Git Cheat Sheet

The essential Git commands every developer must know

CONTENTS

CREATING SNAPSHOTS

INITIALIZING A REPOSITORY

```
git init
```

STAGING FILES

```
git add file1.js # Stages a single file
git add file1.js file2.js # Stages multiple files
git add *.js # Stages with a pattern
git add . # Stages the current directory and all its content
```

VIEWING THE STATUS

```
git status # Full status
git status -s # Short status
```

COMMITTING THE STAGED FILES

```
git commit -m “Message” # Commits with a one-line message
git commit # Opens the default editor to type a long message
```

SKIPPING THE STAGING AREA

```
git commit -am “Message” # git add and git commit –m combined
```

REMOVING FILES

```
git rm file1.js # Removes from working directory and staging area
git rm --cached file1.js # Removes from staging area only
```

RENAMING OR MOVING FILES

```
git mv file1.js file1.txt
```

VIEWING THE STAGED/UNSTAGED CHANGES

```
git diff # Shows unstaged changes
git diff --staged # Shows staged changes
git diff --cached # Same as the above
```

VIEWING THE HISTORY

```
git log # Full history
git log --oneline # Summary
git log --reverse # Lists the commits from the oldest to the newest
git log --oneline --all --graph # Summary with graphical representation of different branches
```

VIEWING A COMMIT

```
git show 921a2ff # Shows the given commit
git show HEAD # Shows the last commit
git show HEAD~2 # Two steps before the last commit
git show HEAD:file.js # Shows the version of file.js stored in the last commit
```

UNSTAGING FILES (UNDOING GIT ADD)

```
git restore --staged file.js # Copies the last version of file.js from repo to index
```

DISCARDING LOCAL CHANGES

```
git restore file.js # Copies file.js from index to working directory
git restore file1.js file2.js # Restores multiple files in working directory
git restore . # Discards all local changes (except untracked files)
git clean -fd # Removes all untracked files
```

RESTORING AN EARLIER VERSION OF A FILE

```
git restore --source=HEAD~2 file.js
```

BROWSING HISTORY

VIEWING THE HISTORY

```
git log --stat # Shows the list of modified files
git log --patch # Shows the actual changes (patches)
```

FILTERING THE HISTORY

```
git log -3 # Shows the last 3 entries
git log --author="Mosh"
git log --before="2020-08-17"
git log --after="one week ago"
git log --grep="GUI" # Commits with "GUI" in their message
git log -S"GUI" # Commits with "GUI" in their patches
git log hash1..hash2 # Range of commits
git log file.txt # Commits that touched file.txt
```

FORMATTING THE LOG OUTPUT

```
git log --pretty=format:"%an committed %H"
git log --pretty=format:"%Cgreen%an%Creset committed %h on %cd"
```

CREATING AN ALIAS

```
git config --global alias.lg "log --pretty=format:'%an committed %h'" # created the alias with name lg, then we can use this alias as >git lg
```

VIEWING A COMMIT

```
git show HEAD~2
git show HEAD~2:file1.txt # Shows the version of file stored in this commit
git show HEAD~2 --name-only # Shows the names of the files stored in this commit
git show HEAD~2 --name-status # Shows the names of the files alon with the status (added, modified, deleted) stored in this commit
```

COMPARING COMMITS

git diff HEAD~2 HEAD # Shows the changes between two commits

git diff HEAD~2 HEAD --name-only # Shows the changes between two commits with file names

git diff HEAD~2 HEAD --name-status # Shows the changes between two commits with file names and their status

git diff HEAD~2 HEAD file.txt # Changes to file.txt only

CHECKING OUT A COMMIT

git checkout dad47ed # Checks out the given commit (complete snapshot of the project at a given commit/time)

Here HEAD points to dad47ed (called as 'Detached HEAD'). Make sure you don't commit anything when HEAD is not pointed to MASTER or any particular BRANCH. If we commit new changes to this (dad47ed) checked out commit and later change the HEAD to MASTER or BRANCH, the commits made to the checked out dad47ef commit will be lost as GIT tracks such dead commits and removes them periodically as it is a dead commit.

Since in GIT we can have multiple braches and a Master, GIT needs to know which branch (or commit in case of detached HEAD) we are currently working on. To do that, GIT uses another special pointer called HEAD. So HEAD points to the current branch (or commit in case of detached HEAD) we are working on.

git checkout master # Checks out the master branch

FINDING A BAD COMMIT

git bisect start # initiate to find the bad commit

git bisect bad # Marks the current commit (where HEAD currently points to) as a bad commit

git bisect good ca49180 # Marks the given commit as a good commit

git bisect reset # Terminates the bisect session

FINDING CONTRIBUTORS

git shortlog # Shows all the people who have contributed to our project.

