Helpful Git Notes:

```
PS C:\Users\jay\Desktop\Simform documents\JayFolder\GitCourse> git ls-files
file1.py
file2.py
file3.py
file4.py
file5.py
file6.py
file7.py
PS C:\Users\jay\Desktop\Simform documents\JayFolder\GitCourse> git ls-files -s
100644 b7363907fb0a8bfb7764233e7869dc35c9dd67b5 0       file1.py
100644 bc989cb451da38bee165359c8f66826512e8e186 0       file2.py
100644 6ade02ca0c9e93f4b340c88392668ffaa0ff402a 0       file3.py
100644 bc989cb451da38bee165359c8f66826512e8e186 0       file4.py
100644 12861586b9b9946af733e9f96e692e9c206eda36 0       file5.py
100644 f8fd1ddf6bf8f53fa5c02a20adf133a92eede89e 0       file6.py
100644 45a4d4915cea73e32fbf33578551647446baac38 0       file7.py
PS C:\Users\jay\Desktop\Simform documents\JayFolder\GitCourse>
```

What is .DS_Store file git?

DS_Store file? It stands **for Desktop Services Store** and it holds meta information about your folder's thumbnails, settings, etc. . DS_Store files are created any time you navigate to a file or folder from the Finder on a Mac.

Documentation and visuals:

In Git, a *head* is a ref that points to the tip (latest commit) of a branch.

https://www.atlassian.com/git/tutorials

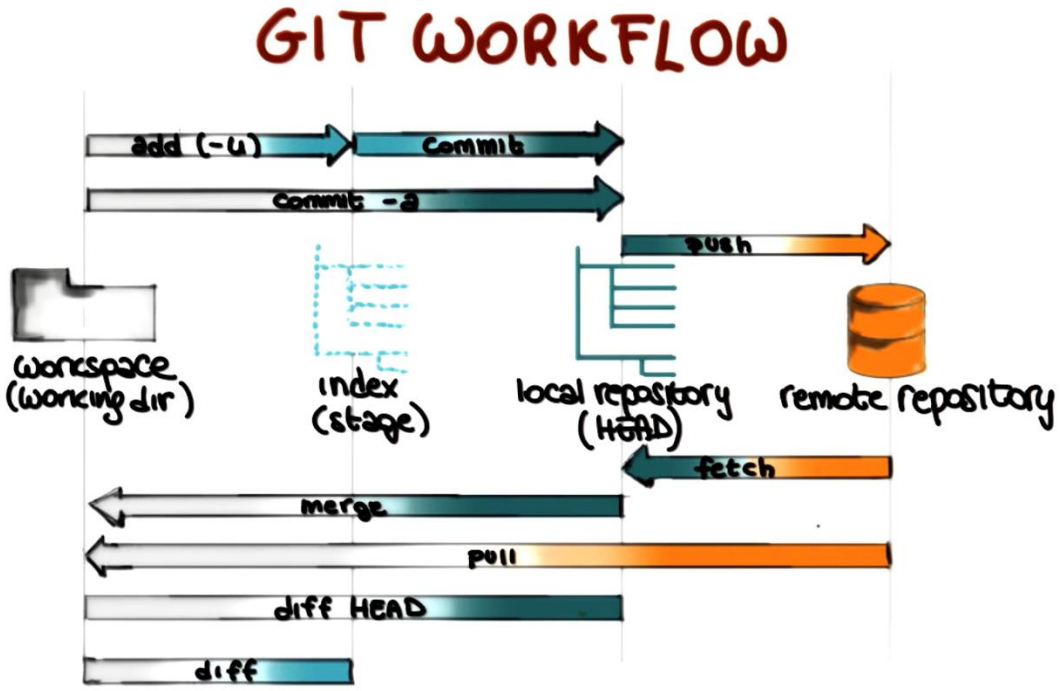
https://ndpsoftware.com/git-cheatsheet.html#loc=stash;

https://git-scm.com/docs

https://git-scm.com/docs/git#_git_commands

Basic commands and brief necessary description (refer doc for detailed info):

GIT WORKFLOW:



1) **git config --global user.name "Jay Rathod"**

   git config –global user.email "jayneversettle@gmail.com"

2) **git init**

   This command is used to initialize git repository

3) **git add file1.txt file2.java file3.py**

   If we want to add specific file name then just give the file name, else if you want to add(stage) all the files then just use:

git add .

