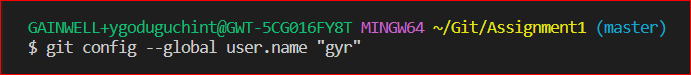
**Git Commands**

**1)git config**

**Usage: git config –global user.name “[name]”**

**Usage: git config –global user.email “[email address]”**

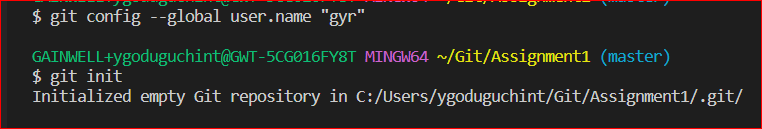
This command sets the author name and email address respectively to be used with your commits.



### ****2)git clone****

**Usage: git clone [url]**

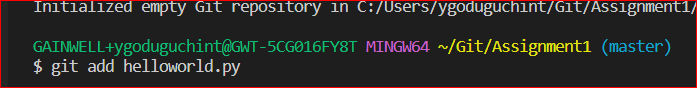
This command is used to obtain a repository from an existing URL.



### 3) ****git add****

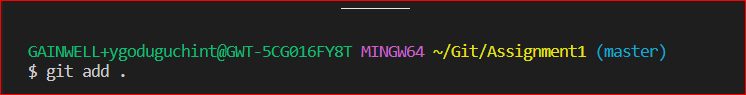
**Usage: git add [file]**

This command adds a file to the staging area.



**Usage: git add \***

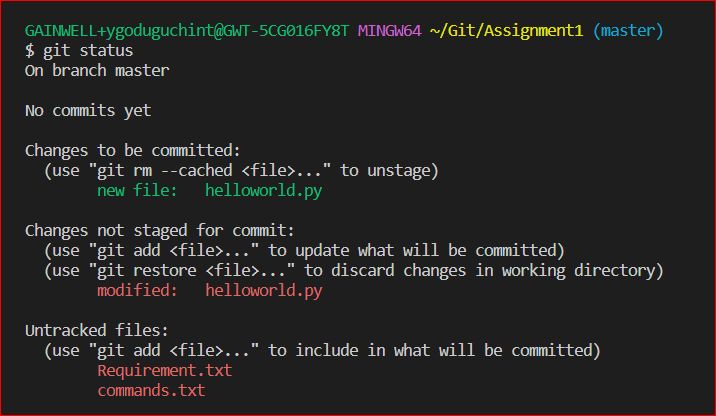
This command adds one or more to the staging area.

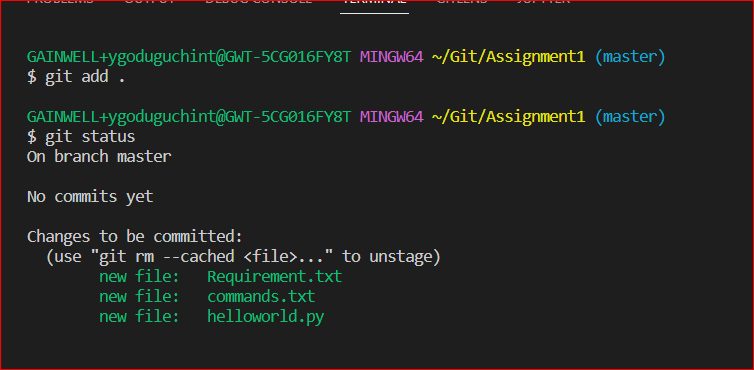


### 4. git status

**Usage: git status**

This command will show the modified status of an existing file and the file addition status of a new file, if any, that has to be committed.





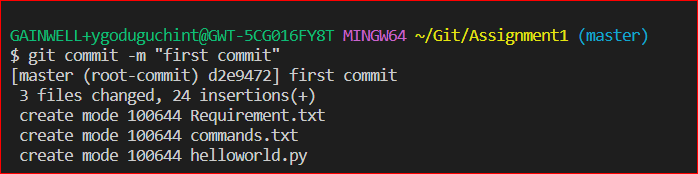
### ****git commit****

**Usage: git commit -m “[ Type in the commit message]”**

This command records or snapshots the file permanently in the version history.

