

Purpose	Git commands
To Configure the username and email to be used with your commits.	git config --global user.name "Ankur Jain" git config --global user.emailankurjain@gmail.com
Transform the current directory into Git repository	git init
To create a working copy of a local repository	git clone /path/to/repository git clone https://github.com/venkywarriors/C-Shap- Repository/
Pretty colours to fit terminal	Git config --global color.ui true
To view git configuration	Git config --list
To Add one staging	git add <filename>
Add list of files in current directory	git add .
To Add only one or more txt to staging	git add *.txt
Add all new and updated files throughout project	Git add -A
Add files in all directory level	Git add --all
Add all modifications and deleted files not new created files	Git add -u Git add --update

Viewing information about remote repository	Git remote -v
List no of branches	Git branch -a
Add new files + modifications but ignore deleted files	Git add --ignore-removal
Remove one file from staging area	Git rm -cached FILE_NAME
Remove a directory	Git rm -r -cached directory
Remove a file	Git rm FILE_NAME or git rm -f <filename>
Remove/Discard unstaged changes in current directory	Git checkout -- .
Remove all unstaged files	Git reset Head/git reset
To unstage one file	Git reset Head FILE_NAME
Remove commit	Git reset --soft <Hashcode>
To commit Changes	git commit -m "Commit message"
Add files to staging and commit	Git commit -a -m "Commit message"
Changed a commit message	Git commit --amend -m "Commit message"
Push code to the master branch of remote repository	git push -u origin master
To check git status	git status
To push the code to the remote server	git remote add origin https://GitHub.com.***.git

To get the list of all currently configured remote repositories	git remote -v
To update local repo from global repository	Git pull origin master
To Create and switch to a new branch	git checkout -b <branchname>
To Switch from one branch to another:	git checkout <branchname>
To List all the branches in the repository	git branch
Create a new branch	Git branch new-branch
Clone a branch	Git clone -b BRANCH_NAME URL "new branch name"
To Delete the particular feature branch:	git branch -d <branchname>
To Push the branch to the remote repository.	git push origin <branchname>
To Push all branches to remote repository:	git push --all origin
To Delete a branch on a remote repository:	git push origin :<branchname>
To merge a branch into an active branch:	git merge <branchname>
To view the changes in the file which was staged	Git diff – cached filename
To view difference in two commits	Git difftool Head~2 Head~1 git diff <Hashcode> <hashcode>

<p>To view all the merge conflicts:</p> <p>View the conflicts against the base file:</p> <p>Preview changes, before merging:</p>	<pre>git diff git diff --base &lt;filename&gt; git diff &lt;sourcebranch&gt; &lt;targetbranch&gt;</pre>
<p>To view changes in local repository and remote repository</p>	<pre>Git diff master origin/master</pre>
<p>Print only merge commits</p>	<pre>Git log --merges</pre>
<p>Limits the number of commits to show to 3.</p>	<pre>git log -3</pre>
<p>--stat option prints below each commit entry a list of modified files, how many files were changed, and how many lines in those files were added and removed. It also puts a summary of the information at the end Show statistics for files modified in each commit.</p>	<pre>git log --stat</pre>
<p>To add a new file to the previous commit</p>	<pre>\$ git add new-file \$ git commit --amend</pre>

will remove the last commit from the current branch, but the file changes will stay in your working tree. Also the changes will stay on your index, so following with a git commit will create a commit with the exact same changes as the commit you "removed" before	<code>git reset -- soft HEAD~1</code>
will lose all uncommitted changes in addition to the changes introduced in the last commit. The changes won't stay in your working tree so doing a git status command will tell you that you don't have any changes in your repository	<code>git reset -- hard HEAD~1</code>
To view difference between two commits	<code>Git diff hash1 hash2</code>
Comparing staging area with the repository	<code>Git diff --staged</code>
How to commit directly to the repository without staging	<code>Git commit -am "commit msg"</code>
Git revert file to a specific branch	<code>git checkout branchName_Which_Has_stable_Commit fileName</code>
Git revert file to a specific commit	<code>git checkout Last_Stable_commit_Number -- fileName</code>
<code>git-show</code> is a command line utility that is used to view expanded details on Git objects such as blobs, trees, tags, and commits. <code>git-show</code> has specific behavior per object type	<p><code>Git show head</code></p> <pre>git show commitA...commitD</pre> <p>This will output all commits in the range from commitA to commit D</p>

Comparing staging area with repository	Git diff --staged
Discard changes in file	Git checkout -- filename
Show the whole commit history, but skip any merges	Git log --no-merges
To get the information about the previous commits to the project.	git log

	New Files	Modified Files	Deleted Files	
<code>git add -A</code>	✓	✓	✓	Stage All (new, modified, deleted) files
<code>git add .</code>	✓	✓	✗	Stage New and Modified files only
<code>git add -u</code>	✗	✓	✓	Stage Modified and Deleted files only

### How to copy a branch from one GitHub repository to another?

Simply add the new remote (Organization) to your old repository (master).  
Once you did it simply push the branch A to the new (organization) repository.

```
cd <old repository>
git remote add origin2 <new_url>
git push origin2 <branch A>
```

Now you should have the new branch A in your new repository.

The point is to add new remote and to push the branch to your new repository.

Second way id to update the current repository remote to point to the new location:

```
git remote set-url origin <new url>
```

And then push the branch.

## git remote

```
$ git remote add origin user@server:/path/to/project.git
```

adding a remote branch

```
$ git remote -v
```

```
origin git@github.com:github/git-reference.git (fetch)
```

```
origin git@github.com:github/git-reference.git (push)
```

list the remotes available

```
$ git remote rm origin
```

removing an existing remote alias

## How can I reset or revert a file to a specific revision?

**You can do it in 4 steps:**

1. revert the entire commit with the file you want to specifically revert - it will create a new commit on your branch
2. soft reset that commit - removes the commit and moves the changes to the working area
3. handpick the files to revert and commit them
4. drop all other files in your work area

**What you need to type in your terminal:**

1. `git revert <commit_hash>`
2. `git reset HEAD~1`
3. `git add <file_i_want_to_revert> && git commit -m 'reverting file'`
4. `git checkout .`

I have a commit abc1 and after it I have done several (or one modification) to a file file.txt.

