Git Commands

```
$ git config --global user.name “Your Name”
$ git config ----global user.email “you@example.com”
$ git config --global color.ui auto     ##  Enable some colorization of Git output.
$ git config --global core.excludesfile [file]  #  System-wide ignore pattern for all local repos (like .gitignore)
$ git config --global –edit  # edit config file in editor
$ git init [project name]     ##  if no name, create a new repo, is initialized in the current directory.
$ git clone ssh://user@domain.com/repo.git  <OR> ~/existing/repo ~/new/repo

$ git add [file]   ## Add a file to the staging area.
$ git rm [file]   ## Remove file from working directory and staging area.
$ git commit  -m [message]  (use -a for all)  ## Create a new commit from changes added to the staging area
$ git checkout [file]    ## Discard local changes in a specific file (replace with remote commit)
$ git checkout $id $file   ## Checkout the $id version of a file

$ git branch [branch_name]    ## Create new branch, referencing the current HEAD.
$ git branch -d [branch_name]     ## Remove branch, if it is already merged into any other. -D forces
$ git checkout [-b][branch_name]    ## Switch working dir to branch; -b: create branch if it does not exist.
$ git checkout -b $new_branch $other   ## Create $new_branch based on branch $other and switch to it
$ git merge [branch_name]    ## Join specified remote branch into your current HEAD
$ git checkout $branch2; git merge branch1    ## Merge branch1 into branch2
$ git checkout --track [remote/branch]    ## Create new tracking branch based on a remote branch

$ git fetch [remote]  ## Fetch changes from remote, don‘t merge into HEAD or tracking branches
$ git pull [remote]     ## Fetch changes from the remote and merge current branch with its upstream.
$ git fetch --prune [remote]    ## Delete remote refs that were removed from the remote repo.
$ git pull --rebase [remote]    ## Same as above, but uses git rebase instead of merge

$ git push [remote] [branch]    ## Publish local commits to a remote. Use --tags to push tags
$ git push -u [remote] [branch]     ## Push local branch to remote repo. Set its copy as an upstream
$ git push --force (careful!) --all (means all branches), --tags (means all tags, aren’t normally pushed)
$ git remote add [remote] [url]      ## Add new remote repository, named [remote]
$ git branch -dr [remote/branch]     ## Delete a branch on the remote
$ git mergetool    ## Use your configured merge tool to solve conflicts
$ git commit –amend  ## Replace the last commit with combined staged changes and last commit. Use with
nothing staged to edit the last commit’s message. Don’t amend published commits.
```

Getting Information

```
$ git remote -v   ## List all currently configured remotes
$ git remote show [remote]    ## Show information about a remote
$ git branch [-a]    ## List branches. A * notes the currently active branch; -a: show all incl. remote).
$ git show [SHA]   ## Show any object in Git in human-readable format
$ git show $id:$file    ## A specific file from a specific $ID
$ git blame $file   ## Who changed what and when in a file

$ git diff [file]   ##  Changes between working directory and staging area.
$ git diff --staged [file]    ## Diff of what is staged but not yet commited
$ git diff HEAD   ## Show difference between working directory and last commit.
$ git diff --cached   ## Show difference between staged changes and last commit
$ git diff $id1 $id2   ##  What changed between $ID1 and $ID2
$ git log --follow [file]    ## how the commits that changed file, even across renames
$ git log [-n count]    ## List commit history of current branch
$ git log --oneline --graph --decorate    ## Overview with reference labels and history graph
$ git log refA..refB       ## Show commits on between branchA and branchB (ref can be branch, tag, etc)
$ git log --follow [file]      ## Show the commits that changed file, even across renames
$ git log –author= "<pattern>"  ## Search for commits by a particular author.
$ git log --grep= "<pattern>"  ##  Search for commits with a commit message that matches
$ git log -p $file $dir/ec/tory/   ## History of changes for file with diffs
$ git log --stat -M   ## Show all commit logs with indication of any paths that moved
$ git reflog    ## List operations (e.g. checkouts or commits) made on local repo.  --relative-date to show date
info, --all to show all refs
```