4) git status

Shows which files are staged and which are unstage.

5) *DIFFERENCE BETWEEN* ***GIT CLONE*** *AND* ***GIT PULL*** *IS:*

- **git clone git://github.com/username/reponame.git**

Is how you get a local copy of an existing repository to work on. It is usually only used once for a given repository, unless you want to have multiple copies of it around.

whereas,

- **git init**

git remote add origin git://github.com/username/reponame.git

git fetch –all

git pull origin master

here the word origin is an alias for the git repo name so that we don't have to copy long url all the time.

And master is the name of the main branch that exist in our local computer.

Git pull (or git fetch + git merge) is how you update that local copy with new commit from the remote repository. If you are collaborating with others, you will run this command frequently.

AND THE DIFFERENCE BETWEEN GIT FETCH AND GIT PULL IS:

- *When you use pull, Git tries to automatically merge. It is context sensitive, so Git will merge any pulled commits into the branch you are currently working on. pull automatically merges the commits without letting you review them first. If you don't carefully manage your branches, you may run into frequent conflicts.*

- *When you fetch, Git gathers any commits from the target branch that do not exist in your current branch and stores them in your local repository. However, it does not merge them with your current branch. This is particularly useful if you need to keep your repository up to date, but are working on something that might break if you update your files. To integrate the commits into your current branch, you must use merge afterwards.*

6) git reset file.txt

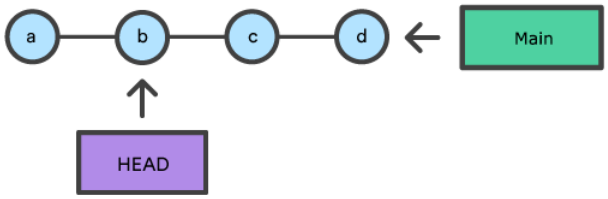
git reset after we have mistakenly staged file, unstages a file.

Main difference between git checkout and git reset:

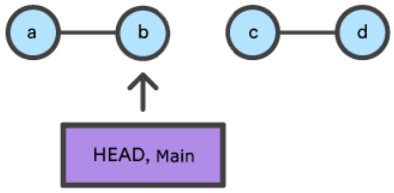
At a surface level, git reset is similar in behavior to git checkout. Where git checkout solely operates on the HEAD ref pointer, git reset will move the HEAD ref pointer and the current branch ref pointer. To better demonstrate this behavior consider the following example:

This example demonstrates a sequence of commits on the main branch. The HEAD ref and main branch ref currently point to commit d. Now let us execute and compare, both git checkout b and git reset b.

git checkout b

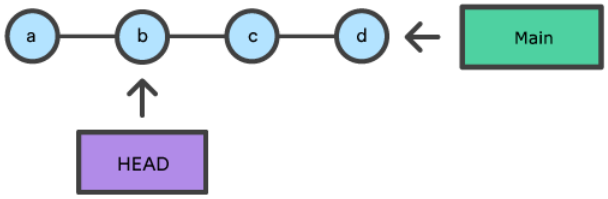


With git checkout, the main ref is still pointing to d. The HEAD ref has been moved, and now points at commit b. The repo is now in a 'detached HEAD' state.

