**Git and GitHub**

**Git/Github Tutorial 4: Basic Commands: add, commit, push**

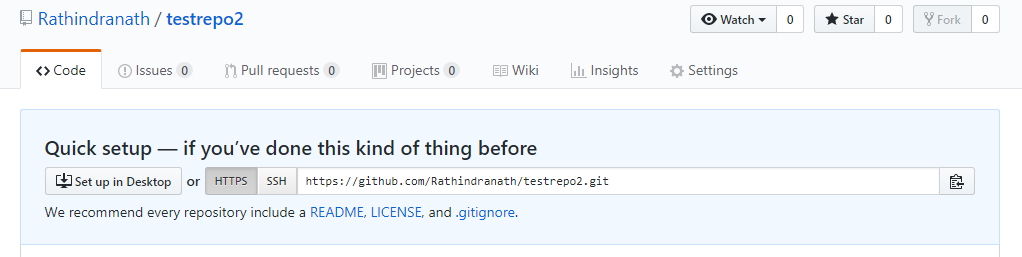
<https://www.youtube.com/watch?v=eL_0Ok_Gkas&index=4&list=PLeo1K3hjS3usJuxZZUBdjAcilgfQHkRzW>

Items to cover,

1. Create new github repo
2. Push code to this repo using git
3. Go over these git commands,
4. git add
5. git commit
6. git push
7. git difftool

**Create new github repo:**

First we need to create our account in GitHub (github.com) and then login to it using that account and then create new github repository. We have created a new repository testrepo2 in github for this video series.

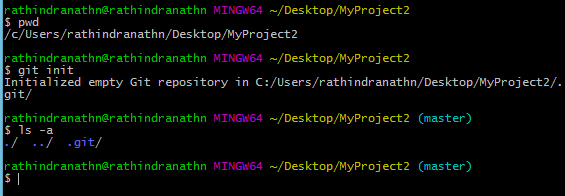


By default, a repository in GitHub is created as public. If we want to create a repository as private, we need to choose the relevant option and also we need to pay the price using credit card. Our repository is public, which is the default. We can give description of the repository and also we can create a readme file during creation of a repository in GitHub, but these are not mandatory. We can create a readme file for the repository later also.

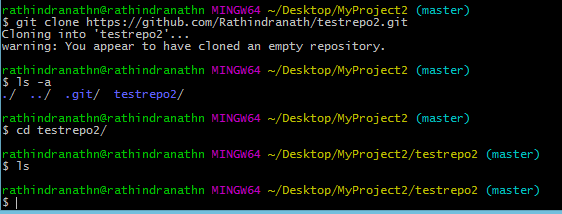
Our repository is in GitHub, which is hosted in the cloud. How do we sync this remote repository in our local system? For that we need to install git in our local system and take help of this.

**Clone remote repository from GitHub to our local system:**

We have a folder called MyProject2 in the desktop in our local system. We want to make it our project folder. Therefore all the project related files will reside here. We also want to track these files with the help of git.



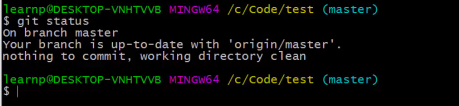
Now, we shall clone our repository testrepo2, which is there in GitHub to our project folder in our local system. Cloning a repository means copying it from GitHub to our local computer. For that we need to go GitHub and then copy the URL of the repository. In the above, the first figure shows the URL of our repository myProject2. We need to click on the icon, which is there at the right end of the URL. It is the icon for “Copy to clipboard” – clicking on it copies the URL. After that we need to paste it in the “git clone….” Command at git bash prompt in our local system,



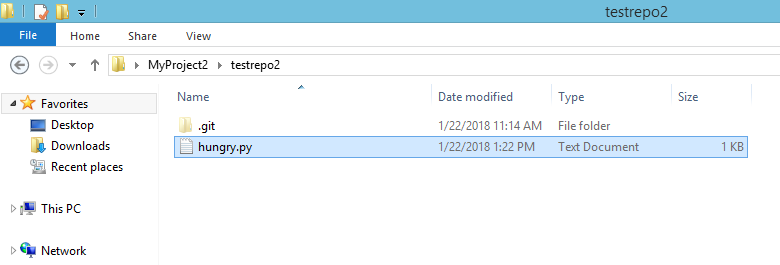
Therefore, we have successfully cloned the remote repository testrepo2 in our project folder MyProject2, which is there in our local system. But there is nothing in testrepo2, since we have not created anything there till now.

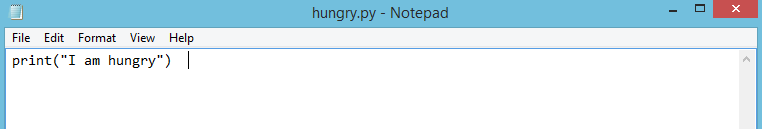
**Git operation:**

The command we frequently use in git is “git status” to check the current status of git. The status of git in present case has been shown in the following screen shot. It says that we are currently in master branch, which is the default branch in git. Also the master branch in up to date and there is nothing to commit.

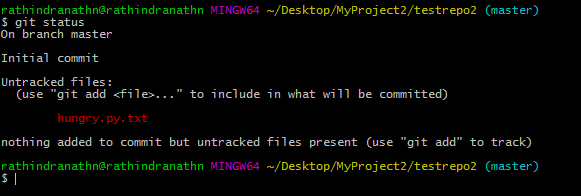


Now we shall create a small file of python code in our project folder MyProject2. The name of the file is hungry.py and it has one line of code.

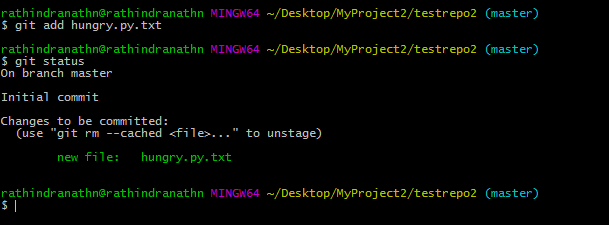




Let’s check the status of git,

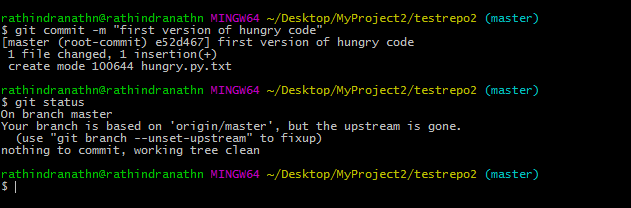


It says that there is one untracked file, called hungry.py.txt. It is in project directory. Now, we need to add this file to staging are. A file is kept in staging area before it is committed. We are advised to add the file to the staging are by executing “git add….” Command. Let’s executing this command,



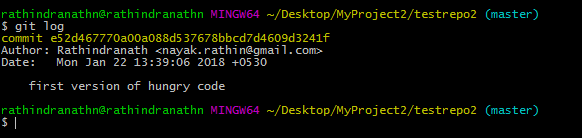
In the above hungry.py.txt file has been added to staging area. Later staging area will be used to commit files when we run “git commit……” command. The color of the file has been turned to green from red.

Now we shall commit the change to the local version control system,



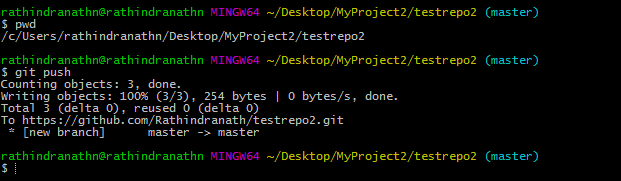
git commit adds our changes to local version control database. After that we executed “git status” command, which tells there is nothing left to commit. We are yet to push the change to remote server.

Now, we shall execute the “git log” command, which tells all the commits in git – this command tells the commit history in git.

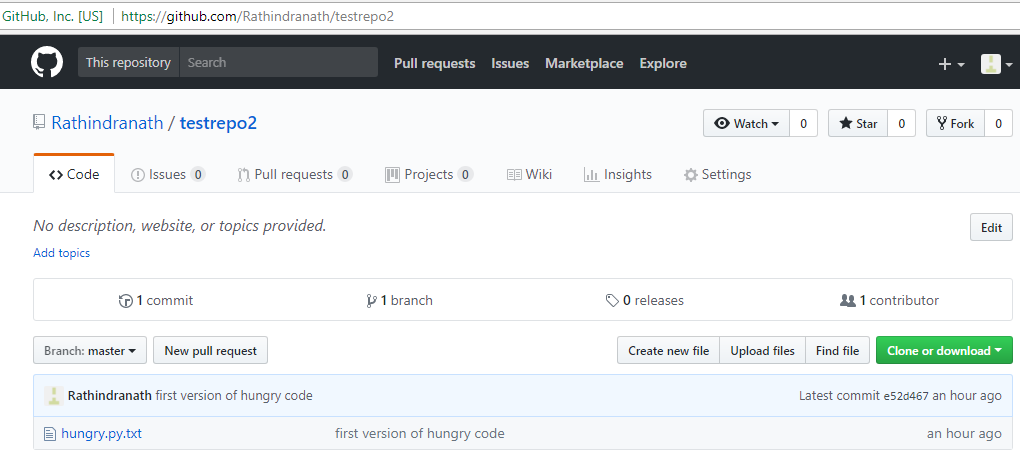


Till now we have committed only once. With each commit, one commit id is generated. It also tells the author of the commit and also date and time of commit.

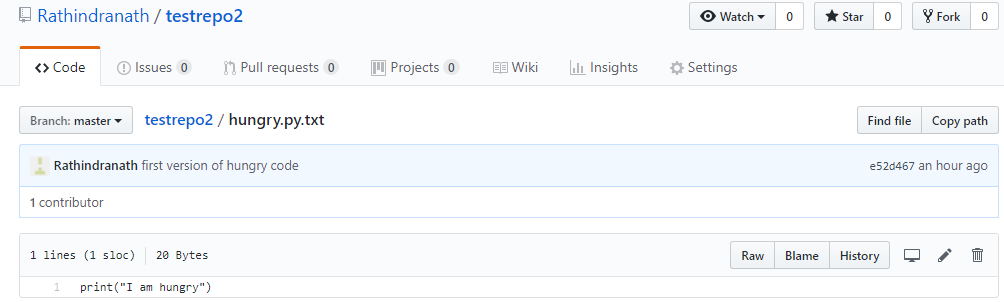
Now we are ready to push the changes to GitHub. For that we need to execute “git push….” Command – it pushes committed changes to remote repository.