Reset: a "rollback"—it points your local environment back to a previous commit. Nothing on remote.

```
$ git reset [commit, tag]    ## Reset HEAD pointer to previous commit, preserve all changes as unstaged
$ git reset [commit] --hard HEAD    ## Discard all local changes in your working directory after commit
$ git reset --keep [commit]  ## If there’s a diff between <commit> and HEAD (local changes) this aborts reset
```

Reset moves the branch pointer back to "undo" and remove commits afterward with little explanation for Git and your programming team (bad!). Revert adds a new commit at the end of the chain to "cancel" changes.

```
$ git revert [commit]    ## Create new commit that undoes all of the changes made in [commit]
$ git revert HEAD  (or commit ID)   ## Revert the last commit
```

Rebase takes differing commits in branches and attempts to "replay" or mix into the other branch – changing commit history on the master branch. Can be destructive! If something breaks , run a git reset to undo it.

```
$ git rebase [branch]    ## Apply commits in branch ahead of remote branch. Don‘t rebase published commits!
$ git rebase --abort  (or) – continue     ## Abort or continue a rebase
$ git rebase -i     ##  Interactively rebase current branch onto <base> specify how to handle each commit

$ git stash   ##  Put current changes in your working directory into stash for later use.
$ git stash list   ##  List stack-order of stashed file changes
$ git stash pop   ##  Apply stored stash content into working directory, and clear stash.
$ git stash drop   ##  Delete a specific stash from all your previous stashes.
$ git clean -n    ## Shows files clean would remove from working directory. The -f flag executes the clean.
$ git tag -a [name] [commit sha]    ## Create a tag object “name” for current commit (often version number)
$ git tag -d [name]     ## Remove a tag from local repo
$ git tag     ## List all tags.
```

<u>Example scenario:</u>

```
          To view the merge conflicts
git diff --base $file      ## (against base file)
git diff --ours $file      ## (against your changes)
git diff --theirs $file    ## (against other changes)
          To discard conflicting patch
$ git reset --hard
$ git rebase --skip
          After resolving conflicts, merge with
$ git add $conflicting_file   ## do for all resolved files
$ git rebase –continue
```

Finding regressions

```
git bisect start   ## to start
git bisect good $id   ## $id is the last working version
git bisect bad $id   ## $id is a broken version
git bisect bad/good   ##  To mark it as bad or good
git bisect visualize   ##  Once you're done
git bisect reset   ##  To launch gitk and mark it

     git fsck             ## Check for errors
     git gc –prune       ## Cleanup repository
     git grep "foo()"  ## Search working directory
```

```
$ git am -3 patch.mbox    ## Apply a patch that some sent you. If a conflict, resolve and use git am --resolved
$ git format-patch origin   ## Prepare a patch for other developers
```

Examples: Helpful command aliases *(posted on Medium in 2016 and is reposted around the net now)*

Git Please $ git config --global alias.please 'push --force-with-lease'

Rebasing, amending, and squashing can rewrite some shared history and spill duplicate commits all over your repo. Force stomps the upstream branch with your local version, and any changes that you hadn't already fetched are erased from history.

Git's --force-with-lease checks that your local copy of the ref that you're overwriting is up-to-date first; that you've at least fetched the changes you're about to stomp.

============================
Git Commend $ git config --global alias.commend 'commit --amend --no-edit'

Ever commit and then immediately realize you'd forgotten to stage a file? Git commend quietly tacks any staged files onto the last commit you created, re-using your existing commit message. So as long as you haven't pushed yet, no-one will be the wiser. Don't amend published commits.

```
$ git add Dockerfile
$ git commit -m ‘Update Bitbucket pipeline with new Docker image’
   # (oops!)
$ git add bitbucket-pipelines.yml
$ git commend
```

============================
Git It $ git config --global alias.it '!git init && git commit -m “root” –allow-empty'

The first commit of a repo can not be rebased like regular commits, so it's good practice to create an empty commit as your repo root. git it both initializes and creates an empty root commit in one quick step.

```
$ cd shiny-new-thing
$ git it
Initialized empty Git repo in /shiny-new-thing/.git/
[master (root-commit) efc9119] root
```

============================
Git Staaash

```
$ git config --global alias.stsh 'stash --keep-index'
$ git config --global alias.staash 'stash --include-untracked'
$ git config --global alias.staaash 'stash --all'
```