There are multiple option to this command, which we can explore.

git shortlog –n –e –s

VIEWING THE HISTORY OF A FILE

git log file.txt # Shows the commits that touched file.txt

git log --stat file.txt # Shows statistics (the number of changes) for file.txt

git log --patch file.txt # Shows the patches (changes) applied to file.txt

RESTORING A DELETED FILE

git checkout a642e12 toc.txt # Restores the given file (toc.txt) from given commit (a642e12)

FINDING THE AUTHOR OF LINES

git blame file.txt # Shows the author of each line in file.txt

git blame –e file.txt # Shows the author (with email) of each line in file.txt

git blame –L 1,3 file.txt # Shows the author of lines between 1 and 3 in file.txt

TAGGING (TO BOOKMARK CERTAIN POINTS IN THE HISTORY OF OUR PROJECT)

git tag v1.0 # Tags (lightweight – just ref) the last commit as v1.0

git tag v1.0 5e7a828 # Tags an earlier commit

git tag -a v1.1 -m “My Version 1.1” # Creates Annotated (a tag with message) Tag from the last commit.

git tag # Lists all the tags

git tag -n # Lists all the tags with messages

git tag -d v1.0 # Deletes the given tag

git checkout v1.0 # to checkout the tag v1.0

BRANCHING & MERGING

Branch is just a pointer to a commit.

MANAGING BRANCHES

git branch bugfix # Creates a new branch called bugfix

git branch -m bugfix bugfix/login-form # rename a branch

git checkout bugfix # Switches to the bugfix branch

git switch bugfix # Same as the above

git switch -C bugfix # Creates and switches

git switch -C feature/email origin/feature/email # Sets up feature/email branch to track remote branch origin/feature/email branch. This is useful when the remote branch is already create in source repo (github) and we want to switch and work on it.

git branch -d bugfix # Deletes the bugfix branch

COMPARING BRANCHES

git log master..bugfix # Lists the commits in the bugfix branch which are not in master

git diff master..bugfix # Shows the summary of changes

git diff --name-only master..bugfix # Shows the summary of changes with file names only

git diff --name-status master..bugfix # Shows the summary of changes with files names and their status

STASHING

Stashing means storing safely in a hidden or secret place. But it is not part of commits to that branch.

By default new untracked files are not included in your stash.

git stash push -m “New tax rules” # Creates a new stash

git stash push --all -m “New tax rules” # Creates a new stash which included untracked files also

git stash push -am “New tax rules” # same as above

git stash list # Lists all the stashes

git stash show stash@{1} # Shows the given stash

git stash show 1 # shortcut for stash@{1}

git stash apply 1 # Applies the given stash to the working dir

git stash drop 1 # Deletes the given stash

git stash clear # Deletes all the stashes

MERGING

1. Fast forward merges (if the branches have not diverged)
2. 3-way merges (if the branches have diverged)

git merge bugfix # Merges the bugfix branch into the current branch

git merge --ff-only bugfix # Creates a merge commit if Fast Forward is possible, otherwise don’t.

git merge --no-ff bugfix # Creates a merge commit even if Fast Forward is an option

git merge --squash bugfix # Performs a squash merge

git merge --abort # Aborts the merge (in case of too much conflicts, if we want to resolve the conflicts later. So for now, we can abort the merge and we will be back to original position)

DISABLE FAST FORWARD MERGES

git config ff no # Disables fast forward merging on current repo

git config --global ff no # Disables fast forward merging on all repositories for currently logged in user on this machine.

VIEWING THE MERGED BRANCHES

git branch --merged # Shows the merged branches into current branch/master

git branch --no-merged # Shows the unmerged branches

SETTING DEFAULT MERGE TOOL

git config --global merge.tool p4merge # Add merge tool placeholder

git config --global mergetool.p4merge.path="C:\Program Files\...p4merge" # Set mergetool path

git mergetool # Opens the conflicting items in the configured mergetool

git config --global mergetool.keepBackup false # to avoid creating backup file in case of conflicts.