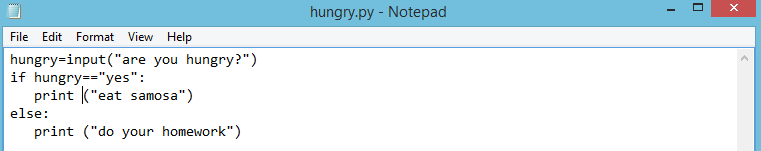
Let’s look at the remote repo in GitHub,



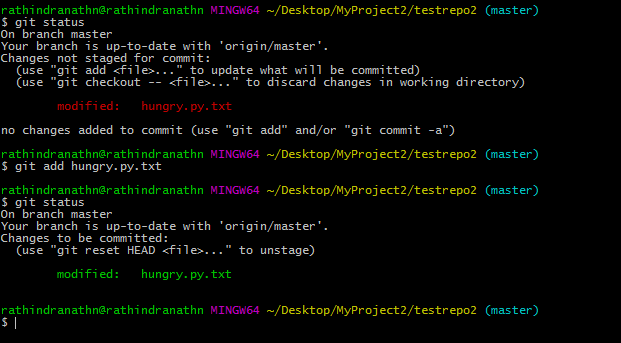
Therefore, the file hungry.py.txt has appeared in testrepo2 in GitHub. It has one line of code as shown below,



Now, we want to make few changes to hungry.py.txt file in our local system.

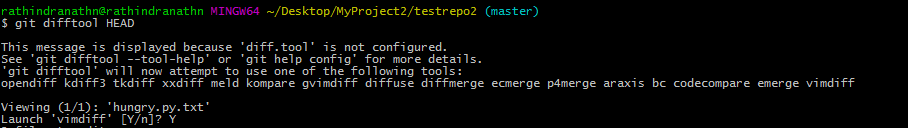


Let’s move the file to the staging area,

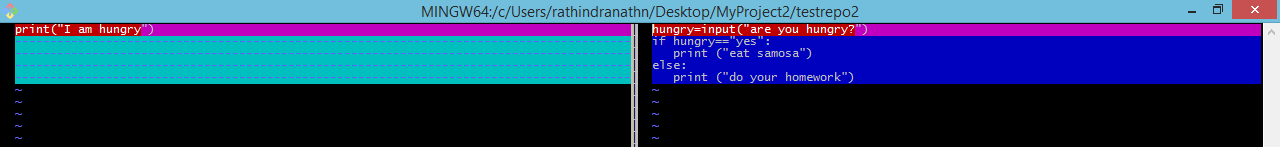


The file hungry.py.txt has moved from working directory to staging area and its color has changed accordingly – from red to green.

Now we shall execute the “git difftool..” command. It’s a command, which allows us to see the difference between whatever code we had previously and whatever code we have right now. For that we need to execute following command “git difftool HEAD” in git bash. Later we shall discuss about the concept of HEAD.

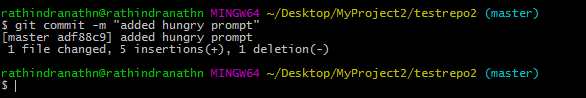


Following window will open. The previous version of the file is in the left and the current version of the file is in the right. We can setup other tool like “meld” as difftool. But by default “vimdiff” is setup. Following output is in “vimdiff”. To quit it, we need to type “:q” multiple times.

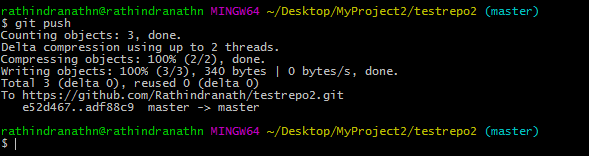


Previously we had one print statement in hungry.py.txt and now we have multiple lines of code with two print statements.

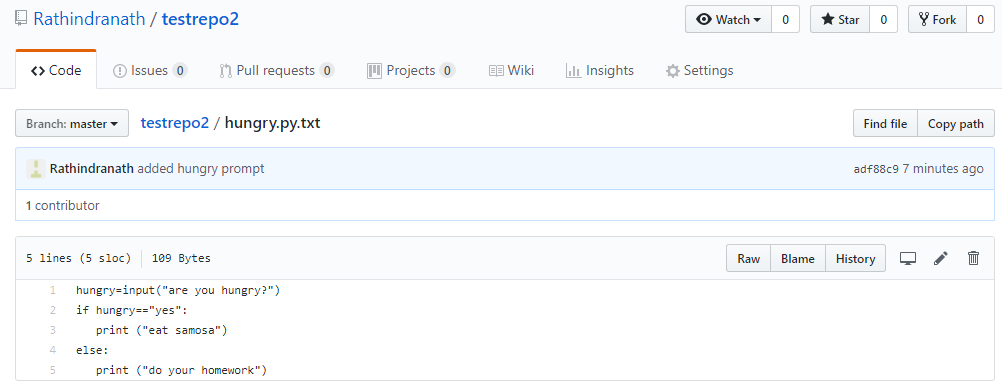
Now, we shall commit the file to our local repository. During commit we must provide a meaningful message and for that we have to use “–m” option.



Now, we shall push the change to remote repository in GitHub,

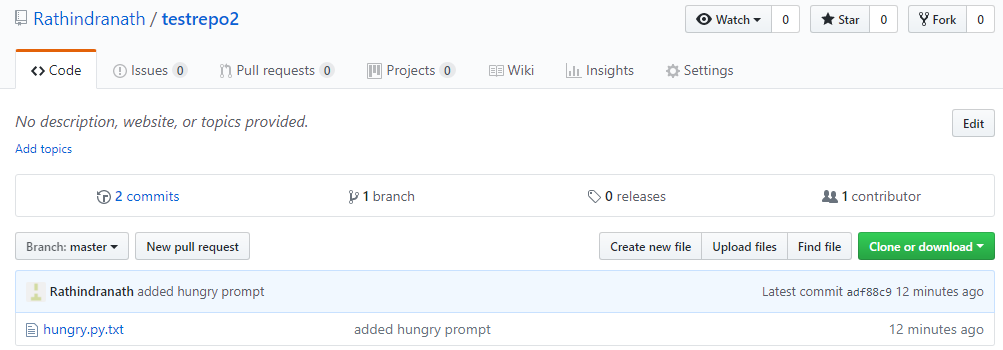


After pushing all the changes will go to off stream and therefore it will be reflected in the remote repository in GitHub.

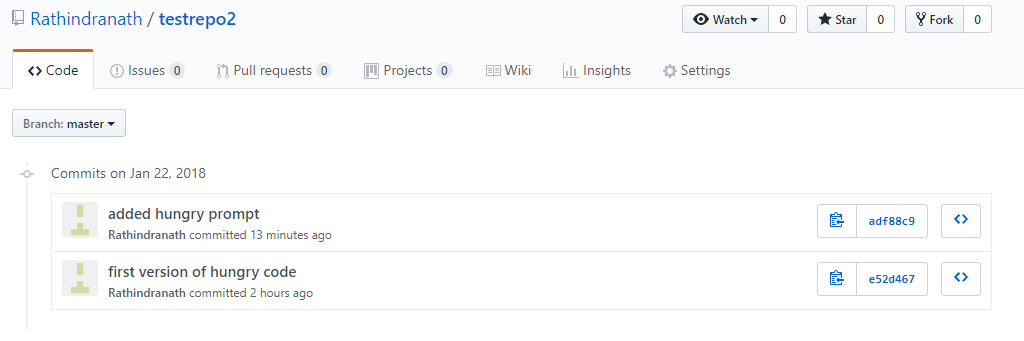


All the newly added lines and the deleted lines are reflected in the off stream, i.e. in “hungry.py.txt” file in “testrepo2” repository in GitHub.

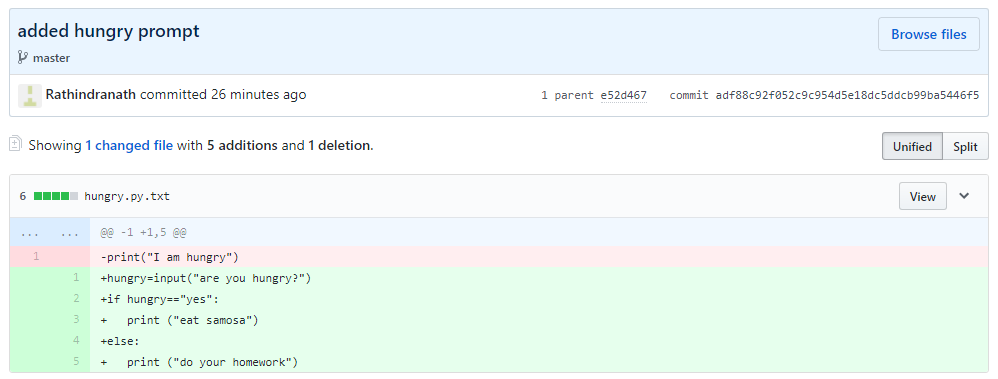
We can see in GitHub that there are two commits till now in the “testrepo2” repository.



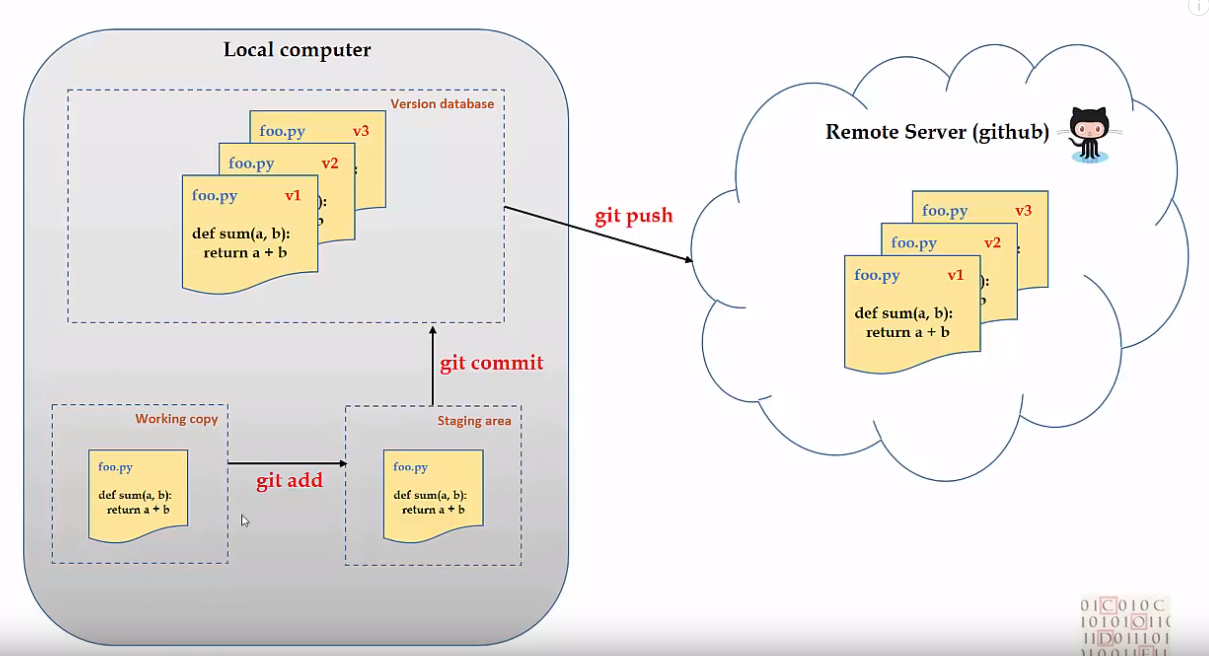
If we click on commits, all the commits till now are displayed in details,



If we click on a particular commit, a full description of change applied to file(s) in the repository is displayed. It shows which lines are added, which lines are deleted and which lines are modified in a particular file, which has been committed. For example, we have clicked on the latest commit “added hungry prompt” in testrepo2 repository and the following screen appears,



Till now, we have discussed about three basic git commands: add, commit and push. Let’s illustrate a picture, shown in the following, to understand how these command work.



