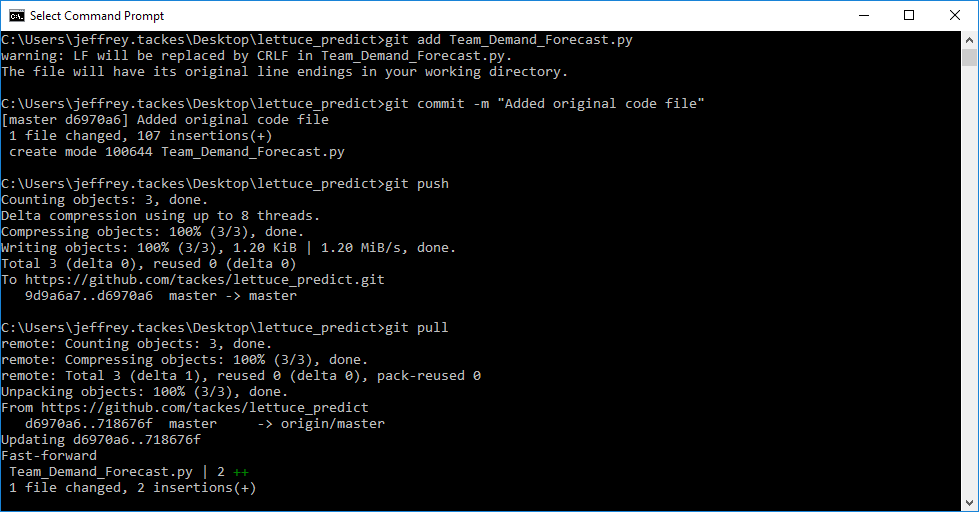
1. **SEND CHANGES BACK TO GITHUB REPOSITORY**

Once you have the file on your computer and you have done your edits to the file and are ready to push it back to the repo, do the following.

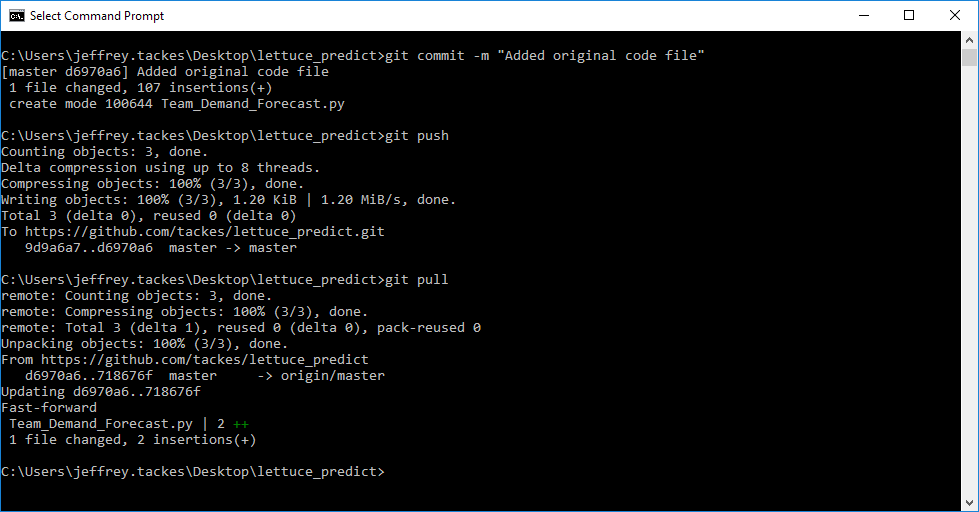
1. In the command prompt, make sure you are in your active repo directory. If you are not, navigate to it by using ***CD <folder name>***



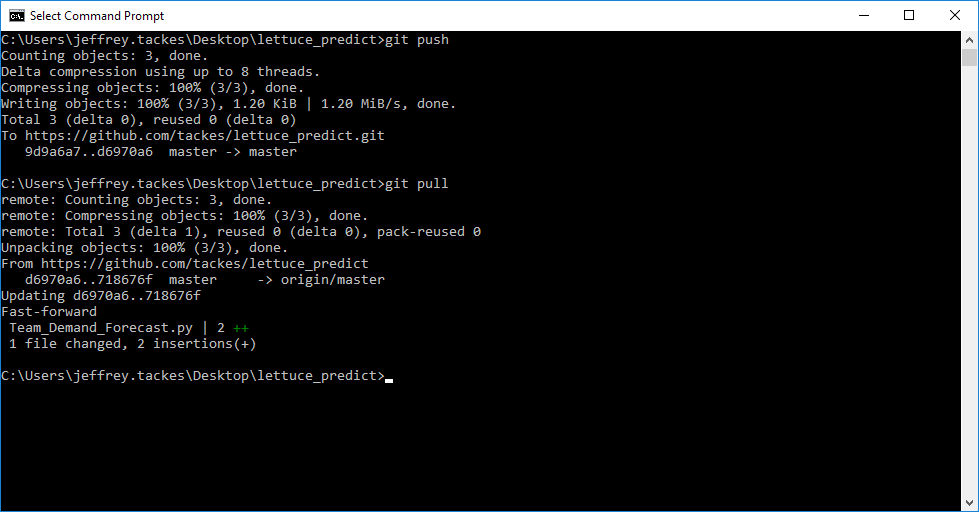
1. Type ***git add Team\_Demand\_Forecast.py***