Takes any changes to tracked files in your work tree and stashes them away for later use, leaving you with a clean work tree to start hacking on something else. However if you've created any new files and haven't yet staged them, git stash won't touch them by default, leaving you with a dirty work tree. Use staash Similarly, the contents of untracked or ignored files are not stashed by default. Use staaash. If in doubt, the long one (git staaash) will always restore your worktree to what looks like a fresh clone of your repo.

```
git stsh      # stash only unstaged changes to tracked files
git stash     # stash any changes to tracked files
git staash    # stash untracked and tracked files
git staaash   # stash ignored, untracked, and tracked files
```

Git Shorty $ git config --global alias.shorty 'status --short --branch'

Git's inline help has gotten a lot more friendly over the years, which is excellent for beginners, but the output is overly verbose for those more familiar with Git. For example, git status emits 18 lines to tell me that I have a couple of staged, unstaged, and untracked changes. Git shorty tells me the same thing in three lines:

```
$ git shorty
## master
AM test
?? .gitignore
```

============================
Git Merc $ git config --global alias.merc 'merge --no-ff'

On standard non-rebasing branching workflows running a standard git merge to combine feature branches with the master is actually not ideal. With no options, git merge uses the --ff merge strategy, which will create a merge commit only if there are no new changes on the master branch, otherwise it simply “fast forwards” your master branch to point at the latest commit on your feature branch. Without a merge commit it's tricky to tell which code was developed on which branches in the git history. The --no-ff strategy, to always create a merge commit.

============================
Git Grog (or “graphical log”)

```
$ git config --global alias.grog 'log --graph --abbrev-commit --decorate --all --format=format:"%C(bold blue)%h%C(reset) - %C(bold cyan)%aD%C(dim white) - %an%C(reset) %C(bold green)(%ar)%C(reset)%C(bold yellow)%d%C(reset)%n %C(white)%s%C(reset)"'
```

master : default development branch
origin : default upstream repository
local environment: local repository, staging area, and working directory.
HEAD : current branch
HEAD^ : parent of HEAD
HEAD~4 : the great-great grandparent of HEAD
https://www.golinuxcloud.com/git-workflow/

https://git-scm.com/docs/merge-strategies
https://www.atlassian.com/git/tutorials/using-branches/merge-strategy
https://www.atlassian.com/git/tutorials/using-branches/git-merge

Git Status Ouput - https://git-scm.com/docs/git-status

There are three different types of states that are shown using this format, and each one uses the XY syntax differently:

When a merge is occurring and the merge was successful, or outside of a merge situation, X shows the status of the index and Y shows the status of the working tree.

When a merge conflict has occurred and has not yet been resolved, X and Y show the state introduced by each head of the merge, relative to the common ancestor. These paths are said to be unmerged.

When a path is untracked, X and Y are always the same, since they are unknown to the index. ?? is used for untracked paths. Ignored files are not listed unless --ignored is used; if it is, ignored files are indicated by !!.

Note that the term merge here also includes rebases using the default --merge strategy, cherry-picks, and anything else using the merge machinery. In the following table, these three classes are shown in separate sections, and these characters are used for X and Y fields for the first two sections that show tracked paths:

' ' = unmodified
M = modified
T = file type changed (regular file, symbolic link or submodule)
A = added
D = deleted
R = renamed
C = copied (if config option status.renames is set to "copies")
U = updated but unmerged

X	Y	Meaning
	[AMD]	not updated
M	[MTD]	updated in index
T	[MTD]	type changed in index
A	[MTD]	added to index
D		deleted from index
R	[MTD]	renamed in index
C	[MTD]	copied in index
[MTARC]		index and work tree matches
[MTARC]	M	work tree changed since index
[MTARC]	T	type changed in work tree since index
[MTARC]	D	deleted in work tree
	R	renamed in work tree

X	Y	Meaning
	C	copied in work tree
D	D	unmerged, both deleted
A	U	unmerged, added by us
U	D	unmerged, deleted by them
U	A	unmerged, added by them
D	U	unmerged, deleted by us
A	A	unmerged, both added
U	U	unmerged, both modified
?	?	untracked
!	!	ignored