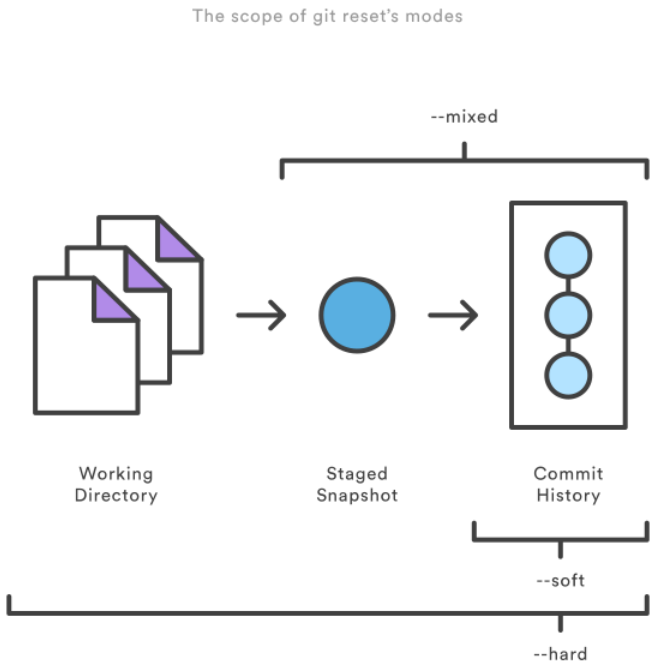


git reset b

Comparatively, git reset, moves both the HEAD and branch refs to the specified commit. In addition to updating the commit ref pointers, git reset will modify the state of the three trees. The ref pointer modification always happens and is an update to the third tree, the Commit tree. The command line arguments --soft, --mixed, and --hard direct how to modify the Staging Index, and Working Directory trees.



Reset can be used for:



1) Going permanently back to past staged or unstaged in the past

- **git reset –hard** : This is the most direct, DANGEROUS, and frequently used option. When passed --hard The Commit History ref pointers are updated to the specified commit. Then, the Staging Index and Working Directory are reset to match that of the specified commit. Any previously pending changes to the Staging Index and the Working Directory gets reset to match the state of the Commit Tree. This means any pending work that was hanging out in the Staging Index and Working Directory will be lost.

I.e. it will delete the last commit and bring the staged and working directory to the previous commit.

- **git reset –mixed** : This is the default operating mode. The ref pointers are updated. The Staging Index is reset to the state of the specified commit. Any changes that have been undone from the Staging Index are moved to the Working Directory. Let us continue.
- **git reset –soft** : When the --soft argument is passed, the ref pointers are updated and the reset stops there. The Staging Index and the Working Directory are left untouched.

soft takes us back to time but doesn't touch the staging index of the time it has taken us to.

2) Removing local commits

git reset –mixed alone would unstaged commits and git reset –mixed commit_hash would go to that commit and remove the files that we had staged at that point of time.

If you decide to go two commits back commits **git reset --hard HEAD~2**

3)Unstaging a file

git reset file_name will remove the file from the unstaged area.

git reset –mixed and git reset are same.

7) git diff and git diff --staged

git diff shows what is changed but not committed

whereas **git diff –staged** shows what is staged but not yet committed

8) git commit -m “commit message”

command to commit and give commit message.

9) git branch

List your branches. a * will appear next to the currently active branch.

10) git branch branchName1

create a new branch at the current commit

11) git checkout filename.txt

Even if you would have made changes to your uncommitted file and saved it, this command will take the content of the file, when the code was last committed

Whereas,

git checkout branchName1

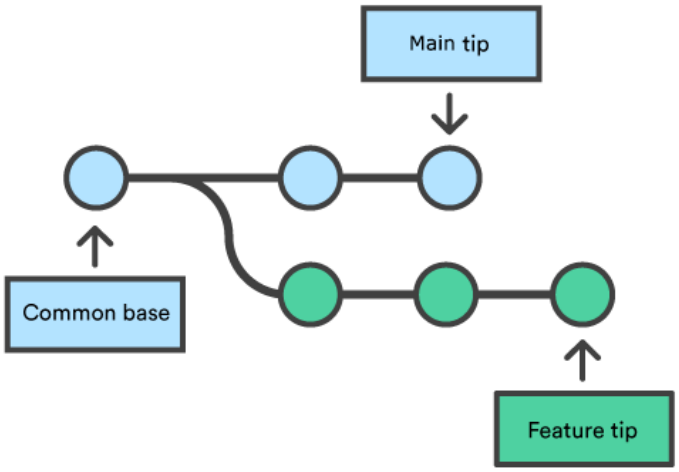
This command takes you to that particular branch after you have created it using **git branch branchName1**

12) git merge branchName1

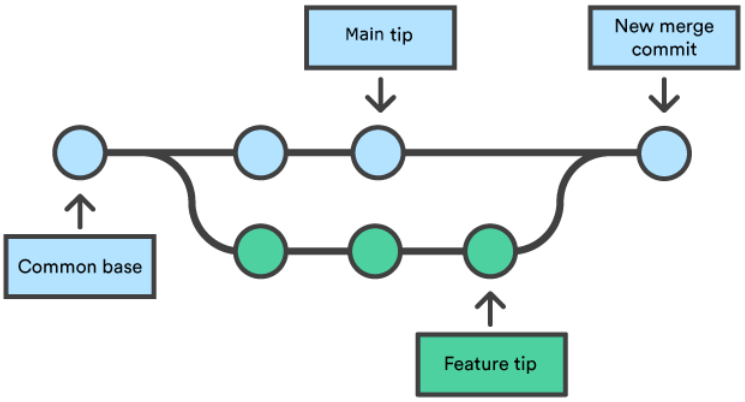
This command will merge the specified branch's history into current one.