**Now say that I messed up something in the file file.txt and I want to go back to a previous commit abc1.**

1. `git checkout file.txt`: this will remove local changes, if you don't need them
2. `git checkout abc1 file.txt`: this will bring your file to your *wanted* version
3. `git commit -m "Restored file.txt to version abc1"`: this will commit your reversion.
4. `git push`: this will push everything on the remote repository

## Copy files between Git branches

To copy file(s) from from another branch to the current one in Git, there are two possible options:

1. use the `git show` command:

2. `$ git show <branch_name>:path/to/file >path/to/local/file`

3. use the `git checkout` command:

4. `$ git checkout <branch_name> path/to/new/file`

the latter form checkouts a file from another branch and adds it to this branch; the file will still need to be added to the branch with `git add` command, but the file is already present.

## How .gitignore Works

`.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

### Negation

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

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

### How can I reset or revert a file to a specific revision?

Assuming the hash of the commit you want is `c5f567`:

```
git checkout c5f567 -- file1/to/restore
```

## What Is Git Rebase?

Git rebase is a command that allows developers to integrate changes from one branch to another.

## How Does Git Rebase Work?

Git rebase compresses all the changes into a single “patch.” Then it integrates the patch onto the target branch. Unlike merging, rebasing flattens history. It transfers the completed work from one branch to another. In the process, unwanted history is eliminated.(No commit hashcode after merging)

## Git Rebase vs. Merge: Similarities and Differences

Git rebase and merge both integrate changes from one branch into another. Where they differ is how it's done. Git rebase moves a feature branch into a master. Git merge adds a new commit, preserving the history.



Table 2. Common options to `git log`

Option	Description
<code>--name-only</code>	Show the list of files modified after the commit information.
<code>--name-status</code>	Show the list of files affected with added/modified/deleted information as well.
<code>--abbrev-commit</code>	Show only the first few characters of the SHA-1 checksum instead of all 40.
<code>--relative-date</code>	Display the date in a relative format (for example, “2 weeks ago”) instead of using the full date format.
<code>--graph</code>	Display an ASCII graph of the branch and merge history beside the log output.
<code>--pretty</code>	Show commits in an alternate format. Option values include oneline, short, full, fuller, and format (where you specify your own format).
<code>--oneline</code>	Shorthand for <code>--pretty=oneline --abbrev-commit</code> used together
3. Options to limit the output of <code>git log</code>	

Option	Description
<code>-&lt;n&gt;</code>	Show only the last n commits
<code>--since</code> , <code>--after</code>	Limit the commits to those made after the specified date.
<code>--until</code> , <code>--before</code>	Limit the commits to those made before the specified date.
<code>--author</code>	Only show commits in which the author entry matches the specified string.
<code>--committer</code>	Only show commits in which the committer entry matches the specified string.
<code>--grep</code>	Only show commits with a commit message containing the string
<code>-S</code>	Only show commits adding or removing code matching the string

**Cherry picking** in Git means to choose a commit from one branch and apply it onto another.

Scenerio1: Accidently make a commit in a wrong branch.

Git cherry-pick is helpful to apply the changes that are accidentally made in the wrong branch. Suppose I want to make a commit in the master branch, but by mistake, we made it in any other branch. See the below commit.

```
omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$ git add file1.txt

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$ git commit -m " new changes proposes for master branch"
[newbranch 4f7cc74] new changes proposes for master branch
1 file changed, 1 insertion(+)
create mode 100644 file1.txt

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/gitexample2 (newbranch)
$
```

In the above example, I want to make a commit for the master branch, but accidentally I made it in the new branch. To make all the changes of the new branch into the master branch, we will use the git pull, but for this particular commit, we will use git cherry-pick command.

```

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (newbranch)
$ git log
commit e43483b4cddedd1b28ac358857ca75d23326e1f9 (HEAD -> newbranch)
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Mon Sep 16 12:21:55 2019 +0530

    commit for the project's main branch

commit 4f7cc7458f79f8c21fa3563dbecee03381dc9645
Author: ImDwivedi1 <himanshudubey481@gmail.com>
Date: Mon Sep 16 12:02:23 2019 +0530

    new changes proposes for master branch

commit 4a6693a71151323623c04dd75cb0d44c1c4dbadf (origin/master, origin/HEAD)
Merge: 30193f3 78c5fbd
Author: ImDwivedi1 <52317024+ImDwivedi1@users.noreply.github.com>
Date: Mon Sep 9 15:24:13 2019 +0530

    Merge pull request #1 from ImDwivedi1/branch2

    Create merge the branch

```

In the given output, I have used the git log command to check the commit history. Copy the particular commit id that you want to make on the master branch. Now switch to master branch and cherry-pick it there. See the below output:

Syntax:

- \$ git cherry-pick <commit id>

Output:

```

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (newbranch)
$ git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (master)
$

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (master)
$ git cherry-pick e43483b4cddedd1b28ac358857ca75d23326e1f9
[master 16d018c] commit for the project's main branch
Date: Mon Sep 16 12:21:55 2019 +0530
1 file changed, 1 insertion(+)
create mode 100644 mastercommit.txt

omkat@LAPTOP-HNN1S7C5 MINGW64 ~/Desktop/GitExample2 (master)
$ |

```

From the given output, you can see that I have pasted the commit id with git cherry-pick command and made that commit into my master branch. You can check it by git log command.

What happens if we want to **roll back to a previous commit**. For example, if we want to reset master to point to the commit two back from the current commit, we could use either of the following methods

\$ git reset 9ef9173 (using an absolute commit SHA1 value 9ef9173) or

\$ git reset current~2 (using a relative value -2 before the "current" tag)

## Clean command

Step 1 is to show what will be deleted by using the -n option:

# Print out the list of files which will be removed (dry run)

git clean -n

# Delete the files from the repository

git clean -f

- To remove directories, run git clean -f -d or git clean -fd
- To remove ignored files, run git clean -f -X or git clean -fx
- To remove ignored and non-ignored files, run git clean -f -x or git clean -fx

Use git clean -f -d to make sure that **directories** are also removed.

1. Don't actually remove anything, just show what would be done.
2. git clean -n or git clean --dry-run
3. Remove untracked directories in addition to untracked files. If an untracked directory is managed by a different Git repository, it is not removed by default. Use the -f option twice if you really want to remove such a directory.
4. git clean -fd

You can then check if your files are really gone with git status.