Locally in our computer we have a working copy of the project. Whenever we make change in any file of the project and then execute “git add …..” command, the file moves to an area called staging area. After that when we execute “git commit …” command, whatever is there in staging area is committed. When we do commit operation all the changes made are pushed to the version database. The version database is present in the local system along with the staging area and the working copy (local copy) of the project. If we change a single file multiple times and then commit each time, all the different versions of the file will be saved in the version database. Therefore all the changes keep on accumulated in the version database in the local computer.

When we do the “git push …” operation, all the changes move to the remote repository in the GitHub and are reflected there. Now, the files of a project reside in the cloud (GitHub). From anywhere in the world we can clone or fetch them and then work on them. This alleviate the issue of losing the code due to system crash and also make it possible to contribute to the project in a distributed fashion.

**Git/Github Tutorial 5: Undoing/Reverting/Resetting code changes**

<https://www.youtube.com/watch?v=3dk3s4LK-Wg&index=5&list=PLeo1K3hjS3usJuxZZUBdjAcilgfQHkRzW>

Here, we shall see how to,

1. Undo uncommitted changes

git checkout --

1. Undo committed changes

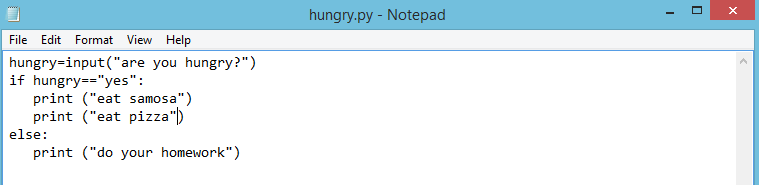
git revert

1. Resetting changes

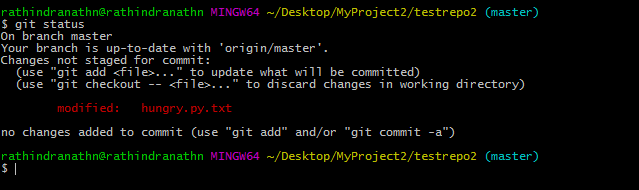
git reset

**Undo uncommitted changes:**

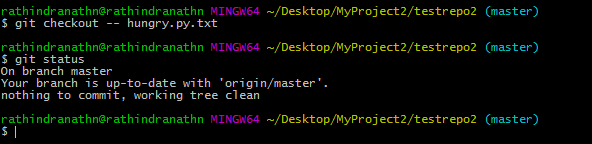
Now, we shall make changes in the file hungry.py.txt.



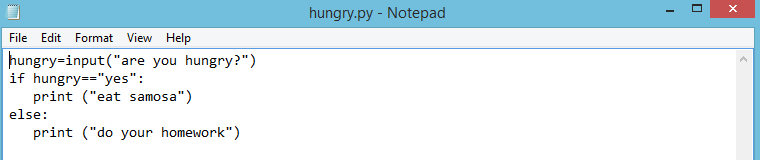
We have added a new line “print (“eat pizza”)”. Let’s check the status of git,



Therefore, the changed file hungry.py.txt is yet to be in staging area. But we don’t want to continue with this change. Following command will discard the change,



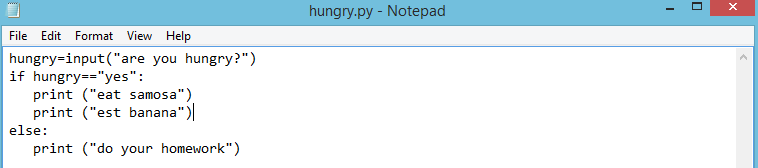
In fact in previous screen shot, the command “git status” shows the same command to discard changes in working directory. After running the command “git checkout -- ….” Our current branch “master” is up to date – there is nothing to commit. Even our file in the local repository in our system has lost the line,



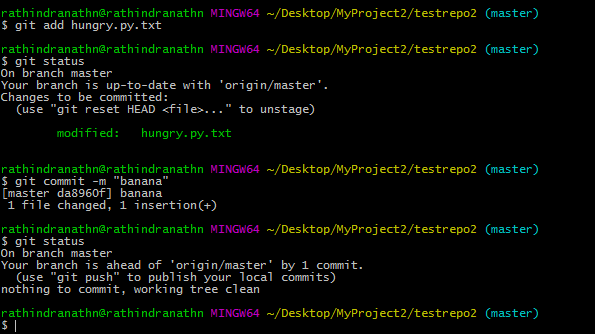
If we want to undo the change made to multiple files we need to either mention all the files separated by space in the command line of “git checkout -- …” command or we can denote all the files using a dot (.). Therefore the command would look like the following: “git checkout -- .”).

**Undo committed changes:**

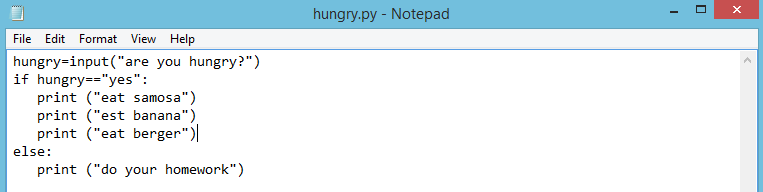
Let’s make change to the file hungry.py.txt and then commit the change to local repository. We have added one line and it looks like the following,

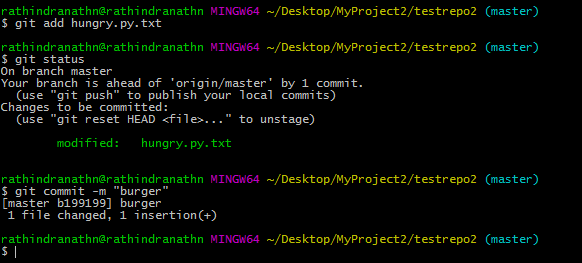


Let’s commit the change in git,



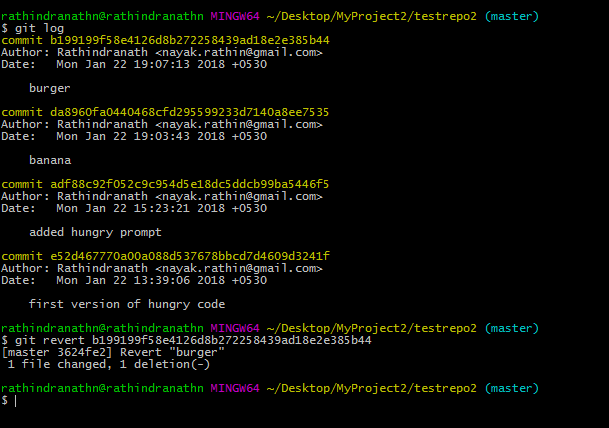
Let’s make another change and then commit,



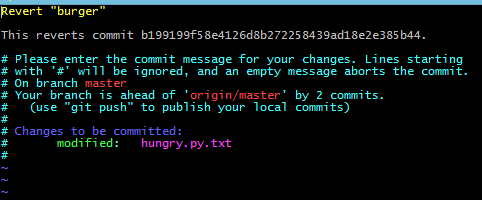


We can commit in one shot from work directory to the local repository by executing the following command: **“git commit –am “<comment>”.** In that case there is no need to execute “git add …” command, since placing the file to staging area is skipped.

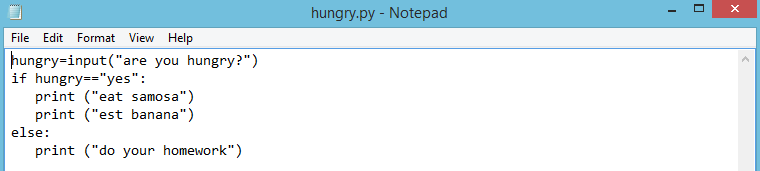
Now, we want to undo the “burger” change. For that we need to execute “git revert ….” Command. First we need to get the commit id for “burger” commit and then pass that commit id to “git revert …” command.



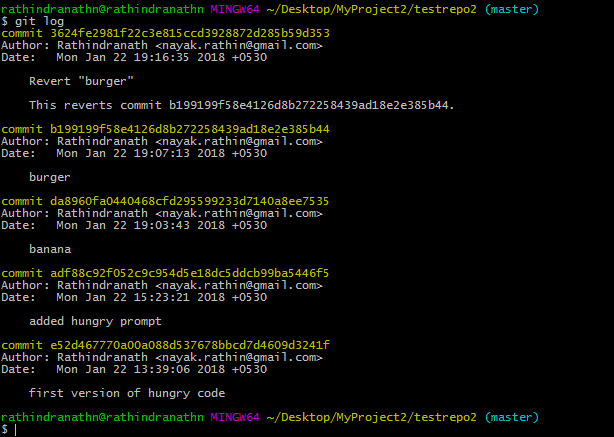
When we execute the above revert command below message is displayed. We need to type “:q” to quit it. Then only the command finishes the execution.



In the file hungry.py.txt the intended line is gone,

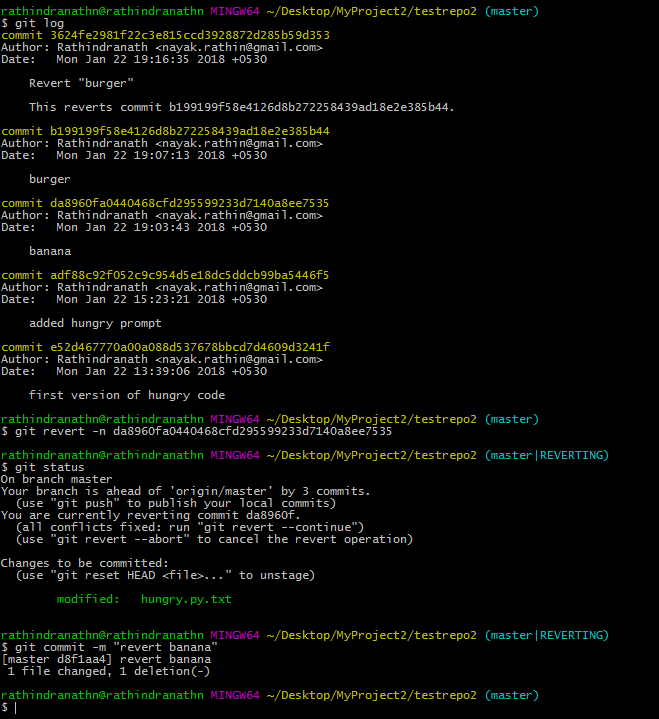


The line “print (“eat burger”) is deleted and the file is reverted back to its previous state. “git log” command also show the change in its output,

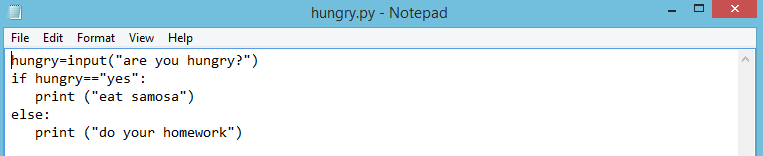


Suppose, we don’t want to commit the change at the same time when we revert back any previous commit. We can do so by using “-n” option in “git revert …” command”. In that case we need explicitly commit later. So it is a two steps process of reverting previous commit back. But it gives us a chance to review the action before it is committed.