Git merge will combine multiple sequences of commits into one unified history. In the most frequent use cases, git merge is used to combine two branches. The following examples in this document will focus on this branch merging pattern. In these scenarios, git merge takes two commit pointers, usually the branch tips, and will find a common base commit between them. Once Git finds a common base commit it will create a new "merge commit" that combines the changes of each queued merge commit sequence.

Say we have a new branch feature that is based off the main branch. We now want to merge this feature branch into main.



Invoking this command will merge the specified branch feature into the current branch, we'll assume main. Git will determine the merge algorithm automatically (discussed below).



Merge commits are unique against other commits in the fact that they have two parent commits. When creating a merge commit Git will attempt to auto magically merge the separate histories for you. If Git encounters a piece of data that is changed in both histories it will be unable to automatically combine them. This scenario is a version control conflict and Git will need user intervention to continue.

13) gitignore

What Kind of Files Should You Ignore?

- Log files
- Files with API keys/secrets, credentials, or sensitive information Useless system files like .DS_Store on macOS
- Generated files like dist folders
- Dependencies which can be downloaded from a package manager
- And there might be other reasons (maybe you make little todo.md files)

You can get an idea for what sort of files to ignore on gitignore.io, by selecting your operating system, text editor or IDE, languages, and frameworks.

How .gitignore Works

Here's how it works. A .gitignore file is a plain text file where each line contains a pattern for files/directories to ignore. Generally, this is placed in the root folder of the repository, and that's what I recommend. However, you can put it in any folder in the repository and you can also have multiple .gitignore files. The patterns in the files are relative to the location of that .gitignore file.

Literal File Names

The easiest pattern is a literal file name, for example:

```
.DS_Store
```

Directories

You can ignore entire directories, just by including their paths and putting a / on the end:

```
node_modules/
logs/
```

If you leave the slash off of the end, it will match both files and directories with that name.

Wildcard

The * matches 0 or more characters (except the /). So, for example, *.log matches any file ending with the .log extension. Another example is *~, which matches any file ending with ~, such as index.html~ You can also use the ?, which matches any one character except for the /.

Negation

You can use a prefix of ! to negate a file that would be ignored.

```
*.log
2!example.log
```

In this example, example.log is not ignored, even though all other files ending with .log are ignored. But be aware, you can't negate a file inside of an ignored directory:

```
logs/
2!logs/example.log
```

Due to performance reasons, git will still ignore logs/example.log here because the entire logs directory is ignored.

Double sAsterisk

- ** can be used to match any number of directories.
- **/logs matches all files or directories named logs (same as the pattern logs)
- **/logs/*.log matches all files ending with .log in a logs directory logs/**/*.log matches all files ending with .log in the logs directory and any of its subdirectories

** can also be used to match all files inside of a directory, so for example logs/** matches all files inside of logs.

Comments

Any lines that start with # are comments:

```
# macOS Files
2.DS_Store
```

https://www.pluralsight.com/guides/how-to-use-gitignore-file

https://www.atlassian.com/git/tutorials/saving-changes/gitignore

14) git alias

git config --global alias.co checkout

Creating the aliases will not modify the source commands. So git checkout will still be available even though we now have the git

co alias. These aliases were created with the --global flag which means they will be stored in Git's global operating system level configuration file. On linux systems, the global config file is located in the User home directory at /.gitconfig.

Using aliases to create new Git commands

A common Git pattern is to remove recently added files from the staging area. This is achieved by leveraging options to the git reset command. A new alias can be created to encapsulate this behavior and create a new alias-command-keyword which is easy to remember:

git config --global alias.unstage 'reset HEAD --'

The preceding code example creates a new alias unstage. This now enables the invocation of git unstage. git unstage which will perform a reset on the staging area. This makes the following two commands equivalent.

git unstage fileA is same as **git reset HEAD -- fileA**

15) git stash list

shows the list of stashes

git stash save “name of the stash”

the saved but not committed changes will disappear and go to stash and to bring those uncommitted saved changes back do:

git stash apply stash@{stashid}

git stash drop stash@{stashid} to delete particular stash.

git stash pop pops the very first stash.