UNDOING THE FAULTY MERGES

UNDOING THE FAULTY MERGES USING RESET

We should do 'reset' only if we have not shared our history with others (in other words only if we have not pushed our local changes)

Resetting to 1 step before current HEAD (assuming current HEAD is at the merged code)

git reset --hard HEAD~1 # Reset current HEAD to the specified state in the SNAPSHOT, Working dir and Staging area.

git reset --mixed HEAD~1 # Reset current HEAD to the specified state in the SNAPSHOT and Staging area. Working dir is not affected.

git reset --soft HEAD~1 # Reset current HEAD to the specified state in the SNAPSHOT. Working dir and Staging area are not affected.

UNDOING THE FAULTY MERGES USING REVERT

We should use this if we have shared our history with others (in other words if we have pushed our local changes to snapshot)

git revert HEAD # Reverts last commit

A Merged commit has 2 parents (one on its main branch (often master) and other one the feature/child branch (where we want to merge from).

git revert –m 1 HEAD # Reverts last commit. If current commit is a ‘Merged commit’, then we need to provide to which parent to revert to. 1 means parent on the master branch, 2 means parent on the feature branch.

SQUASH MERGING

Use it for small, short-lived branches with bad history, like bug fixes or feature branches which you are can built in few ours or a day. It avoids polluting history of our master branch.
The process is -
1. Undo the merge (if the short-lived branch is already merged)
2. Create a commit that combines all the changes in the bug fix branch, so this commit now is single logical change set which represents all the changes for fixing this bug.)
3. Apply the changes to master. That’s it!
Please note that this new commit is a regular commit and not a ‘Merge Commit’ as it will NOT have 2 parents.
4. Now we can delete the bug fix branch (shot-lived).

git switch master
git merge --squash bugfix/photo-upload # Squash merge the branch to master

git branch –D bugfix/photo-upload # Force delete the short-lived branch

REBASING

Rebasing rewrites your history. You should use rebasing only for branches where your commits are only in your local repository. You should not use rebasing if your commits are pushed or shared with other teammates, it will create a whole mess. With rebasing, we can reply a bunch of commits on top of another commit (when rebasing an earlier commit).

git rebase master # Changes the base of the current branch to (last commit of) master

After this, switch to master and do git merge (which will be fast forward merge)

In case of conflicts in git merge, we need to resolve the conflicts and commit the new changes (resolved conflicts).

git rebase --continue # To continue rebasing after resolving current conflict

git rebase --skip # To skip current conflict

git rebase --abort # To abort the rebase if there are many conflicts and we decide to do it later. This will take us to the previous state before we started rebasing.

CHERRY PICKING

Suppose we have few commits in feature branch and we want to merge one of the commits to master but we are not yet ready to merge the whole feature branch. In this case, we can cherry pick that particular commit from feature branch and apply it to master.

git cherry-pick dad47ed # Applies the given commit on the current branch/master

PICKING A FILE FROM ANOTHER BRANCH

git restore --source=feature/send-email toc.txt # Get toc.txt file from feature/send-email branch to current branch/master
git restore --source=feature/send-email -- toc.txt # same as above

COLLABORATION

Centeralized Worklow Vs Integration Manager Workflow

CLONING A REPOSITORY

git clone <url> # Clones the repository with the same name

git clone <url> <custom-name> # Clones the repository with the given custom name.

With git clone, all the commits on the central repository also exists on our local. When we clone a repository, git names the source repository (the one that we have on github) as **origin**. So origin is a reference to that repository.

origin/master tells where is the master branch in that remote repository. Tehnically it's called 'Remote Tracking Branch', this we cannot checkout, cannot commit to it.

origin/HEAD tells where is the HEAD pointer in the 'origin' repository.

A remote repository is a repository that is not on our machine, more accurately which is not in our current directory.
git remote # This shows the list of remote repositories

git remote -v # This shows the list of remote repositories with details

SYNCING WITH REMOTES

With git **fetch** command, git downloads new commits from the Remote repository to our local repository and forwards origin/master pointer to latest downloaded commit. Even though we downloaded this commit, our working directory is not updated. To update our working directory, we have to switch to 'master' and perform git merge origin/master.

git fetch origin master # Fetches master from origin

git fetch origin # Fetches all objects from origin

git fetch # Shortcut for "git fetch origin"

git branch -vv # Shows how our local and remote branches are diverging

git pull # pull = fetch + merge

git pull --rebase # Fetch + Rebase

With git **push** command, git will push our local commits to remote repository, then it will move master pointer forward to this new commit in the source repository (github) and finally it will move origin/master to latest commit on local repository.

git push origin master # Pushes master to origin

git push # Shortcut for “git push origin master”

STORING CREDENTIALS

When we do a git push, we have to provide credentials of the repository (github). It would be tedious to do it every time we push, better idea is to store these credentials so that we don’t have to every time enter it manually

git config --global credential.helper cache # git will store our credentials for 15 mins in our memory

To permanently store the credentials, we need to use Windows Credentials Store on Windows and keychain on Mac.