Suppose we want to remove the line “print ("est banana")” from hungry.py.txt file. We can do so in the following way by not reverting and committing at the same time. In the following, we have first copied the commit id of the “banana” commit from “git log” command output. Then we executed the command “git revert –n <commit id>”. After that we committed it.

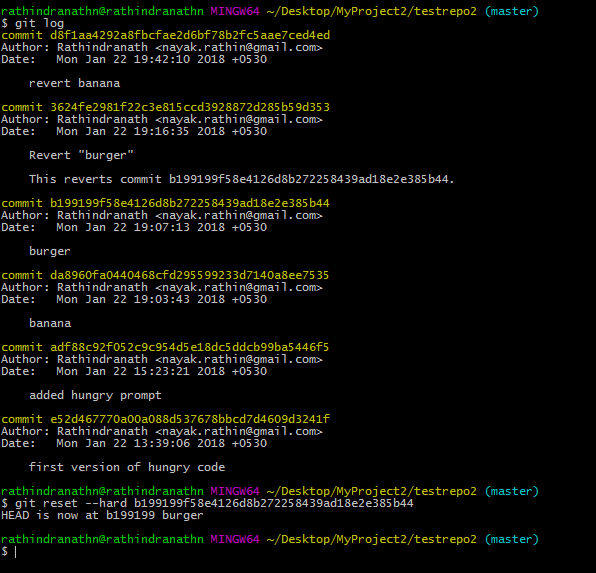


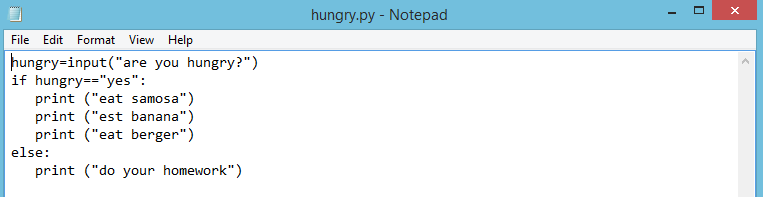
The file has lost the concerned line as the following screen shot shows that,



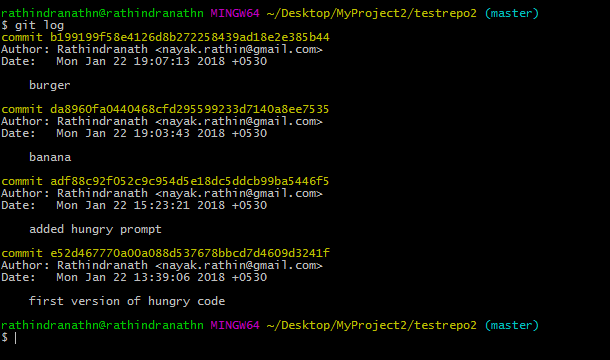
**Resetting changes:**

We have the whole git log history. It illustrates the history of the code – how the code evolved. Now we want to go back to the status where we had “banana” and “burger”. The “git reset …” command will help us to achieve that. We have to do in the following way,





In the above we have first fetched the commit id for “burger” commit. Then we executed the command “git reset –hard <concerned commit id>”. The file burger.py.txt has got back two lines regarding banana and burger. Following is the latest “git log” output.



‘git reset” thus helps us go to a certain time stamp of code history.

**Git/Github Tutorial 6: Branches (Create, Merge, Delete a branch)**

<https://www.youtube.com/watch?v=sgzkY5vFKQQ>

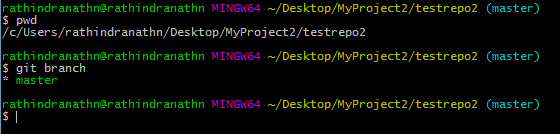
Here we shall discuss about branches in git. Following are the topics, which we shall discuss here,

1. Creating a new branch
2. Merging a branch
3. Deleting a branch

Let’s discuss why branch is needed. In the above we have developed code in testrepo2 folder. There is a file hungry.py.txt, where we wrote few lines of codes. Now, we want to make few changes to it for implementing new ideas. What we can do is making a copy of the original file and then we can do modification of the copy file. In that case there is no need to touch the original file and thus the original file remains intact. This is not a very efficient way to work on existing code base if the size of code is quite big. Git comes with an efficient technique to do the same and it is called branching. Branching in git allows us to create two branches of same code and then do parallel development on both the branches. The original code resides in a branch. Then a new branch is created out of it. So the new branch contains the same copy of code. Developer now works on the code, kept in new branch. He makes modification as per requirement. Then the changes are tested and validated. After everything is fine, the new branch is merged with the original branch. Through merging the changes are merged with the original code and thus a modified or enhanced version of the code results in.

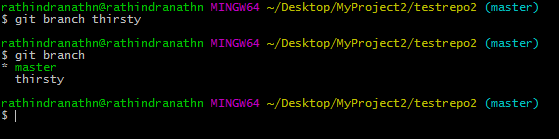
When we initialize a folder or directory by git, a new branch is created, which is called master and we automatically land in that branch. Master is the default branch in git.

We are currently in our project directory testrepo2, which is a subdirectory of MyProject2. Let’s try to get the branch information,



The command “git branch” lists all the existing branches and it also tells what the current branch is. Above command and its output tells that there is one branch called master and it is our current branch. Our currently active branch is shown in green color and it is preceded by a star (\*).

Now, we shall create a new branch “thirsty” out of the main branch,

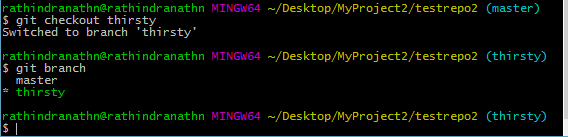


The “git branch <branch name>” creates a new branch from the current branch. After that we checked the list of all the existing branches. We can see from the above output that we now have two branches: master and thirsty. But our current branch is master.

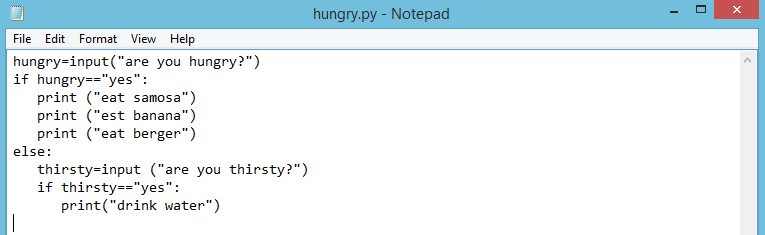
To make another branch as currently active branch we need to execute following command,

git checkout <branch name>

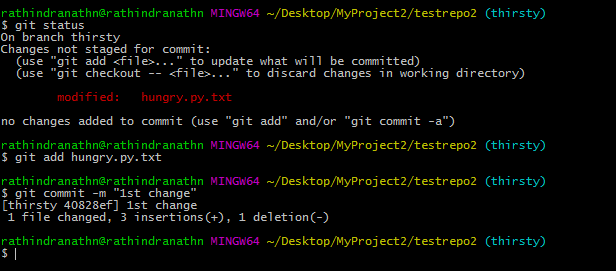
Now we shall switch to the branch “thirsty” and it will be our active branch,



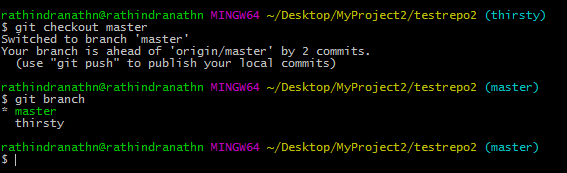
Now we shall make change to the file hungry.py.txt,

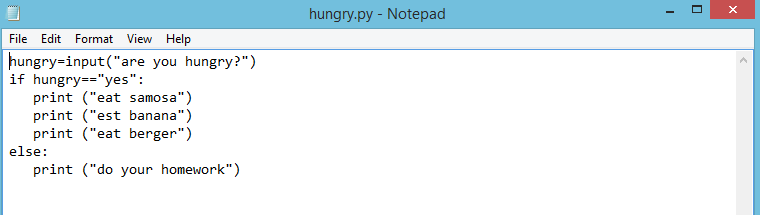


We have changed the else part of the code in the file hungry.py.txt. Let’s commit the change using git,



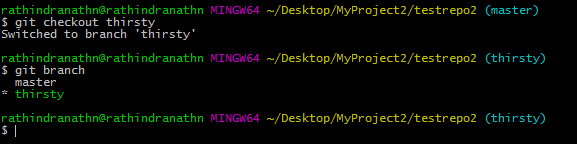
But this commit we have done in the “thirsty” branch. Let’s checkout master branch and then check the content of the file,

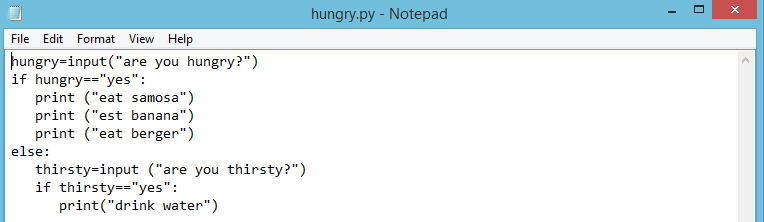




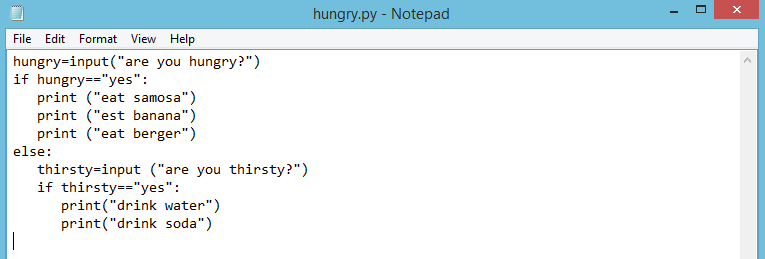
Above output shows that the changes in the file are not reflected in the main branch.