## git reflog

Git keeps track of updates to the tip of branches using a mechanism called reference logs, or "reflogs." (you can see what you have done so far)

You can also use the `--all` option to get more detailed information about different branches and even the stash

```
gitready master $ git reflog --all
6c8749c... refs/heads/master@{0}: commit: Switching to
94cd08f... refs/heads/pt-br@{0}: commit: Fixing index
8a8cbf3... refs/heads/pt-br@{1}: commit: Hiding some
e6ca7c5... refs/heads/pt-br@{2}: merge master: Merge
c3a7284... refs/heads/master@{1}: commit: Bringing out
208e4e4... refs/heads/pt-br@{3}: rebase: updating HEAD
2c92f41... refs/heads/pt-br@{4}: rebase finished: refs
208e4e4... refs/heads/pt-br@{5}: HEAD~1: updating HEAD
ebb2d68... refs/heads/pt-br@{6}: commit: Hiding some
208e4e4... refs/heads/pt-br@{7}: pull taylorrf pt-br:
a012e5b... refs/heads/master@{2}: commit: Looks like y
9f1c6f0... refs/heads/master@{3}: commit: git aliases
0dc68db... refs/heads/master@{4}: commit: Trying out
e96610a... refs/stash@{0}: WIP on master: b27f606...
```

```
git revert a72ef023
```

```
$ git revert HEAD
[master b9cd081] Revert "prepend content to demo file"
1 file changed, 1 deletion(-)
```

Git `revert` expects a commit ref was passed in and will not execute without one. Here we have passed in the `HEAD` ref. This will revert the latest commit. This is the same behavior as if we reverted to commit `3602d8815dbfa78cd37cd4d189552764b5e96c58`. Similar to a merge, a revert will create a new commit which will open up the configured system editor prompting for a new commit message. Once a commit message has been entered and saved Git will resume operation. We can now examine the state of the repo using `git log` and see that there is a new commit added to the previous log:

```
$ git log --oneline
1061e79 Revert "prepend content to demo file"
86bb32e prepend content to demo file
3602d88 add new content to demo file
299b15f initial commit
```

Note that the 3rd commit is still in the project history after the revert. Instead of deleting it, `git revert` added a new commit to undo its changes.

```
$ git fetch origin
```

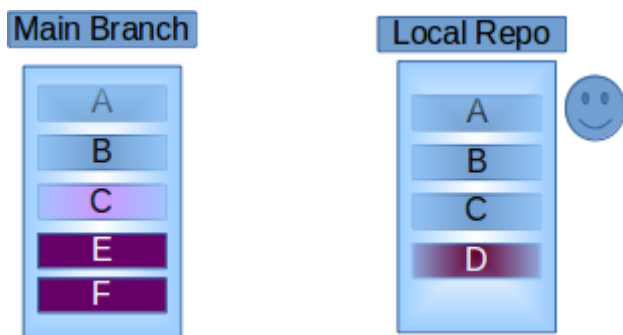
git fetch really only downloads new data from a remote repository - but it doesn't integrate any of this new data into your working files.

Git fetch origin <branch>. It would fetch all the changes into your computer, but keep it separate from your local development/workspace.

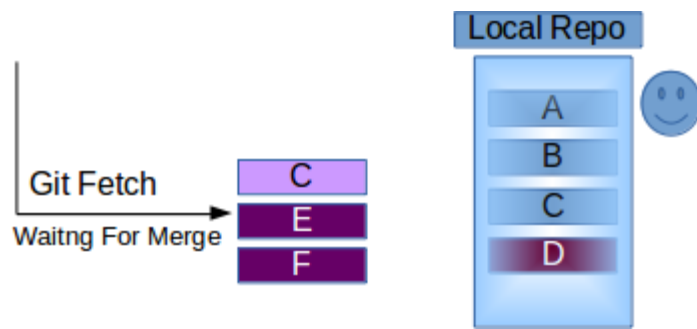
```
$ git pull origin master
```

git pull, in contrast, is used with a different goal in mind: to update your current HEAD branch with the latest changes from the remote server. This means that pull not only downloads new data; it also directly integrates it into your current working copy files.

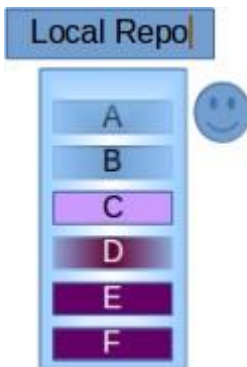
Git fetch + git merge = git pull



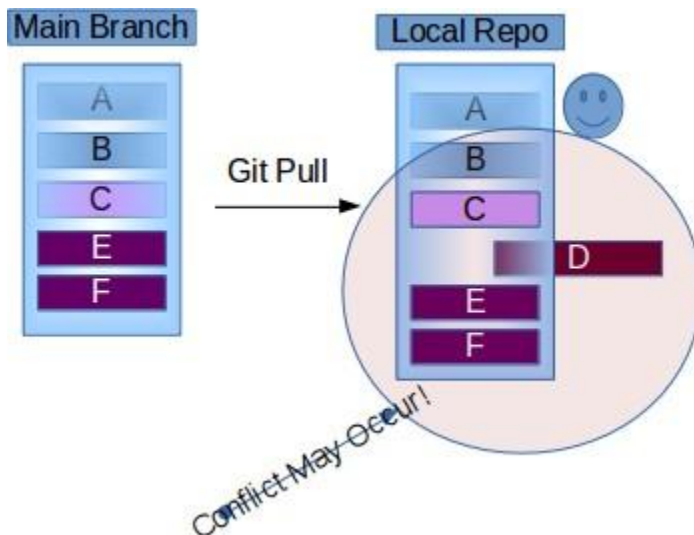
**1. Git Fetch-** This will Download all the changes that have been made to the origin/main branch project which are not present in your local branch. And will wait for the Git Merge command to apply the changes that have been fetched to your Repository or branch.



So now You can carefully monitor the files before merging it to your repository. And you can also modify D if required because of Modified c.



**2. Git Pull-** This will update your local branch with the origin/main branch i.e. actually what it does is a combination of Git Fetch and Git merge one after another. *But this may Cause Conflicts to occur, so it's recommended to use Git Pull with a clean copy.*



Let's say you had commits:

C  
B  
A

`git revert B`, will create a commit that undoes changes in B.

`git revert A`, will create a commit that undoes changes in A, but will not touch changes in B

Note that if changes in B are dependent on changes in A, the revert of A is not possible.

`git reset --soft A`, will change the commit history and repository; staging and working directory will still be at state of C.

`git reset --mixed A`, will change the commit history, repository, and staging; working directory will still be at state of C.