SHARING TAGS

git push origin v1.0 # Pushes tag v1.0 to origin

git push origin --delete v1.0 # To delete a tag from origin and push the changes to source (github)

SHARING BRANCHES

git branch -r # Shows remote tracking branches

git branch -vv # Shows local & remote tracking branches

git push -u origin bugfix # Pushes bugfix branch to origin (-u is short for --set-upstream which maps local branch to a branch in remote repository)

git push -d origin bugfix # Removes bugfix branch from origin (github) as well as removes its Remote branch Tracking branch reference on local repository.

MANAGING REMOTES

git remote # Shows remote repos

git remote add upstream url # Adds a new remote called upstream

git remote rename upstream base # Renames remote repo upstream to base

git remote rm upstream # Removes upstream

git remote prune origin # Removes Remote tracking branches references on local that are not in remote repository (already deleted on source repo (github)).

PULL REQUESTS

With a pull request, we open a discussion with the team to get feedback our implemented code before merging the code into a branch or master.

GITHUB MILESTONES

We use milestones to track the progress of various issues. So we can add bunch of issues to a milestone and then see the progress of the milestone.

REWRITING HISTORY

Git allows us to rewrite history. We can drop or modify commits, we can combine and split them and so on.

BENEFITS OF REWRITING (READABLE, MEANINGFUL) HISTORY

1. If we have small related commits, we can squash then into single commit that represents a logical change set.
2. If we have a large commit that contains lot of unrelated changes, we should split that commit into a bunch of small commits, each representing a logically separate change set.
3. We can reword commit messages so that they are meaningful to read.
4. If we make commits by accident, we can drop them.
5. We can also modify the contents of commits. E.g. if we forgot to add a file to a commit, we can go back and modify that commit to include that file as well.

GOLDEN RULE OF REWRITING HISTORY

Don't rewrite 'public' (already pushed and shared with others) history. We can and we should rewrite our 'private' history (local committed changes which are not pushed or shared with others yet).

UNDOING COMMITS

reset --soft HEAD^ # Removes the last commit, keeps changed staged

git reset --mixed HEAD^ # Unstages the changes as well

git reset --hard HEAD^ # Discards local changes

REVERTING COMMITS

git revert 72856ea # Reverts the given commit

git revert HEAD~3..HEAD # Reverts the commits between HEAD and HEAD~3, excluding the HEAD~3 commit. Basically HEAD, HEAD~1 and HEAD~2 will be reverted.

git revert HEAD~3.. # Same as above. (reverts **last** 3 commits.)

git revert --no-commit HEAD~3..HEAD # With this, git will figure out the changes to be undone and will apply those changes (between HEAD and HEAD~3(exclusive)) in the staging area, but will not commit it.
git revert --abort # If we want to abort this revert. Used along with --no-commit, as with no-commit, the changes are in staging area only.

git revert --continue # If we are happy and wants to continue this revert. Used along with --no-commit, as with no-commit, the changes are in staging area only.

RECOVERING LOST COMMITS

git reflog # Shows the history of HEAD

git reflog show bugfix # Shows the history of bugfix pointer

We can then use git reset --hard to reset our HEAD to any specific commit using commit id or unique reflog id (e.g. HEAD@{1}, etc.)

AMENDING THE LAST COMMIT

To update a file or add a new file to existing last commit –
git commit --amend # Make required amendments and commit using --amend, this will create a new commit and replace existing one.

Remember – Commits in GIT are immutable (neither content in that nor meta data of that commit like parent of that commit). So amending a commit means creating a new commit and replacing another.

To remove one of the files from last commit –

git reset --mixed HEAD~1 # Reset to last – 1 commit, resets snapshot and staging area, but not local directory.

git clean -fd # removes untracked files from local

git add

git commit -m “message”

INTERACTIVE REBASING (AMMENDING AN EARLIER COMMIT)

For interactive rebasing, we must choose the parent commit of the commit which we want to amend (update, drop, reword, etc.).
We get lots of predefined options to amend chain of commits with the interactive rebasing like pick, edit, drop, reword, squash, break, reset, etc.