**b)Usage: git commit -a**

This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then.



### 6. git remote

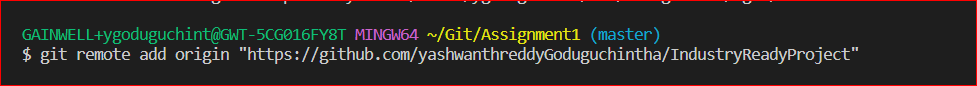
**Usage: git remote add origin “[URL]”**

Once everything is ready on our local system, we can start pushing our code to the remote (central) repository of the project. For that, follow the below steps:

**Step 3:** Click on the Copy icon on the right side of the URL box of the Github repository to copy the link and paste it as shown below:

git remote add origin “URL”

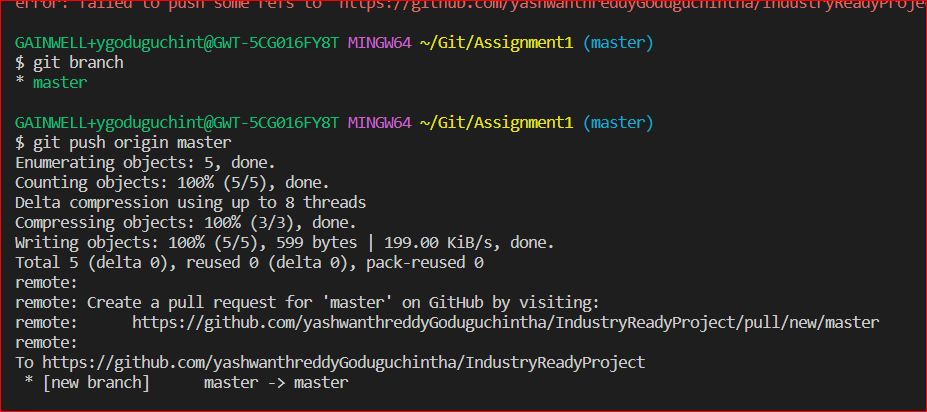
Now, we are ready to operate the remote commands in our repository that we have just created.



### 7) git push

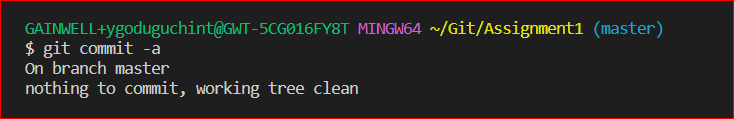
**Usage: git push origin [branch name]**

Suppose, we have made some changes in the file and want to push the changes to our remote repository on a particular branch. By using the command ‘git push,’ the local repository’s files can be synced with the remote repository on Github.



b) **Usage: git commit -a**

This command commits any files you’ve added with the git add command and also commits any files you’ve changed since then



### 8) 8. git branch

**Usage (i): git branch [name-of-the-branch]**

So far, we saw how we can work on Git. Now, imagine, multiple developers working on the same project or repository! To handle the workspace of multiple developers, we can use branches. To create a branch (say, the ‘name-of-the-branch’ is ‘branch1’), we use this command:

### 

### 

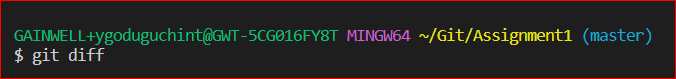
**Usage (ii): git branch -D [name -of-the-branch]**

Similarly, to delete a branch, we use the **git branch -D**command:

### 

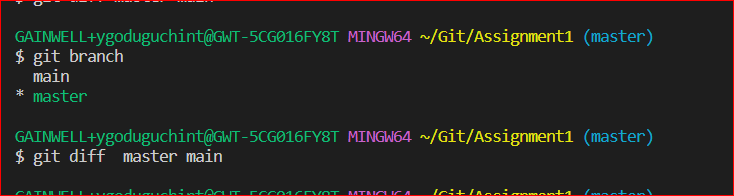
**9)Usage: git diff**

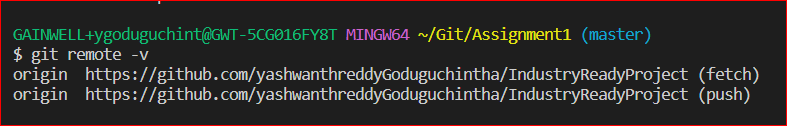
This command shows the file differences which are not yet staged.



**Usage: git diff –staged**

This command shows the differences between the files in the staging area and the latest version present.

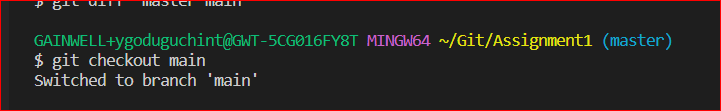




### 10. git checkout

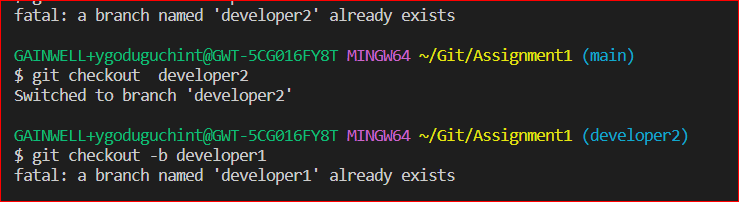
**Usage (i): git checkout [name-of-the-new-branch]**

We use this command to navigate to an existing branch, add new files, and commit the files:



**Usage (ii): git checkout -b [name-of-the-new-branch]**

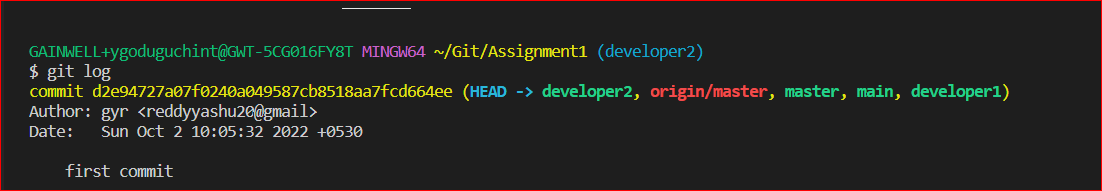
We use this command to create a branch and navigate to that particular branch (say, the ‘name-of-the-new-branch’ is ‘branch2’) at the same time:



### 11) git log

**Usage (i): git log**

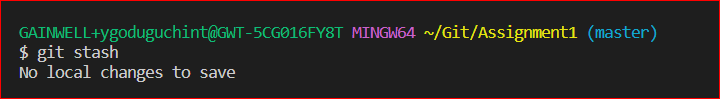
This command is used when we want to check the log for every commit in detail in our repository.



### 12) git stash

**Usage (i): git stash**

This command can be used when we want to save our work without staging or committing the code to our Git repository and want to switch between branches.