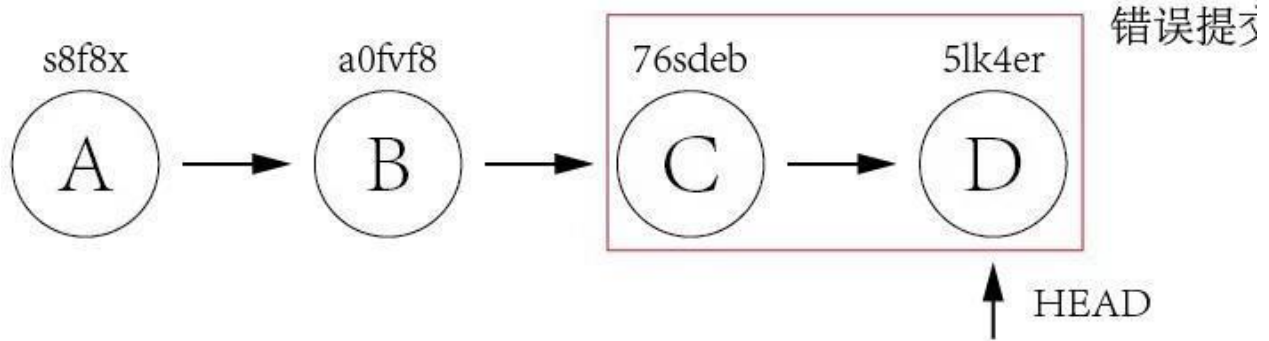
`git reset --hard A`, will change the commit history, repository, staging and working directory; you will go back to the state of A completely.

## git reset vs git revert

When maintaining code using version control systems such as git, it is unavoidable that we need to rollback some wrong commits either due to bugs or temp code revert.

### git reset

Assuming we have below few commits.



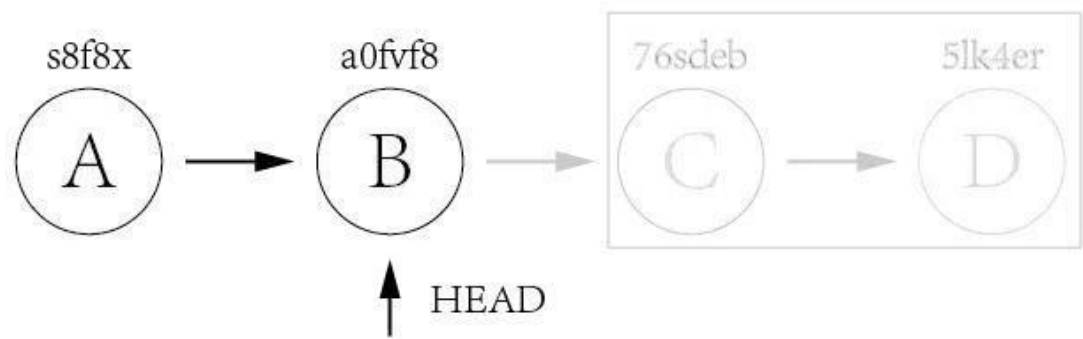
Commit A and B are working commits, but commit C and D are bad commits. Now we want to rollback to commit B and drop commit C and D. Currently HEAD is pointing to commit D **5lk4er**, we just need to point HEAD to commit B **a0fvf8** to achieve what we want.

It's easy to use `git reset` command.

```
git reset --hard a0fvf8
```

After executing above command, the HEAD will point to commit B.





But now the remote origin still has HEAD point to commit D, if we directly use `git push` to push the changes, it will not update the remote repo, we need to add a `-f` option to force pushing the changes.

```
git push -f
```

The drawback of this method is that all the commits after HEAD will be gone once the reset is done.

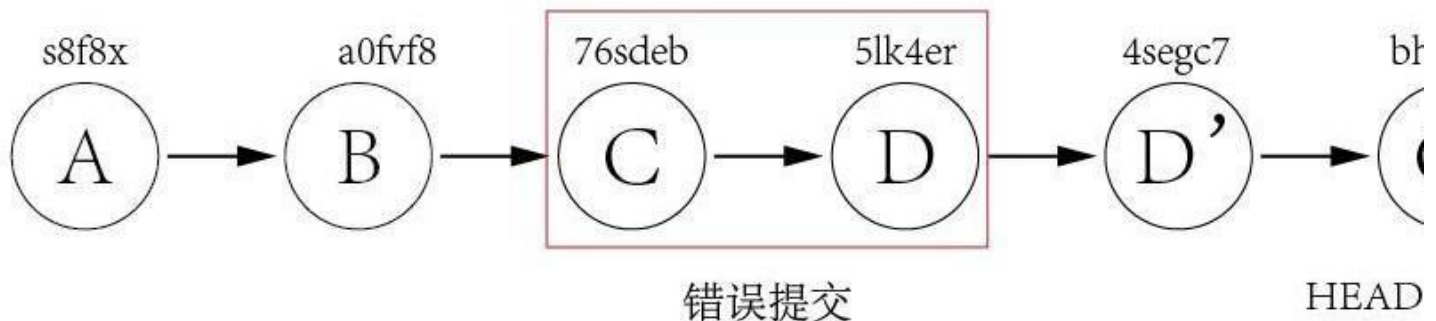
## git revert

The use of `git revert` is to create a new commit which reverts a previous commit. The HEAD will point to the new reverting commit.

For the example of `git reset` above, what we need to do is just reverting commit D and then reverting commit C.

```
git revert 5lk4er
git revert 76sdeb
```

Now it creates two new commit D' and C'



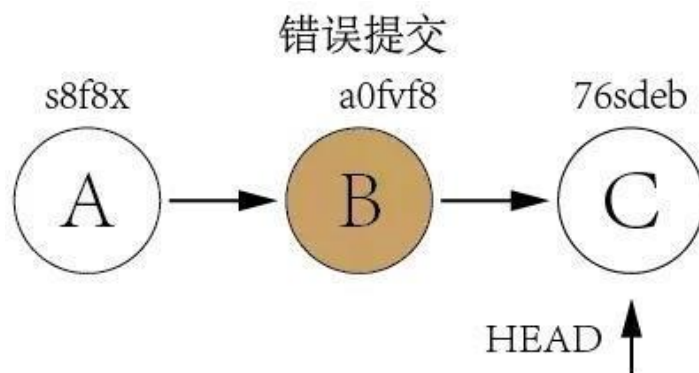
In above example, we have only two commits to revert, so we can revert one

by one. But what if there are lots of commits to revert? We can revert a range indeed.