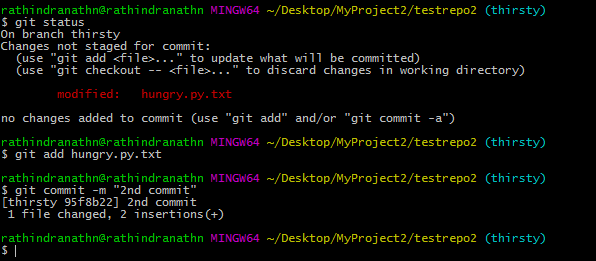
If we switch to “thirsty” branch we can see the changes in the file again,



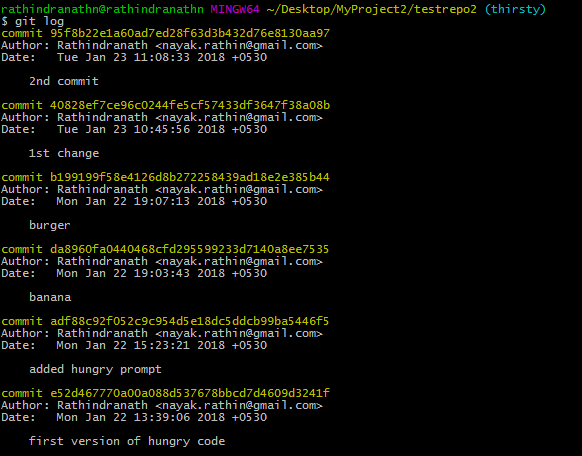


Let’s make further change in the hungry.py.txt file and then commit the change,

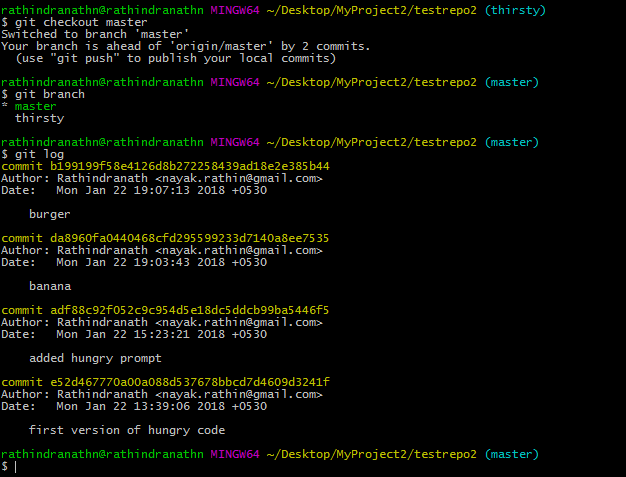




Let’s check the commit log in each branch. Let’s start with “thirsty” branch where we are currently in.

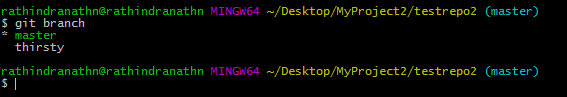


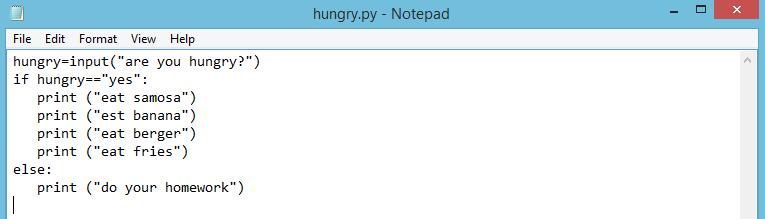
Let’s checkout master branch and then check the commit log there,



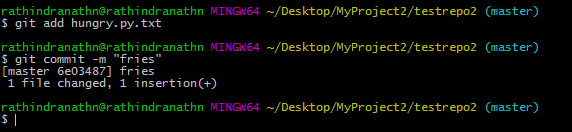
When we are in the master branch “git log” command does not show the commits, done in the “thirsty” branch. But, when we are in “thirsty” branch all the commits, whether they were done in the master branch or in the “thirsty” branch, are displayed.

We are currently in master branch and we want to edit the hungry.py file to incorporate few changes.

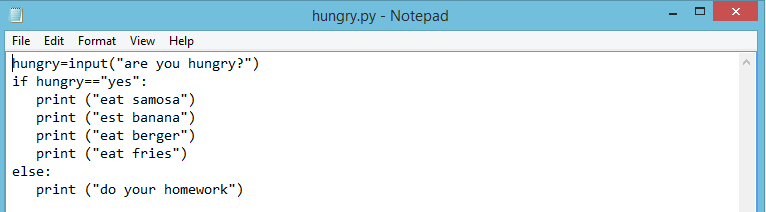




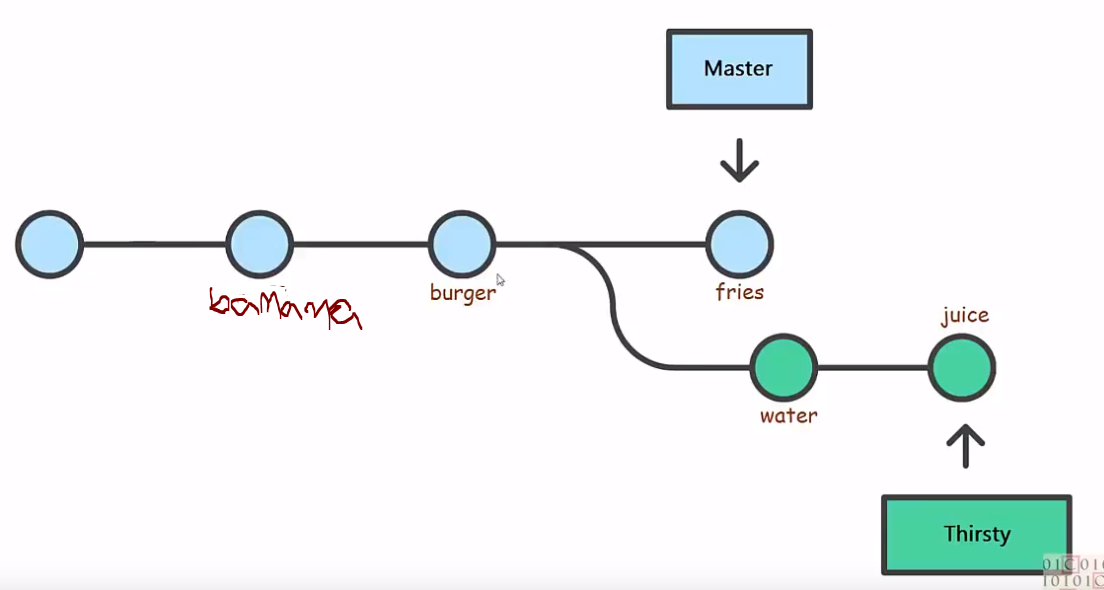
Let’s commit the change in git,



This change is again not reflected in “thirsty” branch.



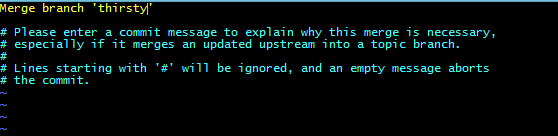
Following is the pictorial view of what we have done so far with hungry.py.txt file. All the circles are commit points – each circle is an individual commit. Initially we made changes in the mater branch and commit points show this. Then we created a branch “thirsty” out of it and also made changes to the file and also committee those changes. We added the “banana” line when we were in master branch, then we added “burger” line while staying in the same master branch. Then we created a new branch called “thirsty” and made few changes in the file and then committed also. All the green color commits are in “thirsty” branch. Commits done while we are in master branch are denoted in blue color. As per commit points in the figure, lines regarding banana, burger and fries are there in the master branch – they are not reflected in “thirsty” branch. Similarly, lines regarding water and juice are in thirsty branch and they are not reflected in the master.



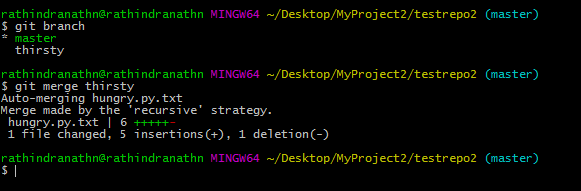
Now we shall merger the “thirsty” branch to master. For that we need to go to the base branch, to which we want to merge. Here base branch is master and currently we are in that branch. Next, we need to execute following command,

git merge <branch name>

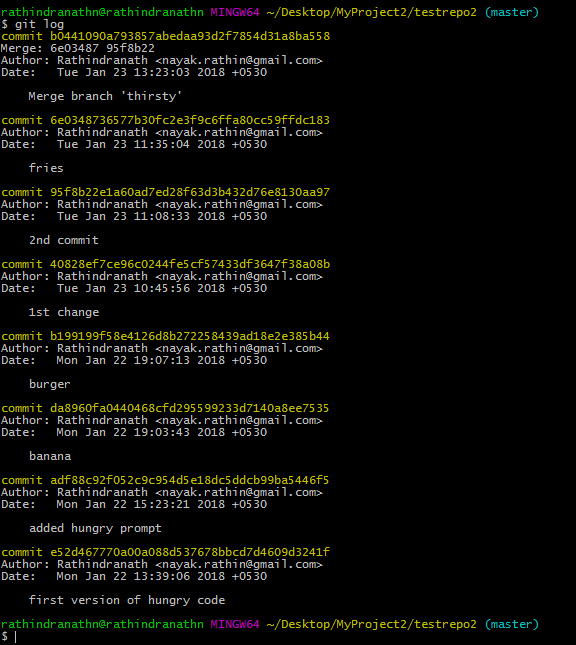
In the following, a new window appears with a line by default. Do save and quit (:wq).