1. Type ***git commit –m “<write a brief comment to the changes you made>”***



1. Type ***git push***

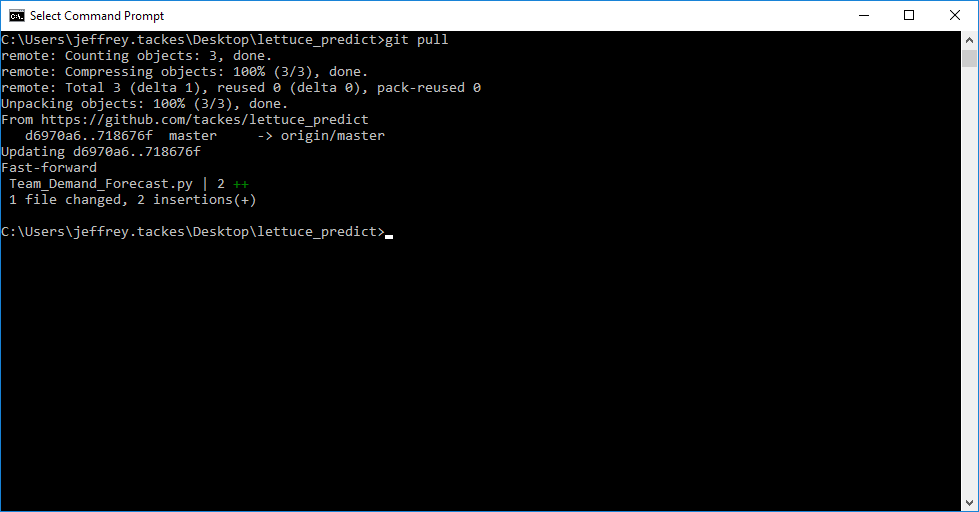


Your changes should now be uploaded to the active repo on github.

1. **UPDATE FOLDER TO THE LATEST FILE IN GITHUB REPOSITORY**

When you are ready to start working in the files again, you will want to update your computer files to the latest version on the github repository. You will need to PULL the files from github into your active git directory on your computer.

1. Navigate to your active directory on your computer in the command prompt
2. Pull the github repository by typing ***git pull***



Displayed in the command prompt will be any changes it noticed from the file that was on your computer vs. the file you just downloaded from the github repository.

**Best practices using GIT**

1. Commit Related Changes
   1. A commit should be a wrapper for related changes. For example, fixing two different bugs should produce two separate commits. Small commits make it easier for other team members to understand the changes and roll them back if something went wrong. With tools like the staging area and the ability to stage only parts of a file, Git makes it easy to create very granular commits.
2. Commit Often
   1. Committing often keeps your commits small and, again, helps you commit only related changes. Moreover, it allows you to share your code more frequently with others. That way it’s easier for everyone to integrate changes regularly and avoid having merge conflicts. Having few large commits and sharing them rarely, in contrast, makes it hard both to solve conflicts and to comprehend what happened
3. Don’t commit Half-done work
   1. You should only commit code when it’s completed. This doesn’t mean you have to complete a whole, large feature before committing. Quite the contrary: split the feature’s implementation into logical chunks and remember to commit early and often. But don’t commit just to have something in the repository before leaving the office at the end of the day. If you’re tempted to commit just because you need a clean working copy (to check out a branch, pull in changes, etc.) consider using Git’s “Stash” feature instead.
4. Test before you commit
   1. Resist the temptation to commit something that you “think” is completed. Test it thoroughly to make sure it really is completed and has no side effects (as far as one can tell). While committing half-baked things in your local repository only requires you to forgive yourself, having your code tested is even more important when it comes to pushing / sharing your code with others.
5. Write good commit messages
   1. Begin your message with a short summary of your changes (up to 50 characters as a guideline). Separate it from the following body by including a blank line. The body of your message should provide detailed answers to the following questions: What was the motivation for the change? How does it differ from the previous implementation? Use the imperative, present tense („change“, not „changed“ or „changes“) to be consistent with generated messages from commands like git merge.
6. Version control is not a backup system
   1. Having your files backed up on a remote server is a nice side effect of having a version control system. But you should not use your VCS like it was a backup system. When doing version control, you should pay attention to committing semantically (see “related changes”) – you shouldn’t just cram in files.