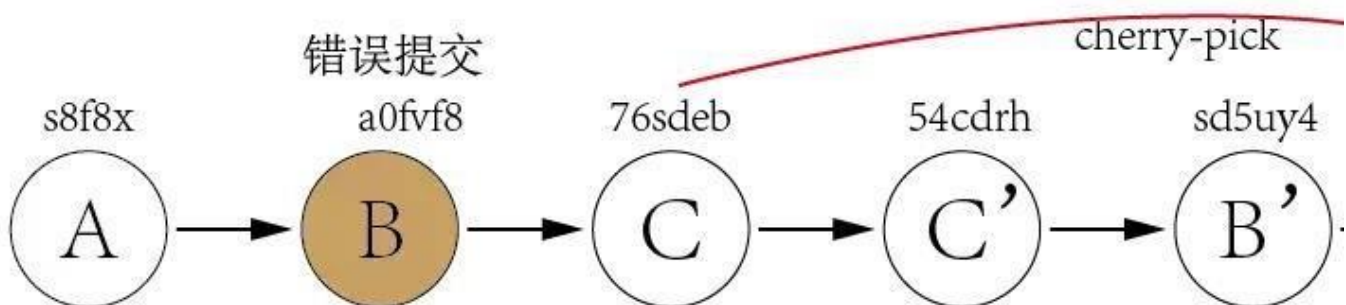
```
git revert OLDER_COMMIT^..NEWER_COMMIT
```

This method would not have the disadvantage of git reset, it would point HEAD to newly created reverting commit and it is ok to directly push the changes to remote without using the -f option.

Now let's take a look at a more difficult example. Assuming we have three commits but the bad commit is the second commit.



It's not a good idea to use git reset to rollback the commit B since we need to keep commit C as it is a good commit. Now we can revert commit C and B and then use cherry-pick to commit C again.



From above explanation, we can find out that the biggest difference between git reset and git revert is that git reset will reset the state of the branch to a previous state by dropping all the changes post the desired commit while git revert will reset to a previous state by creating new reverting commits and keep the original commits. It's recommended to use git revert instead of git reset in enterprise environment.

## Git Checkout File

Checking out a file is similar to using `git reset` with a file path, except it updates the working directory instead of the stage. Unlike the commit-level version of this command, this does not move the HEAD reference, which means that you won't switch branches.

For example, the following command makes `foo.py` in the working directory match the one from the 2nd-to-last commit:

```
git checkout HEAD~2 foo.py
```

Just like the commit-level invocation of `git checkout`, this can be used to inspect old versions of a project—but the scope is limited to the specified file.

If you stage and commit the checked-out file, this has the effect of “reverting” to the old version of that file. Note that this removes all of the subsequent changes to the file, whereas the `git revert` command undoes only the changes introduced by the specified commit.

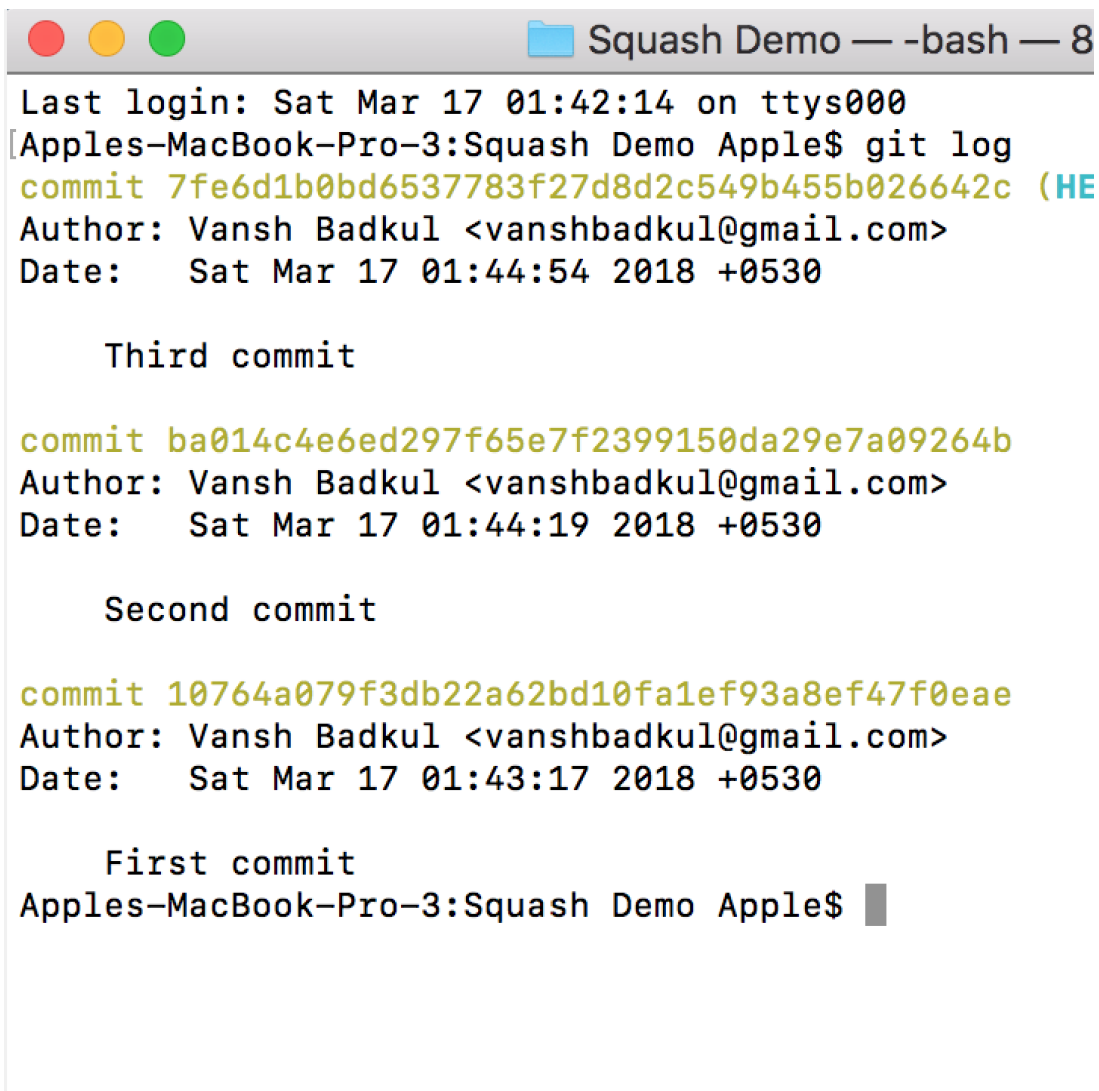
Like `git reset`, this is commonly used with `HEAD` as the commit reference. For instance, `git checkout HEAD foo.py` has the effect of discarding unstaged changes to `foo.py`. This is similar behavior to `git reset HEAD --hard`, but it operates only on the specified file.