TO AMEND AN EARLIER COMMIT

Even if we ‘edit’ one commit from the chain, all next commits in the chain will be amended, to be correct – all those next commits will be recreated and added on top of the ‘edited’ commit. This is because commits in git are immutable, we cannot change parent of a commit, we have to recreate a commit in order to change/amend it. And because of this, then change made to one the commits will be carried out with next chain of commits.

So remember rebasing is a destructive operation, it rewrites history, so we should only use it if we have not shared or pushed the commits with others.

git rebase -i 8527033 # Rebase to this commit and replay further commits on top of this commit.

git rebase --abort # To abort the rebasing
git rebase --continue # To continue rebasing, if we are satisfied with the rebasing

TO DROP A COMMIT

Let’s say we want to drop commit 2b6157b
git rebase -i 2b6157b^ # Rebase to parent of 2b6157b

Choose ‘drop’ option for the commit you want to drop and save. Now if a new file was added in that ‘to be dropped’ commit, then we will get a conflict. In this case, we have to resolve the conflicts.

git mergetool # To resolve the conflicts, choose ‘m’ for modify.

TO REWORD A COMMIT

Let’s say we want to just change the commit message (reword a commit message)

git rebase -i 2b6157b^ # Rebase to parent of 2b6157b

Choose ‘reword’ option for the commits you want to reword and save. Now git will reply all the commits in the chain and will open our default editor for the commits for which we chose ‘reword’ option. When the editor opens, update the commit message and save and close. That’s it.

TO RE-ORDER COMMITS

Let’s say we want to reorder commits, i.e. we mistakenly committed a perquisite code change (commit-2) after another commit (commit-1) which actually was dependent on commit-2. And now we want to place commit-2 before commit-1 for a meaningful history.

git rebase -i 70ef834 # Rebase to 70ef834

Now you will see code editor with all the existing commits one below other ordered from oldest to newest commits.
To reorder a commit, just move that particular line (starting with ‘pick…’) to your desired position. Save and close. That’s it.

TO SQUASH RELATED COMMITS INTO ONE COMMIT

Let's say we want to squash multiple related commits into a single commit,

git rebase -i d976e6d^ # Rebase to parent of d976e6d

Now you will see code editor with all the existing commits one below other ordered from oldest to newest commits.
To squash multiple commits into one, write 'squash' in front of the commit which you want to squash into the commit one above it. (If you have scattered commits, you can reorder those and use 'squash' option). Save and close. Then another editor window will open for the commit message. It will by default show all the existing commit messages, which you can edit and save and close. That's it!

```
pick d976e6d Render restaurants on the map.
squash 7ebb79a Fix a typo.
squash 86a79b4 Change the color of restaurant icons.
pick c2328ee Update terms of service and Google Map SDK version v1.0
pick 86f2ca6 Render cafes on the map
```

Note: There is another option similar to 'squash' which is 'fixup'. 'squash' and 'fixup' are exactly same except when you use fixup, git won't ask for a new commit message, it will use the existing commit message of the commit, in which 'fixup' commits will be squashed into.

```
pick d976e6d Render restaurants on the map.
fixup 7ebb79a Fix a typo.
fixup 86a79b4 Change the color of restaurant icons.
pick c2328ee Update terms of service and Google Map SDK version v1.0
pick 86f2ca6 Render cafes on the map
```

TO SPLIT A COMMIT INTO MULTIPLE COMMITS REPRESENTNG SINGLE LOGICAL UNIT OF WORK

Let's say we want to split a commit into multiple commits, each representing a single logical unit of work,

```
git rebase -i d976e6d^ # Rebase to parent of d976e6d
```

Now you will see code editor with all the existing commits one below other ordered from oldest to newest commits.
Add 'edit' option in front of the commit, which you want to split into. Save and close.
Now back on the git bash, git will stop at the position of 'edit' commit, points HEAD to this commit and gives us chance to do necessary updates.

Now we should 'soft' reset to current HEAD-1. With 'soft' option, our changes will be in the staging area. So we can unstage some of the changes and make another commit.

```
git reset --soft HEAD^ # Soft reset to current HEAD – 1.
```

Alternatively we can also use 'mixed' reset. With 'mixed' reset, our changes will be unstaged and will be present only in the working directory. Then we can stage and commit them separately.

```
git reset --mixed HEAD^ # Mixed reset to current HEAD – 1.

git status -s # See current status
```

Commit changes separately as per our need. And once done, continue rebasing.

```
git rebase --continue
```