P.S. Works over different branches too. As long as the code makes sense.

16) git tag

git tag is usually used for tagging the versions of the application.

17) git blame

blame can be used to show who committed, commit hash, commit message, date and time of commit.

18) git clean

git clean command operates on untracked files. Untracked files are files that have been created within your repo's working directory but have not yet been added to the repository's tracking index using the git add

command. A convenience method for deleting untracked files in a repo's working directory

git clean -n

This will show you which files are going to be removed without actually removing them.

-f or **--force**

git clean –force to force delete the unstaged files.

19) git revert <first 7 digit of the hash>

git revert goes to the previous commit using the commit hash that you would have specified then recommits, creating a new commit so that the changes that you did in previous commit can become your present source code.

This command will commit the file with the message *Revert "commit message of that commit".* And if you want to give it a name after reverting you can use **git revert –n <first 7 digit of hash>** then the files will be just staged from the previous commit then you need to type commit command and commit message.

20) git reset head filename.txt

It is used to remove files from the staging area.

git reset –hard <commit_hash> will delete the commit that exist ahead of the commit hash.

21) git rm –cached fileName.txt

To make the file unstaged

git rm filename.txt

to delete the files.

22) Git commit –amend

Commits the file that are in the staging area to the very recent commit i.e. head, so that you don't have to make a new commit.

Git commit –amend –no-edit

Does the same but you will have to name it in vim editor.

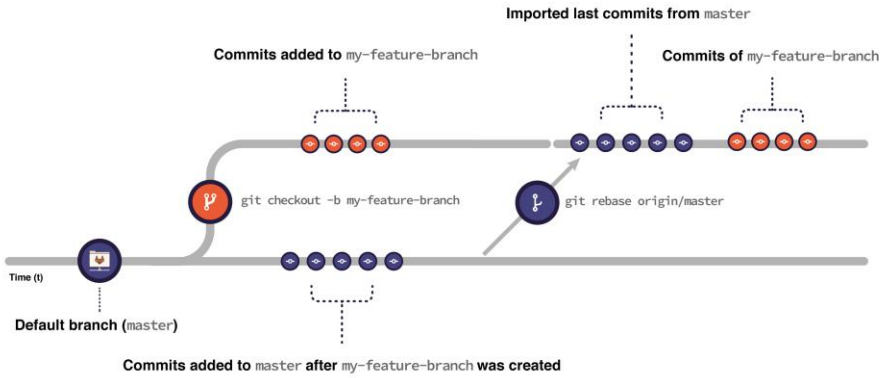
To write something in vim editor type i

To exit vim editor after saving, tap escape and then type :x and to exit not saving :q

23) When you are in feature branch and do **git rebase master**

Rebasing moves the entire feature branch (any specific branch) to begin on top of the master branch, effectively incorporating all of the new commits in the master.

P.S. You do not use rebase in public branch.



24) git reflog

Reference logs, give us the logs of what we have previously done with the branches.

25) git remote -v

shows to and from which repo will the branch and commits be pushed and fetched.

26) Pull request

Pull requests can only be opened between two branches that are different.

If you want to create a new branch for your pull request and do not have write permissions to the repository, you can fork the repository first.

Step 1: To create a pull request you first need to fork someone's repository.

Step 2: ...explained at the bottom.

27) **git log -n 2** will show last 2 commits and **git log** will show all. And **git log –oneline** will show oneline commit details.

28) **git fetch branchName**

git fetch origin by default fetches everything from the remote named "origin" and updates (or creates) the so-called "remote-tracking branches" for that remote. Say, for the remote named "origin" which contain branches named "master" and "feature", running git fetch remote will result in the remote-tracking branches named "origin/master" and "origin/feature" being updated (or created, if they're not exist). You could see them in the output of git branch -a (notice "-a").

Now, the usual Git setup is that (some of) your local branches follow certain remote branches (usually same-named). That is, your local "master" branch follows "origin/master" etc.