Command	Scope	Common use cases
<code>git reset</code>	Commit-level	Discard commits in a private branch or throw away uncommitted changes
<code>git reset</code>	File-level	Unstage a file
<code>git checkout</code>	Commit-level	Switch between branches or inspect old snapshots
<code>git checkout</code>	File-level	Discard changes in the working directory
<code>git revert</code>	Commit-level	Undo commits in a public
<code>git revert</code>	File-level	(N/A)

# Squashing

In Git you can merge several commits into one with the powerful interactive rebase.

1). Check your commit history:*git log*

A terminal window titled "Squash Demo — -bash — 8" with three colored window control buttons (red, yellow, green) on the left. The terminal shows the output of the 'git log' command. It lists three commits in reverse chronological order. The first commit (top) is 'commit 7fe6d1b0bd6537783f27d8d2c549b455b026642c (HEAD)' by Vansh Badkul. The second commit is 'commit ba014c4e6ed297f65e7f2399150da29e7a09264b' by Vansh Badkul. The third commit (bottom) is 'commit 10764a079f3db22a62bd10fa1ef93a8ef47f0eae' by Vansh Badkul. The terminal ends with the prompt 'Apples-MacBook-Pro-3:Squash Demo Apple\$' and a cursor.

```
Last login: Sat Mar 17 01:42:14 on ttys000
[Apples-MacBook-Pro-3:Squash Demo Apple$ git log
commit 7fe6d1b0bd6537783f27d8d2c549b455b026642c (HEAD
Author: Vansh Badkul <vanshbadkul@gmail.com>
Date:   Sat Mar 17 01:44:54 2018 +0530

    Third commit

commit ba014c4e6ed297f65e7f2399150da29e7a09264b
Author: Vansh Badkul <vanshbadkul@gmail.com>
Date:   Sat Mar 17 01:44:19 2018 +0530

    Second commit

commit 10764a079f3db22a62bd10fa1ef93a8ef47f0eae
Author: Vansh Badkul <vanshbadkul@gmail.com>
Date:   Sat Mar 17 01:43:17 2018 +0530

    First commit
Apples-MacBook-Pro-3:Squash Demo Apple$
```

2). At this moment the current HEAD is at the last commit, in my case it's the "Third commit".

Now comes the squashing part! Let's say you want to squash the last 3 commits (i.e commit pointing to the current HEAD + the previous 2 commits) :

Get the previous 2 commits from the current HEAD, use:

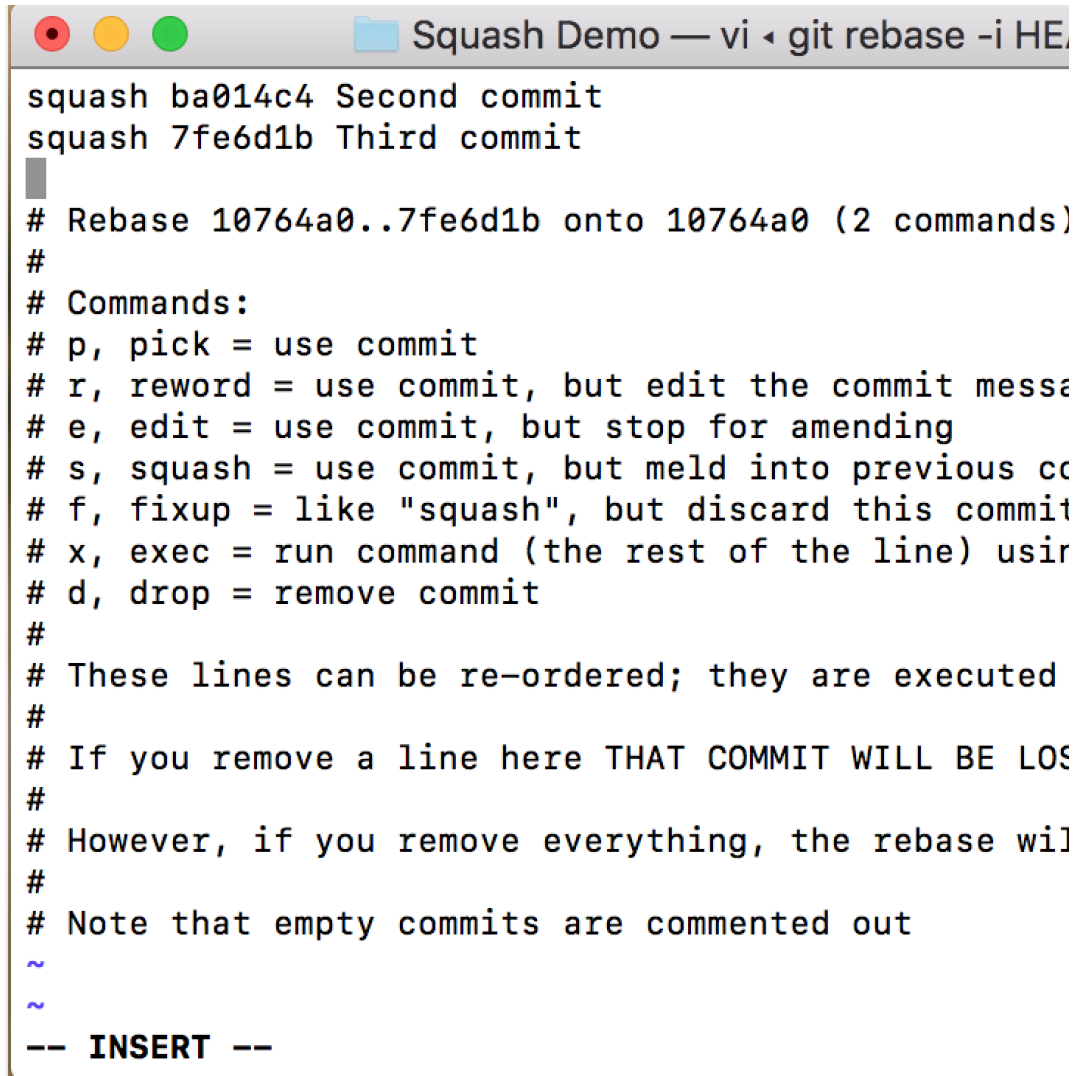
```
git rebase -i HEAD~2
```

Note: You can get previous "n" commits, depends on your case.

```
Squash Demo — vi ◀ git rebase -i HEAD~2
pick ba014c4 Second commit
pick 7fe6d1b Third commit

# Rebase 10764a0..7fe6d1b onto 10764a0 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's commit message
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed in sequence.
#
# If you remove a line here THAT COMMIT WILL BE LOST.
#
# However, if you remove everything, the rebase will abort.
#
# Note that empty commits are commented out
~
~
"~/Desktop/Squash Demo/.git/rebase-merge/git-rebase"
```