Following the screen shot of git bash,

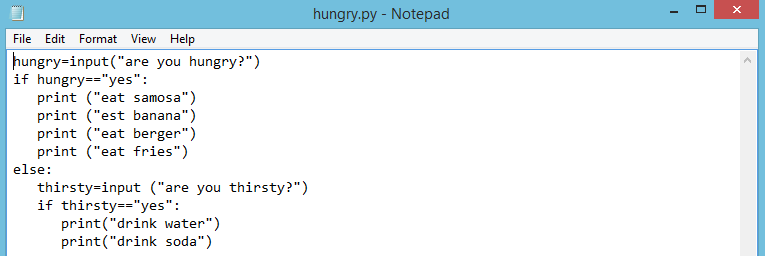


“git log” command shows this merge operation in its output,

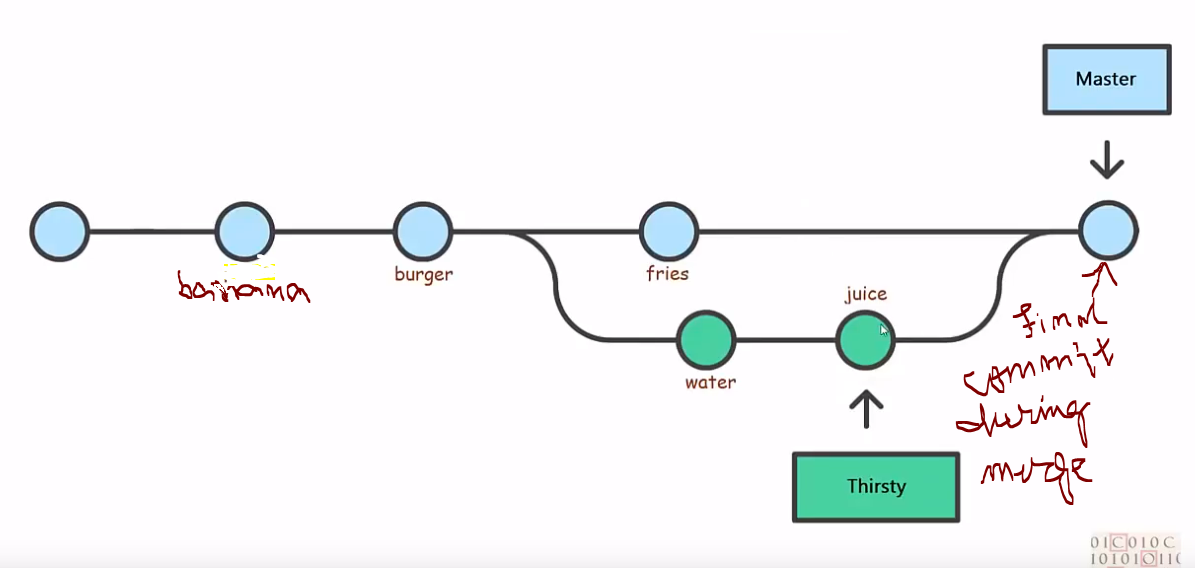


Default merge comment “Merge branch ‘thirsty’” is also showing in the output.

Let’s look at the content of hungry.py.txt file.

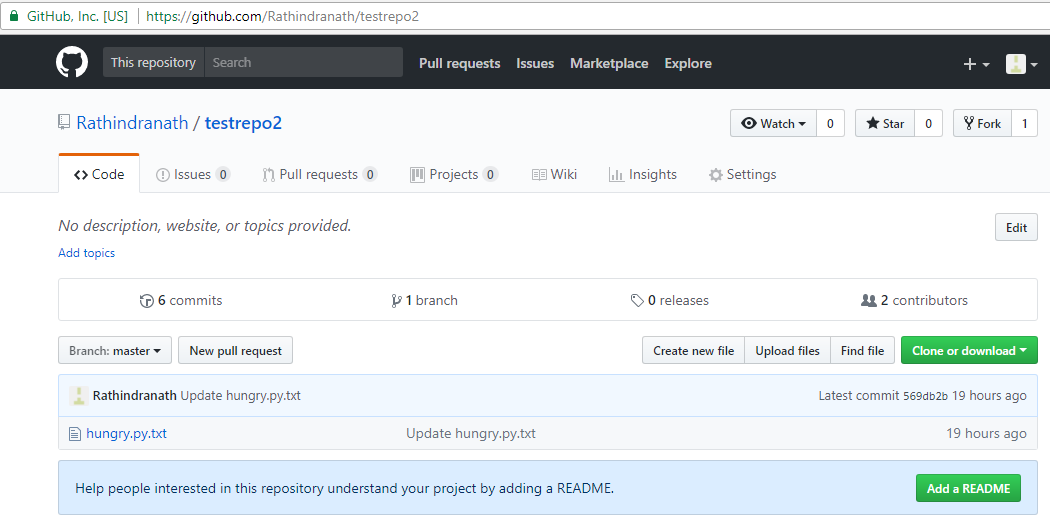


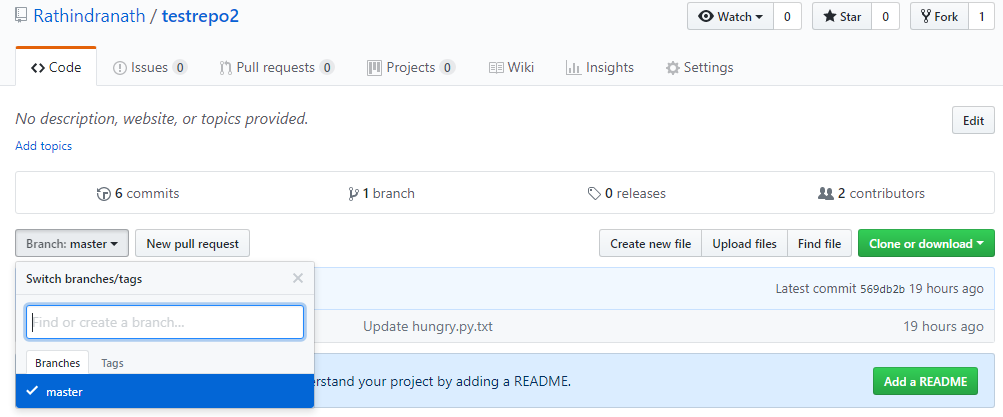
All the changes made to the file in “thirsty” branch are merged to this file and they are reflected in master branch. Following figure depicts this,



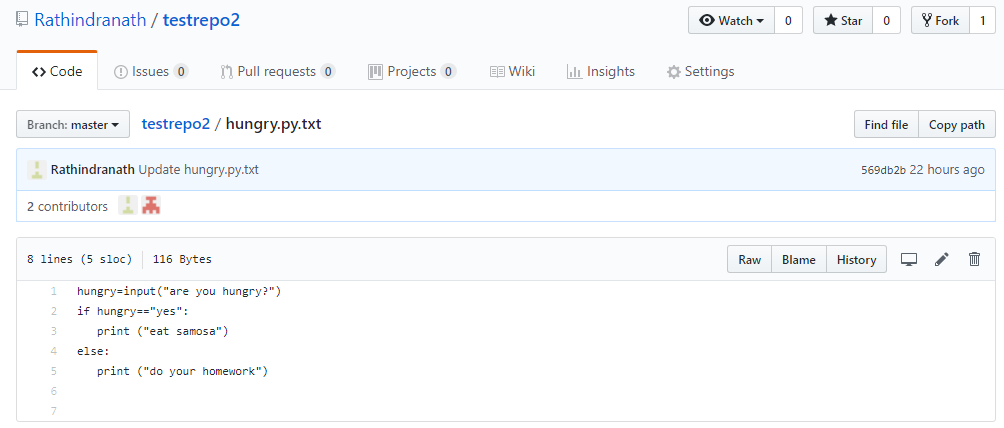
During merge the final commit takes place, which results in the final content in the hungry.py.txt.

Following is what is there in GitHub as far as testrepo2 repository is concerned. It has only one branch, which is master branch and there is one file called hungry.py.txt in it.

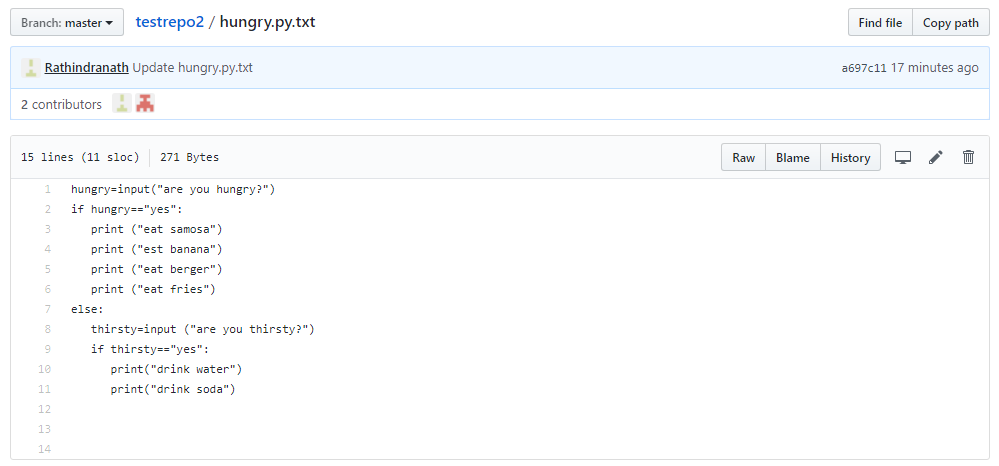




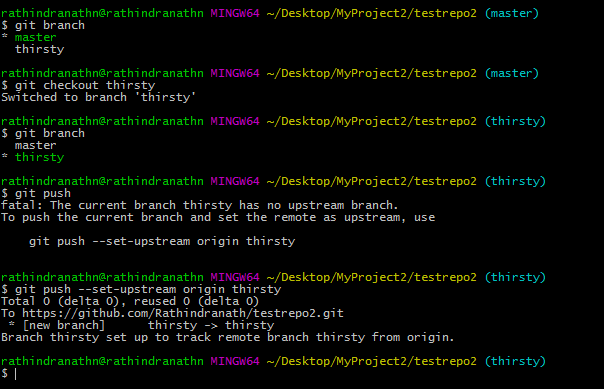
Now we shall push the master branch to the remote repository “testrepo2” in GitHub. Before doing so, let’s check the content of hungry.py.txt in testrepo2 in GitHub.



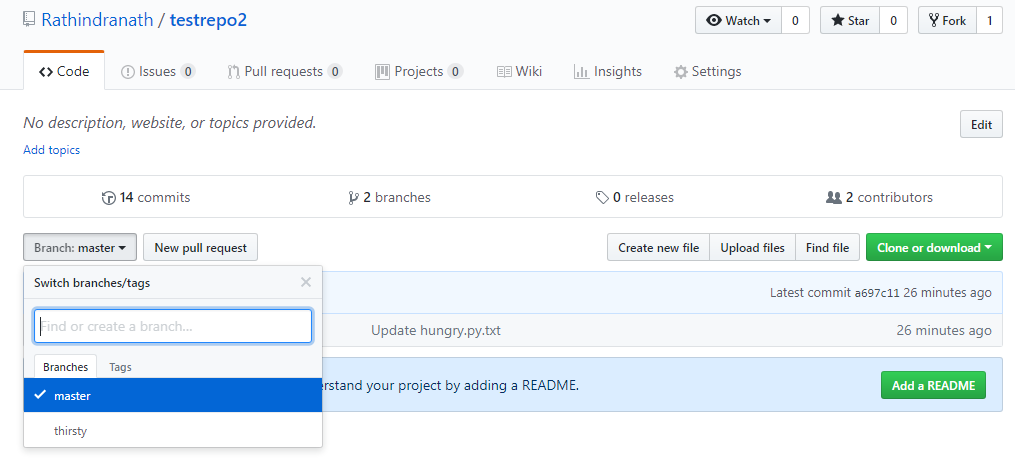
Let’s execute the git bash command “git push” in master branch in our local system and then the file content of hungry.py.txt in GitHub,



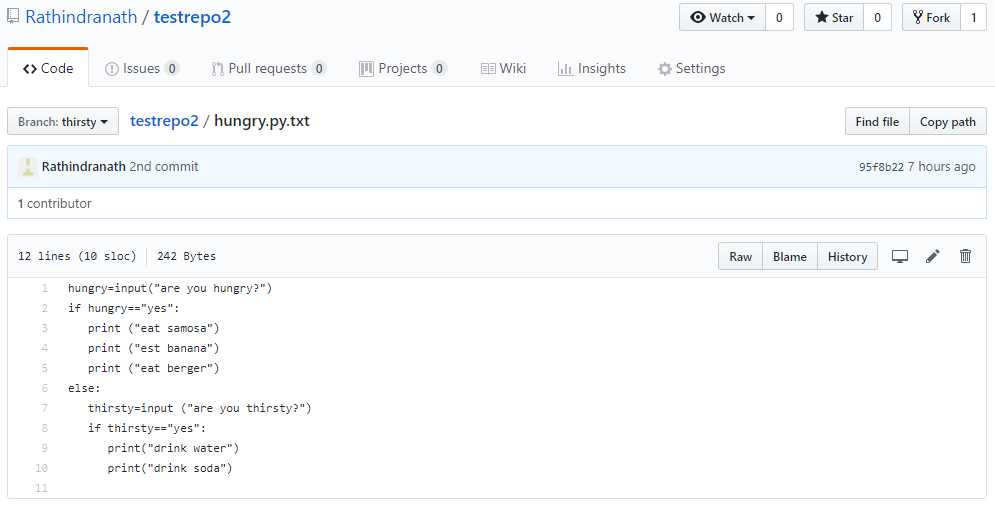
Now we want to push “thirsty” branch to testrepo2 in GitHub. For that we need to checkout “thirsty” branch first and then execute “git push ……” command.