So, after you fetched, to see what remote "master" has compared to your local "master", you ask Git to show you exactly this:

git log origin/master ^master

which means «all commits reachable from "origin/master" which do not include commits reachable from "master"» or, alternatively

git log master..origin/master

which has the same meaning. See the "gitrevisions" manual page for more info, especially the "Specifying ranges" part. Also see the examples in the git-log manual page

You're free to customize the output of git log as you see fit as it supports a whole lot of options affecting it.

Note that your local branch might also have commits which the matching remote branch does not contain (yet). To get an overview of them you have to reverse the revisions passed to git log for (hopefully) obvious reasons.

As usual, it's essential to educate yourself to understand the underlying concepts before starting to use a tool. Please do.

https://www.atlassian.com/git/tutorials/syncing/git-fetch

29) git pull origin master

Pulls the updated remote repository to the local repository and puts the remotely bought commits ahead of the first local commit before pull.

This is same as **git pull <remote>** = **git fetch ＜remote＞** followed by **git merge origin/＜current-branch＞**.

30) git cherry-pick

https://www.atlassian.com/git/tutorials/cherry-pick

31) ….add another git command if any….

32) B

33) C

34) D

35) E

Situations in git where you can run into trouble:

1) **Scenario in which form master, branchName1, branchName2 was created and in branchName1 someone creates function abc, saves it, commits it, at the same time in branchName2 someone creates function abc, saves it and then commits it.**

   - **At the time of merge when things are to be merged with master first, first person merges branchName1 with master and then second person merges branchName2 with master. (things altered in the same file)**

     Merge conflict would arise :

To see the beginning of the merge conflict in your file, search the file for the conflict marker <<<<<<<. When you open the file in your text editor, you'll see the changes from the HEAD or base branch after the line <<<<<<< HEAD. Next, you'll see =======, which divides your changes from the changes in the other branch, followed by >>>>>>> BRANCH-NAME. In this example, one person wrote "open an issue" in the base or HEAD branch and another person wrote "ask your question in IRC" in the compare branch or branch-a.

Think of these new lines as "conflict dividers". The ======= line is the "center" of the conflict. All the content between the center and the <<<<<<< HEAD line is content that exists in the current branch main which the HEAD ref is pointing to. Alternatively all content between the center and >>>>>>> new_branch_to_merge_later is content that is present in our merging branch.

https://docs.github.com/en/pull-requests/collaborating-with-pull-requests/addressing-merge-conflicts/resolving-a-merge-conflict-using-the-command-line

Step 1: In the master the content in master.txt file was:

This is the master file

Step 2: in branch branchName1, in file master.txt, the content was:

This branch was derived from master

This branch is branchName1 in the same file

Step 3: in branch branchName2, in file master.txt, the content was:

This branch was derived from master

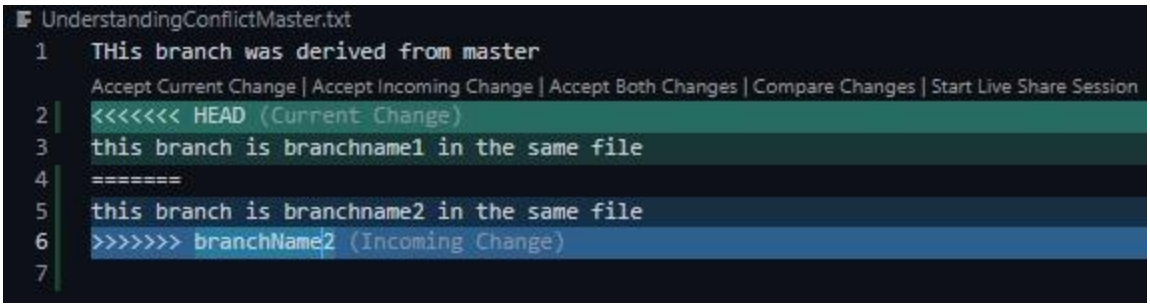
This branch is branchName2 in the same file

Step 4: merging branchName1 to master, the content of branchName1 was added and some lines were deleted.

As we are merging branchName1 to master, branchName1will be given the priority and dominance.