3) Choose the commits that you want to squash. For that, replace “pick” with “squash” to the commits you want to squash.



```
Squash Demo — vi ◀ git rebase -i HEAD~2
squash ba014c4 Second commit
squash 7fe6d1b Third commit
# Rebase 10764a0..7fe6d1b onto 10764a0 (2 commands)
#
# Commands:
# p, pick = use commit
# r, reword = use commit, but edit the commit message
# e, edit = use commit, but stop for amending
# s, squash = use commit, but meld into previous commit
# f, fixup = like "squash", but discard this commit's
# x, exec = run command (the rest of the line) using shell
# d, drop = remove commit
#
# These lines can be re-ordered; they are executed in
# the order you want them to be executed.
# If you remove a line here THAT COMMIT WILL BE LOST.
# However, if you remove everything, the rebase will
# abort.
# Note that empty commits are commented out
~
~
-- INSERT --
```

Note: Press “i” button to edit in the nano editor.

Notice the third line in the editor : It says Rebase <commit-id-1>..<commit-id-2> onto <commit-id-3> which means that you can choose to squash your commits from commit-id-2 to commit-id-3 into commit-id-1

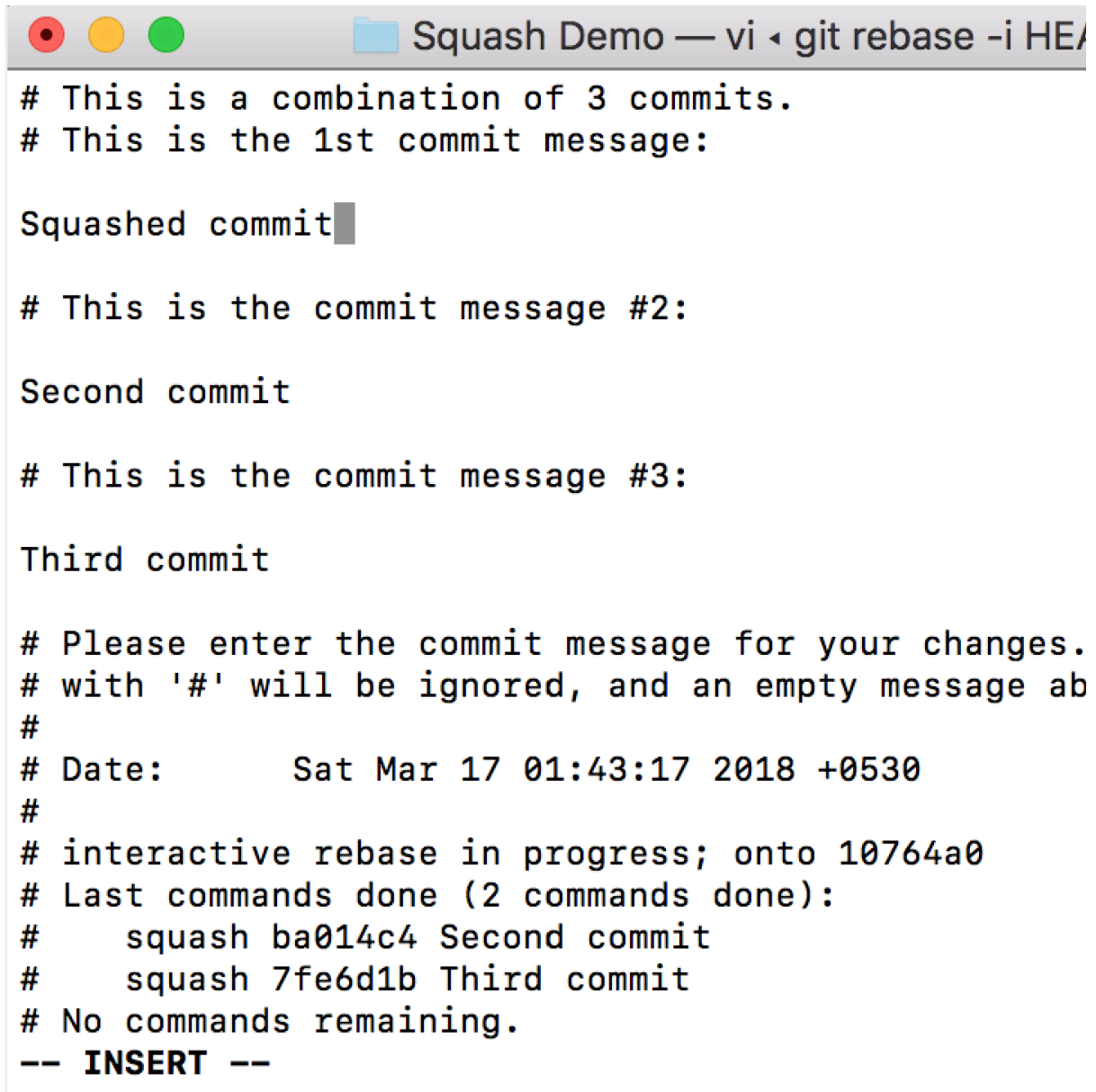
In my case : It will mean that I’m squashing “Second commit” and “Third commit” into my “First commit”, which will result in a combination of 3 commits.

Save and exit the editor :

*:wq*

4) On pressing enter, you'll get a screen with the editor where you can edit your commit messages.