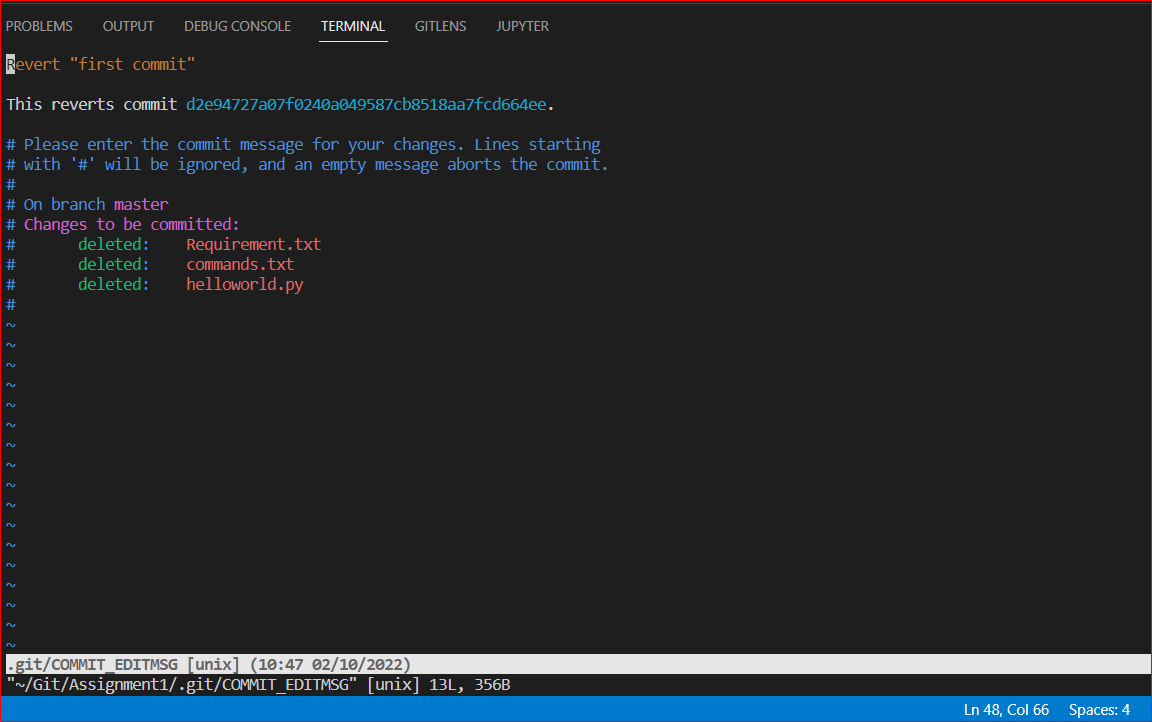


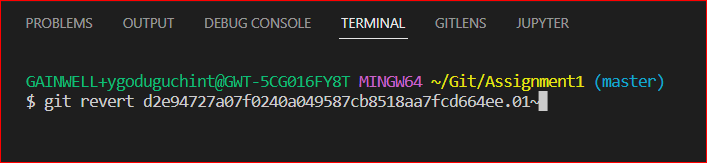
### 12. git revert

**Usage: git revert [commit id]**

The git revert command can be considered as an ‘undo’ command. However, it does not work as the traditional ‘undo’ operation. It figures out how to invert the changes introduced by the commit and appends a new commit with the resulting inverse content.

Git revert head

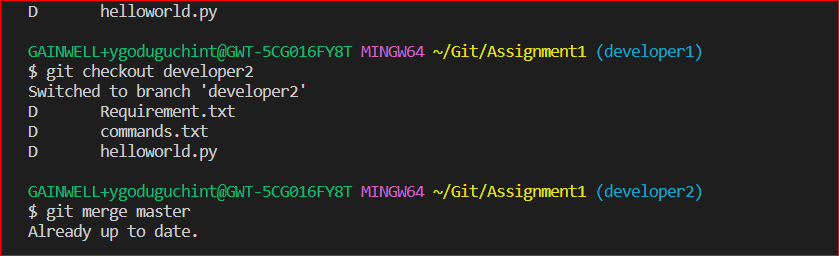




### 14. git

**Usage: git merge [another-file-name]**

This command 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 git merge command takes two commit pointers, usually the branch tips, and finds a common base commit between them. Once it finds a common base commit, it will create a commit sequence.

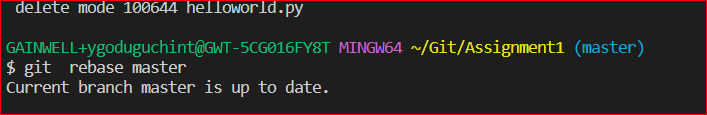


### 15. git rebase

**Usage: git rebase [base]**

Rebase is the process of moving and combining a sequence of commits to a new base commit. Rebasing is changing the base of our branch from one commit to another, making it appear as if we’ve created our branch from a different commit. Internally, Git accomplishes this by creating new commits and applying them to the specified base. It’s very important to understand that even though the branch looks the same, it is composed of entirely new commits.