Let’s look at testrepo2 in GitHub,

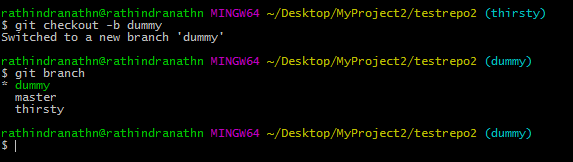


“thirsty” branch has appeared in GitHub. Following is the file content of hungry.py.txt file. The line “print(“eat fries”)” is not there in that file. Therefore same content is reflected from local repository to remote repository as far as “thirsty” branch is concerned.



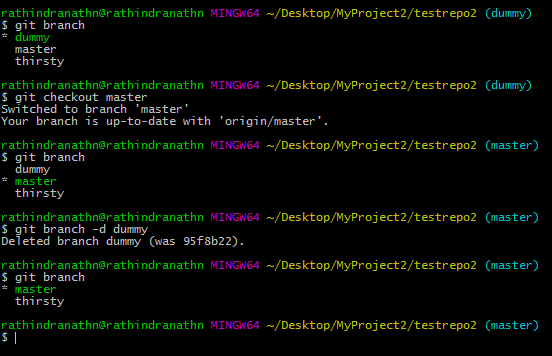
**Deleting a branch:**

Here we shall discuss how to delete a branch. To demonstrate first we shall create a new branch called “dummy”. We shall create the branch and then checkout that branch in a single command,



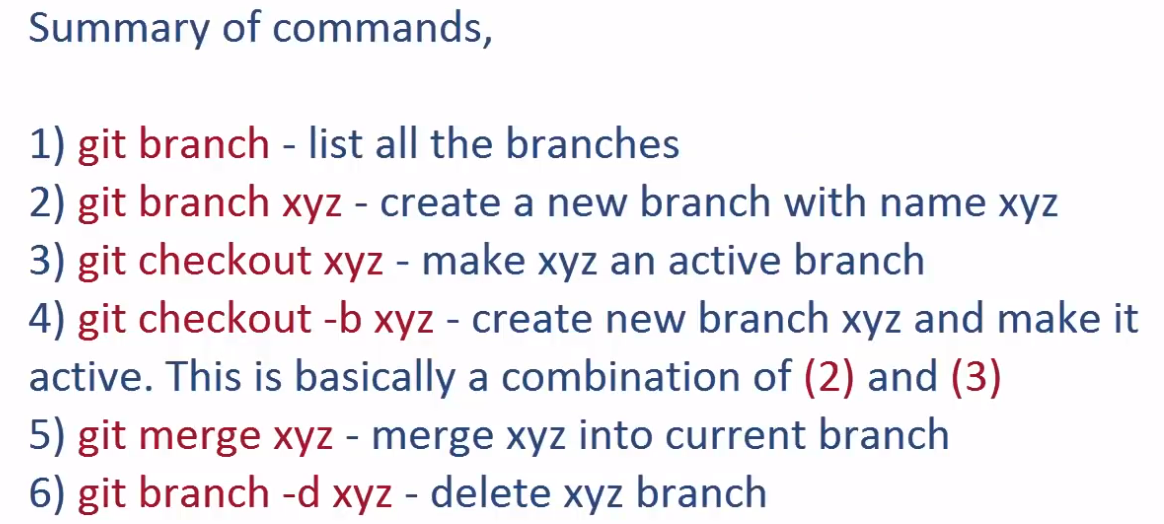
To delete the branch “dummy” we need to first checkout master branch and then execute the following command,

git branch -d <name of the branch to be deleted>



To delete remote branch we need execute following command,

git push origin --delete <name of the branch to be deleted>



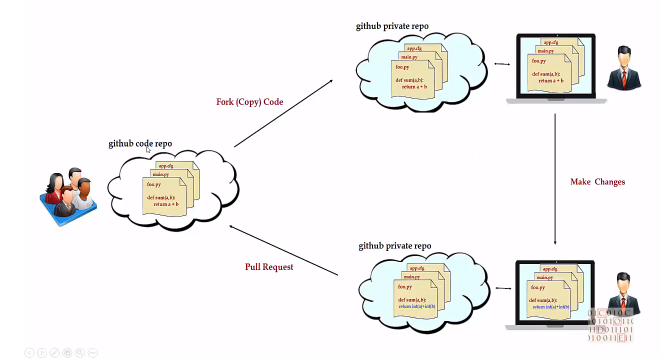
**Git/Github Tutorial 7: What is Pull Request?**

<https://www.youtube.com/watch?v=e3bjQX9jIBk&list=PLeo1K3hjS3usJuxZZUBdjAcilgfQHkRzW&index=7>

In this tutorial we shall discuss what pull request is. Following are few salient points about pull request:

1. On GitHub, owner can share his code with others
2. Other person can make code changes and send a request to owner to pull/merge his code changes into owner’s repository
3. This request in above step is called pull request

Let’s try to understand pull request using the following figure,

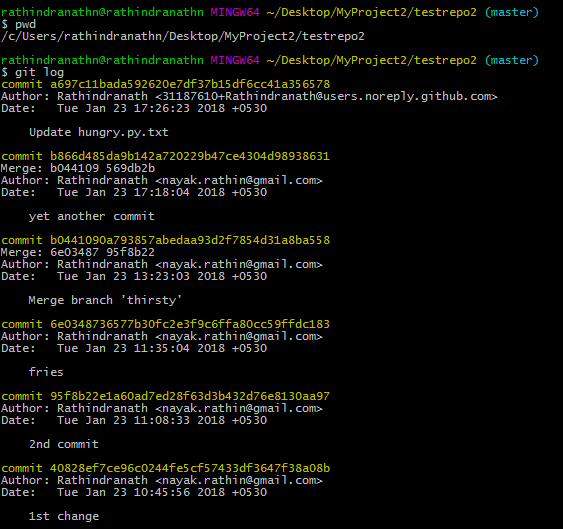


Suppose four people in the figure own the GitHub repository. Let’s say it’s a repository for NodeJS, which is an open source project. One developer looks at the code in that repository and wants to contribute by fixing few bugs and also enhancing few features. So he can fork the project in his private GitHub repository and from there he can clone the project in his local system. He then makes changes to the code in his local system and then commits those changes in the local repository. After that he pushes his change in his GitHub repository. He then raises a pull request to the owner of the code. He requests the owners of the code to pull his code to owner’s repository. Owners then check the difference and then either accept or reject or advise what to do next with appropriate comment. In this way pull request allows collaboration to take place among contributors.

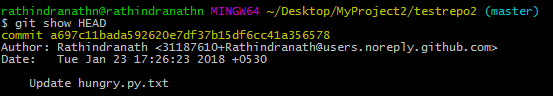
**Git/Github Tutorial 8: What is HEAD?**

<https://www.youtube.com/watch?v=ZaI1co-rt9I&list=PLeo1K3hjS3usJuxZZUBdjAcilgfQHkRzW&index=8>

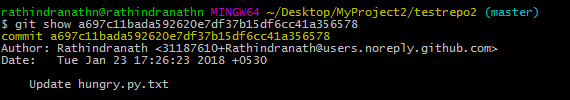
HEAD is a reference to the most recent commit in current branch. Let’s execute the “git log” command in git bash,



Above output is truncated – only few first lines have been taken. HEAD refers to the latest commit, which is a697c11bada592620e7df37b15df6cc41a356578 in this case. We can verify the same by running the following command,

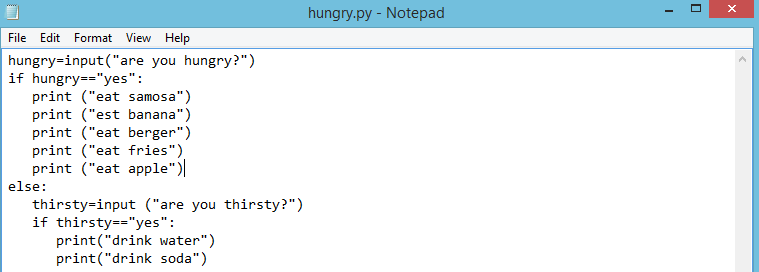


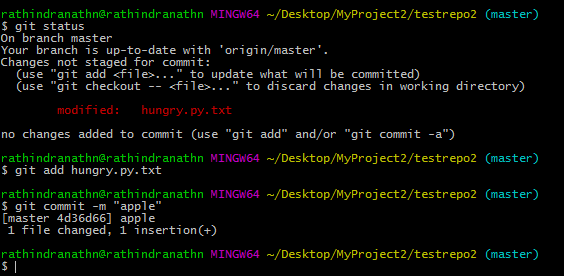
Above command is same as “git show <commit id>”.



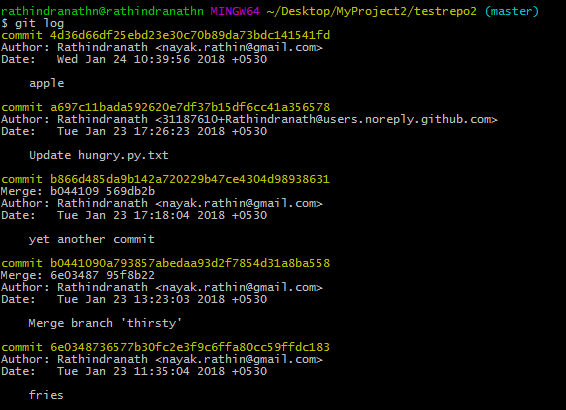
Therefore HEAD is the alternate short version of the 40 character long commit id of the last commit.