I'm editing my 1st commit message to ("Squashed commit") because this will be the combined commit message.



```
Squash Demo — vi git rebase -i HEAD~3
# This is a combination of 3 commits.
# This is the 1st commit message:

Squashed commit

# This is the commit message #2:

Second commit

# This is the commit message #3:

Third commit

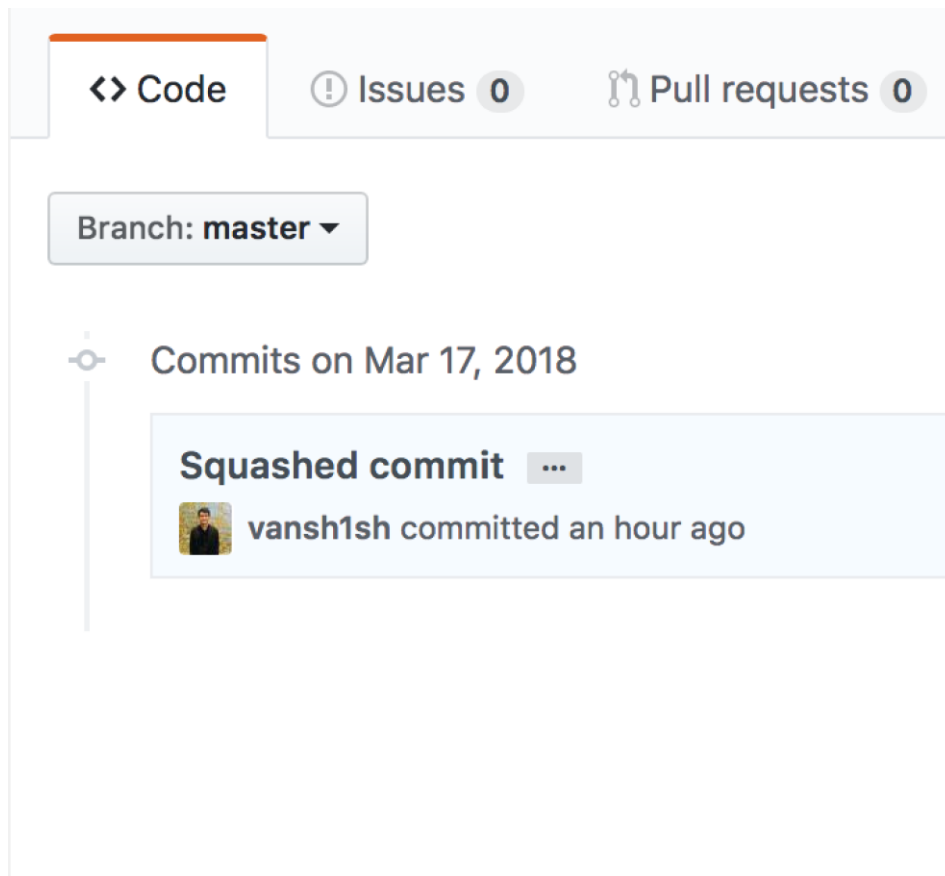
# Please enter the commit message for your changes.
# with '#' will be ignored, and an empty message aborts the rebase.
#
# Date:      Sat Mar 17 01:43:17 2018 +0530
#
# interactive rebase in progress; onto 10764a0
# Last commands done (2 commands done):
#   squash ba014c4 Second commit
#   squash 7fe6d1b Third commit
# No commands remaining.
-- INSERT --
```

Save and exit the editor.

5). Use this command to push the changes to your github repository:

*git push -f origin master*

You can check the squashed commit on your online Github repository. Also now, when you check your PR, the multiple commits will be updated to a single commit automatically.



## git stash - How to Save Your Changes Temporarily

The "git stash" command can help you to (temporarily but safely) store your uncommitted local changes - and leave you with a clean working copy

```
$ git stash
```

```
Saved working directory and index state WIP on master:
```

```
2dfe283 Implement the new login box
```

```
HEAD is now at 2dfe283 Implement the new login box
```



Your working copy is now clean: all uncommitted local changes have been saved on this kind of "clipboard" that Git's Stash represents. You're ready to start your new task (for example by pulling changes from remote or simply switching branches).

## Continuing Where You Left Off

---

As already mentioned, Git's Stash is meant as a temporary storage. When you're ready to continue where you left off, you can restore the saved state easily:

```
$ git stash pop
```

The "pop" flag will reapply the *last saved* state and, at the same time, delete its representation on the Stash (in other words: it does the clean-up for you).

In case you want to apply a specific Stash item (not the most recent one), you can provide the index name of that item in the "pop" option:

```
$ git stash pop stash@{2}
```

1. Git stash save
2. Git stash list
3. Git stash apply
4. Git stash pop
5. Git stash show
6. Git stash branch <name>
7. Git stash clear
8. Git stash drop

### **Git stash save**

This command is like Git stash. But this command comes with various options. I will discuss some important options in this post.

### **Git stash with message**

```
git stash save "Your stash message".
```

The above command stashes with a message. We will see how this is helpful in a bit.

### **Stashing untracked files**

You can also stash untracked files.

```
git stash save -u  
  
or  
  
git stash save --include-untracked
```

### ***Git stash list***

Before discussing this command, let me tell you something about how stash works.

When you Git stash or Git stash save, Git will actually create a Git commit object with some name and then save it in your repo.

```
git stash list
```

So it means that you can view the list of stashes you made at any time. See the example below:

```
stash@{0}: On master: Stash with message  
stash@{1}: WIP on master: 11b8136 Initial commit  
(END)
```

git stash list example

You can see the list of stashes made. And the most recent stash made is in the top.

### ***Git stash apply***

This command takes the top most stash in the stack and applies it to the repo. In our case it is **stash@{0}**

If you want to apply some other stash you can specify the stash id.

```
git stash apply stash@{1}
```

### ***Git stash pop***

This command is very similar to stash apply but it deletes the stash from the stack after it is applied.

```
stash@{0}: WIP on master: 11b8136 Initial commit  
(END)
```

#### Git stash pop example

As you can see the top stash is deleted and **stash@{0}** is updated with older stash.

Likewise, if you want a particular stash to pop you can specify the stash id.

```
git stash pop stash@{1}
```

#### ***Git stash show***

This command shows the summary of the stash diffs. The above command considers only the latest stash.

Here's the example:

```
second.md | 1 +  
third.md  | 3 +++  
2 files changed, 4 insertions(+)  
(END)
```

#### Git stash show example

If you want to see the full diff, you can use

```
git stash show -p
```

Likewise with other commands, you can also specify the stash id to get the diff summary.

```
git stash show stash@{1}
```

### ***Git stash branch <name>***

This command creates a new branch with the latest stash, and then deletes the latest stash ( like stash pop).

If you need a particular stash you can specify the stash id.

```
git stash branch <name> stash@{1}
```

This will be useful when you run into conflicts after you've applied the stash to the latest version of your branch.

### ***Git stash clear***

This command deletes all the stashes made in the repo. It maybe impossible to revert.

### ***Git stash drop***

This command deletes the latest stash from the stack. But use it with caution, it maybe be difficult to revert.

```
git stash drop stash@{1}
```

## Git tree movements visualized