Step 5: When we try to merge branchName2 to the master it gives error (merge conflict)

```
jay@SF-CPU-309 MINGW64 ~/Desktop/Simform documents/JayFolder/GitCourse (master)
$ git merge branchName2
Auto-merging UnderstandingConflictMaster.txt
CONFLICT (content): Merge conflict in UnderstandingConflictMaster.txt
Automatic merge failed; fix conflicts and then commit the result.
```



Now as VS code has an interface to handle merge conflict there are options to choose either to

- **Keep the current change or,**

  will not change anything, leave everything as it is.

- **Accent the incoming change or,**

  accept the incoming merge changes and ignore the previous merge changes.

- **Accept both change**

  Both changes will be kept in the file first merge change first and second merge change second.

  P.S. Conflict will only take place if you commit.

Step 6: Then we need to stage (git add .) the changes and commit it. (After merge conflicts) otherwise the prompts will keep giving an error and the state will be at (master|Merging) and not purely master.

- **What happens when branchName2 merges with branchName3 (vice-versa)**

  If we have created branchName1 and branchName2 from master then we cannot merge branchName1 to branchName2 or vice-versa. We can only merge the branch to the branch that we had created it from.

  Conclude the cases in which we cannot branch if there exists any.

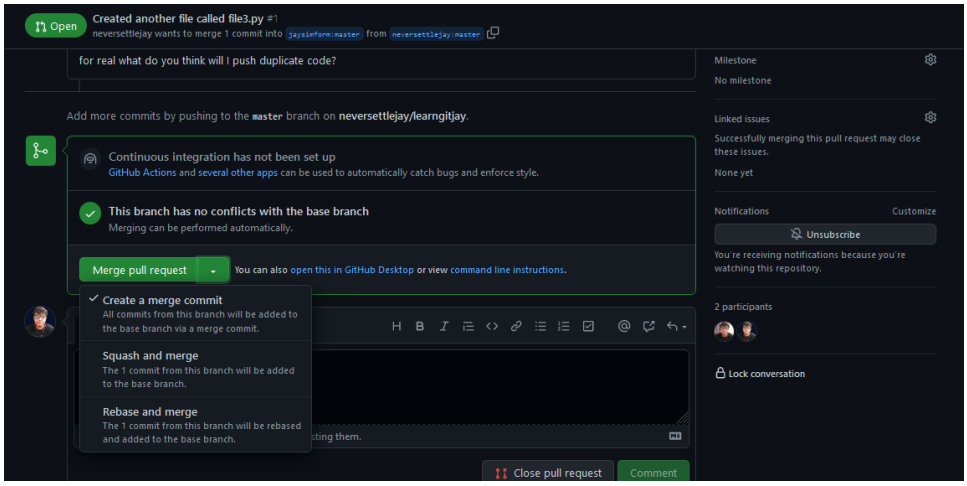
- **When we are working in two different files and try to merge the data in master later**

  If we are working in different files then there does not exist any conflict as long as we don't touch the files and folders that we had originally branched from.

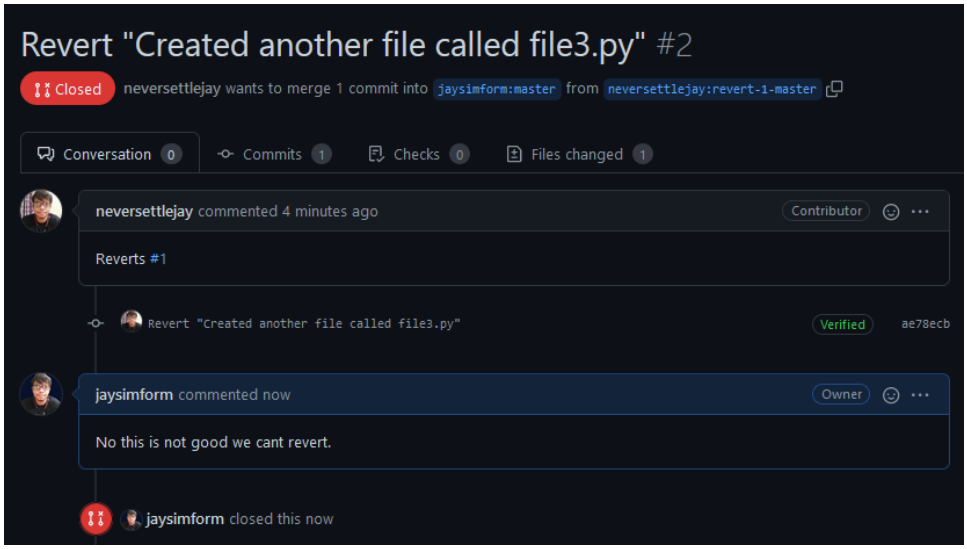
2) When we pull someone's repo and make some changes create alias of their repo and try to push it, it will show this error:

So, if it's a big product and you think your contribution is worth it then you can fork it, upload your committed changes in your forked repository and then send them pull request, by going to the pull request interface in any VCS.