Let’s do a modification to the file hungry.py.txt and add a line there,





After modifying the file hungry.py.txt we have committed the change in local repository. Let’s use HEAD,



Above is the truncated output of “git log”.



From above outputs, it is clear that HEAD refers to the latest commit. Therefore HEAD is a pointer, which always moves and point to the latest commit.

We can refer a particular commit in the form of HEAD. The commit, done before the last commit can be referred to as HEAD~1 and the commit before HEAD~1 can be referred to as HEAD~2 and so on. Therefore following commits can be represented alternatively,

4d36d66df25ebd23e30c70b89da73bdc141541fd -> HEAD

a697c11bada592620e7df37b15df6cc41a356578 -> HEAD~1

b866d485da9b142a720229b47ce4304d98938631 -> HEAD~2

If we want to see the difference in the file hungry.py.txt in last two commits we can execute either of the following command,

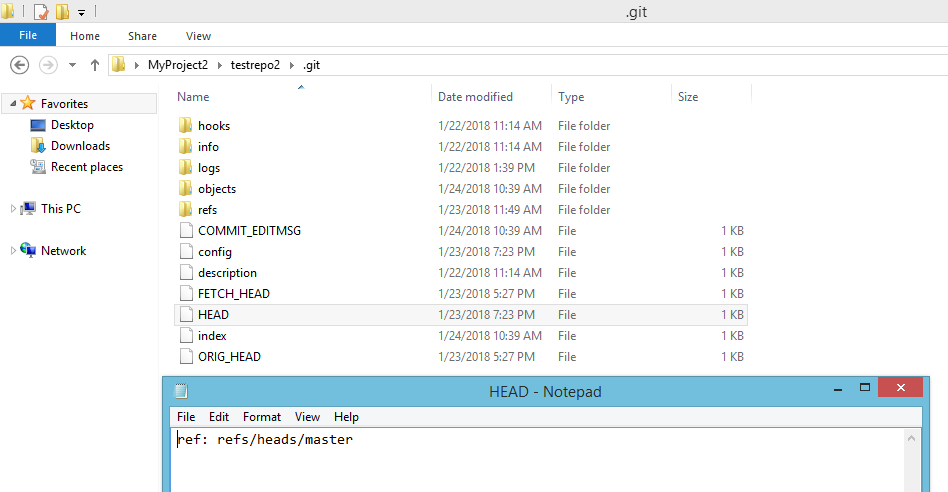
git difftool 4d36d66df b866d485

or

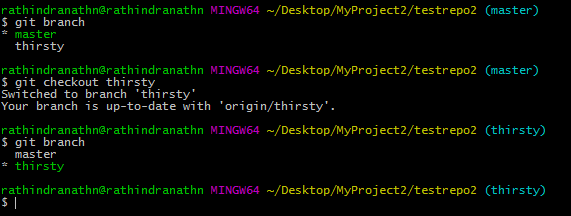
git difftool HEAD~1 HEAD~2

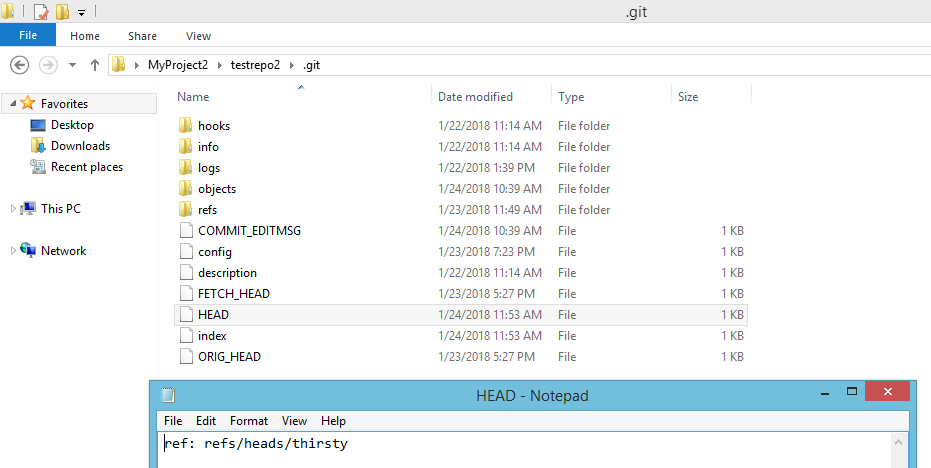
N.B: The character ‘~’ is pronounced as ‘tilda’.

Let’s discuss how git manages HEAD pointer internally. For that we need to go to our project folder or working directory and open the .git folder. Inside it there is one file called “HEAD”. If we open this file there will be line which refers the current branch.



Currently we are in the master branch. Therefore the line, which is there in the file, refers the master branch. Let’s checkout another existing branch and then check the HEAD file inside .git directory.





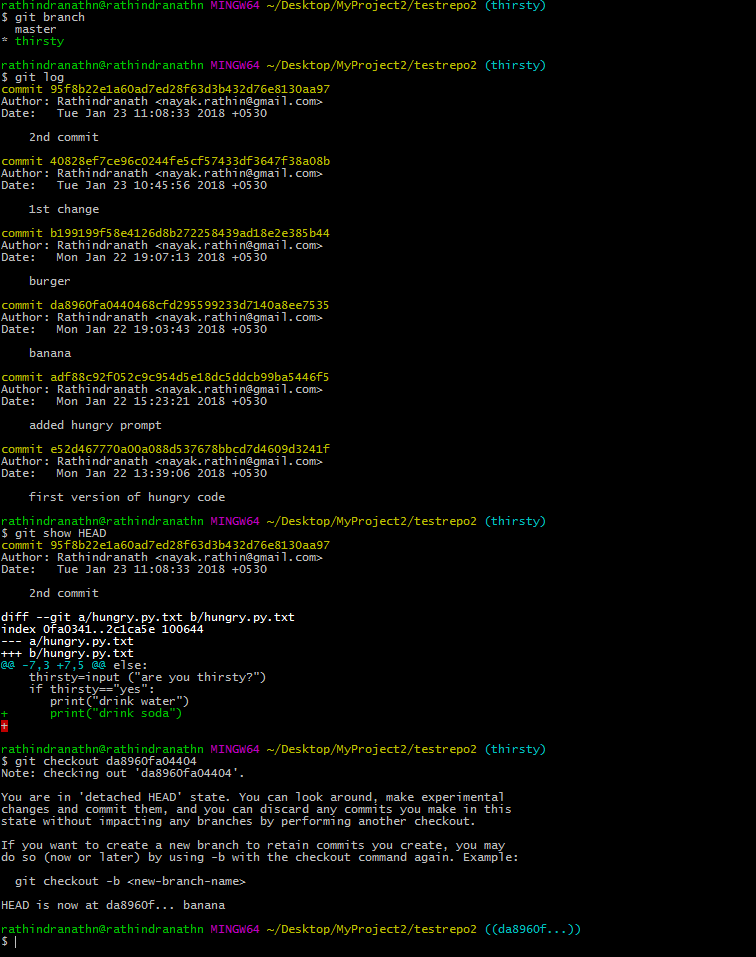
Now the line is referring the “thirsty” branch.

**Scenario where HEAD does not points the most recent commit: detached HEAD**

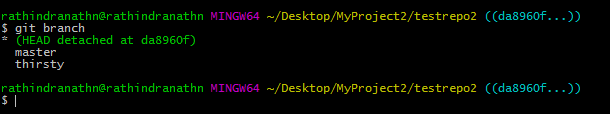
When HEAD doesn’t point to most recent commit we go into detached HEAD state. Suppose we want HEAD points to a commit point, which is not the most recent. For that we need to execute following command,

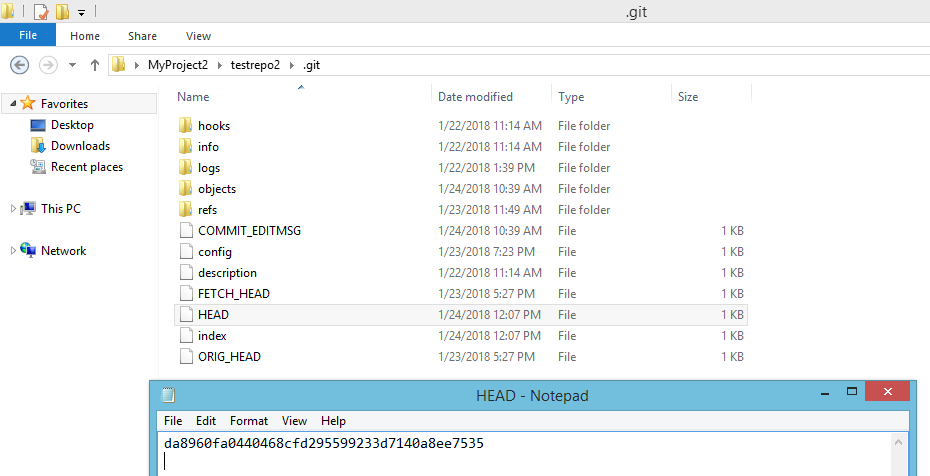
git checkout <commit id of the concerned commit>

Now HEAD will point to the commit point, which is mentioned in the above command. Following is the current scenario. And we want to checkout “banana” commit, whose commit id is da8960fa04404……… After doing so, HEAD will start pointing to commit id da8960fa04404………

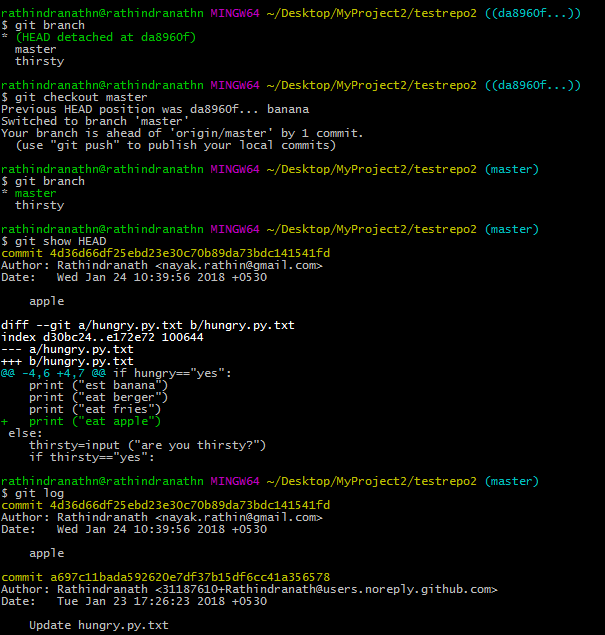


Now we are in detached HEAD state. We can check it in git bash by executing “git branch” command. Also we can check current HEAD in HEAD file in .git folder.





Now we want to come out of the detached HEAD state. For that we need to checkout an existing branch like master.



The output of “git log” is truncated.

Why detached HEAD state is necessary? Sometimes we want to go back to a particular commit and do some changes. Therefore that commit point will be our base. In such scenario detached HEAD state is necessary.