The git rebase command performs an automatic git checkout <branch> before doing anything else. Otherwise, it remains on the current branch.



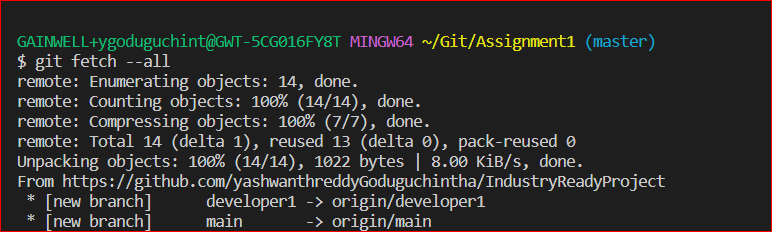
Consider a situation where we have branched off from the master and have created a feature branch, but the master branch is still having more commits. We want to get the updated version of the master branch in our feature branch, keeping our branch’s history clean, so that it appears as if we are working on the latest version of the master branch.

**Note**: We don’t rebase public history. We should never rebase commits once they are pushed to a public repository. Why because the rebase would replace the old commits with the new ones, and it would appear that a part of our project history got abruptly vanished.

### 16. git fetch

**Usage: git fetch**

When we use the command git fetch, Git gathers any commit from the target branch that does not exist in our current branch and stores it in our local repository. However, **it does not merge it with our current branch**.

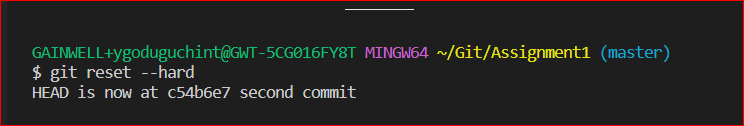


This is particularly useful when we need to keep our repository up to date but are working on something that might break if we updated our files. To integrate the commits into our master branch, we use merge. It fetches all of the branches from the repository. It also downloads all the required commits and files from another repository.

### 17) git reset

**Usage: git reset –hard [SOME-COMMIT]**

We use this command to **return** the entire working tree to the last committed state.



This will discard commits in a private branch or throw away the uncommitted changes!

Here, we have executed a ‘hard reset’ using the **–hard** option. Git displays the output indicating that the HEAD is pointing to the latest commit. Now, when we check the state of the repo with git status, Git will indicate that there are no pending changes (if any prior addition of a new file or modification of an existing file is done before using the ‘git reset –hard’ command). Our modifications to an existing file, if not committed, and the addition of a new file, if not staged, will be destroyed. It is critical to take note that this data loss cannot be undone.

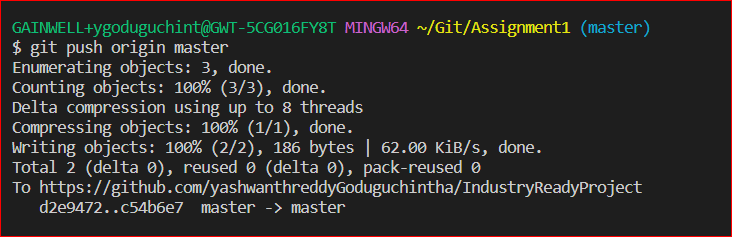
If we do**git reset –hard [SOME-COMMIT]**, then Git will**:**

* Make our current branch (typically master) back to point <SOME-COMMIT>
* Make the files in our working tree and the index (“staging area”) the same as the versions committed at <SOME-COMMIT>

### 18) git pull

**Usage: git pull origin master**

The git pull command first runs ‘git fetch’ which downloads the content from the specified remote repository and then immediately updates the local repo to match the content.



Task 2:-