And on the repository owner's side: he will receive a pull request



Or you can deny the PR like:



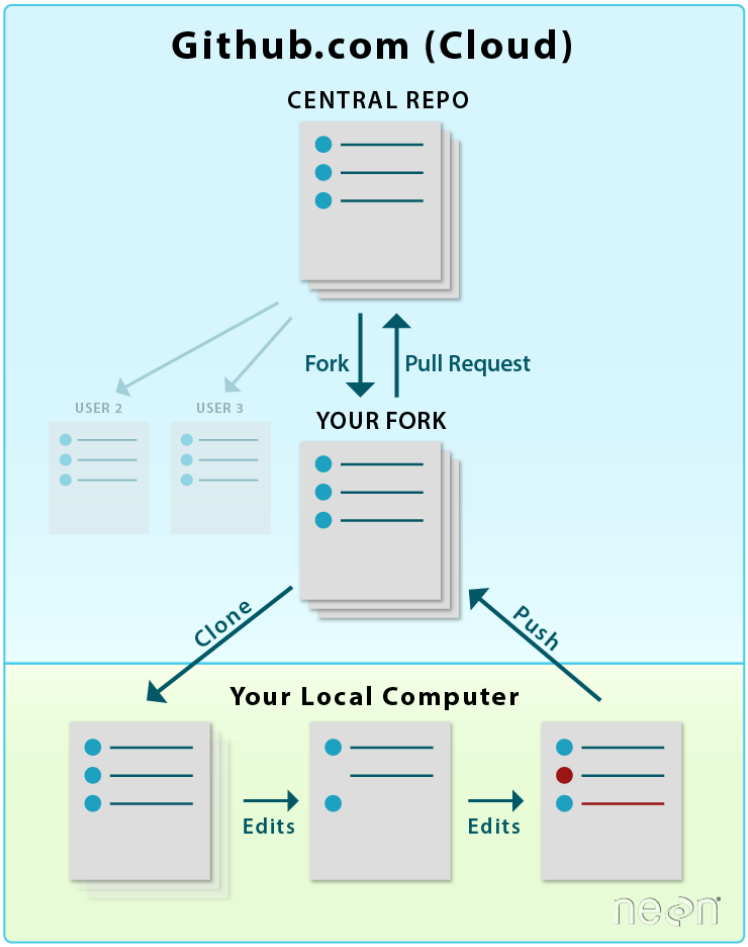
If there exist an update remote repository that others are trying to push in their outdated local repository then it shows this error

so in this scenario, users must pull (fetch + merge) first, make their repo up to date, add their contribution then push or issue PR.

```
jay@SF-CPU-309 MINGW64 ~/Desktop/Simform documents/JayFolder/GitCourse (master)
$ git push origin master
To github.com:jaysimform/learngitjay.git
 ! [rejected]        master -> master (fetch first)
error: failed to push some refs to 'github.com:jaysimform/learngitjay.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

3) When we have made other commits in our local computer and our remote repository was also updated then when we pull it will show error. (merge conflict)

4) How to keep up with the repository you have forked from:



https://www.earthdatascience.org/courses/intro-to-earth-data-science/git-github/github-collaboration/update-github-repositories-with-changes-by-others/

There are a few ways to update or sync your repo with the central repo (e.g. your colleague's repo).

A) You can perform a "Reverse Pull Request" on GitHub. A reverse pull request will follow the same steps as a regular pull request. However, in this case, your fork becomes the **base** and your colleague's repo is the **head**. If you update your fork this way, you will then have to PULL your

changes down to your local clone of the repo (on your computer) where you are working.

B) You can manually set or pull down changes from the central repo to your clone locally. This can be done in the Terminal. When you update your local clone, you will then need to push the changes or commits back up to your fork on GitHub.com.