**Git**

Git is a distributed version control system, very easy to learn and supper fast!

**Install Git**

There are a few different ways to install git (from source or for Linux) but the purpose of this page is to focus on git commands, so I am going to assume you are installing git on a Mac.

To view other ways of installing it visit the [Git official site](http://git-scm.com/book/en/Getting-Started-Installing-Git)

Click [here](http://git-scm.com/download/mac) to download and install Git

**Setting up git**

$ git config --global user.name "User Name"

$ git config --global user.email "email"

**Applying colour to git**

$ git config --global color.ui true

**Initializing a repository in an existing directory**

If you’re starting to track an existing project in Git, you need to go to the project’s directory and type:

$ git init

This creates a new subdirectory named .git that contains all of your necessary repository files — a Git repository skeleton. At this point, nothing in your project is tracked yet.

To start version-controlling existing files you should start by tracking those files and do an initial commit. To accomplish that you should start with a few $ git add that specifies the files you want to track followed by a commit.

$ git add <file>

$ git add README

$ git commit -m 'Initial project version'

**Checking the status of your files**

The main tool you use to determine which files are in which state is the $ git status command. If you run this command directly after a clone, you should see something like this:

$ git status

# On branch master

nothing to commit (working directory clean)

If you add a new file to your project, and the file didn't exist before, when you run a $ git status you should see your untracked file like this:

$ git status

# On branch master

# Untracked files:

# (use "git add <file>..." to include in what will be committed)

#

# README

nothing added to commit but untracked files present (use "git add" to track)

**Staging files**

After initializing a git repository in the chosen directory, all files will now be tracked. Any changes made to any file will be shown after a $ git status as changes not staged for commit.

To stage changes for commit you need to add the file(s) - or in other words, stage file(s).

# Adding a file

$ git add filename

# Adding all files

$ git add -A

# Adding all files changes in a directory

$ git add .

# Choosing what changes to add (this will got through all your changes and you can 'Y' or 'N' the changes)

$ git add -p

**Stashing files**

Git stash is a very useful command, where git will 'hide' the changes on a dirty directory - but no worries you can re-apply them later. The command will save your local changes away and revert the working directory to match the HEAD commit.

# Stash local changes

$ git stash

# Stash local changes with a custom message

$ git stash save "this is your custom message"

# Re-apply the changes you saved in your latest stash

$ git stash apply

# Re-apply the changes you saved in a given stash number

$ git stash apply stash@{stash\_number}

# Drops any stash by its number

$ git stash drop stash@{0}

# Apply the stash and then immediately drop it from your stack

$ git stash pop

# 'Release' a particular stash from your list of stashes

$ git stash pop stash@{stash\_number}

# List all stashes

$ git stash list

# Show the latest stash changes

$ git stash show

# See diff details of a given stash number

$ git diff stash@{0}

**Committing files**

After adding/staging a file, the next step is to commit staged file(s)

# Commit staged file(s)

$ git commit -m 'commit message'

# Add file and commit

$ git commit filename -m 'commit message'

# Add file and commit staged file

$ git commit -am 'insert commit message'

# Amending a commit

$ git commit --amend 'new commit message' or no message to maintain previous message

# Squashing commits together

$ git rebase -i

This will give you an interface on your core editor:

# 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 log message

# x, exec = run command (the rest of the line) using shell

# Squashing commits together using reset --soft

$ git reset --soft HEAD~number\_of\_commits

$ git commit

\*\* WARNING: this will require force pushing commits, which is OK if this is on a branch before you push to master or create a Pull Request.

**Branching and merging**

# Creating a local branch

$ git checkout -b branchname

# Switching between 2 branches (in fact, this would work on terminal as well to switch between 2 directories - $ cd -)

$ git checkout -

# Pushing local branch to remote

$ git push -u origin branchname

# Deleting a local branch - this won't let you delete a branch that hasn't been merged yet

$ git branch -d branchname

# Deleting a local branch - this WILL delete a branch even if it hasn't been merged yet!

$ git branch -D branchname

# Remove any remote refs you have locally that have been removed from your remote (you can substitute <origin> to any remote branch)

$ git remote prune origin

# Viewing all branches, including local and remote branches

$ git branch -a

# Viewing all branches that have been merged into your current branch, including local and remote

$ git branch -a --merged

# Viewing all branches that haven't been merged into your current branch, including local and remote

$ git branch -a --no-merged

# Viewing local branches

$ git branch

# Viewing remote branches

$ git branch -r

# Rebase master branch into local branch

$ git rebase origin/master

# Pushing local branch after rebasing master into local branch

$ git push origin +branchname

**Fetching and checking out remote branches**

# This will fetch all the remote branches for you.

$ git fetch origin

# With the remote branches in hand, you now need to check out the branch you are interested in, giving you a local working copy:

$ git checkout -b test origin/test

# Deleting a remote branch

$ git branch -rd origin/branchname

$ git push origin --delete branchname or $ git push origin:branchname

**Merging branch to trunk/master**

# First checkout trunk/master

$ git checkout trunk/master

# Now merge branch to trunk/master

$ git merge branchname

# To cancel a merge

$ git merge --abort

**Updating a local repository with changes from a Github repository**

$ git pull origin master

**Tracking existing branch**

$ git branch --set-upstream-to=origin/foo foo

**Resetting**

# Mixes your head with a give sha

# This lets you do things like split a commit

$ git reset --mixed [sha]

# Upstream master

$ git reset HEAD origin/master -- filename

# The version from the most recent commit

$ git reset HEAD -- filename

# The version before the most recent commit

$ git reset HEAD^ -- filename

# Move head to specific commit

$ git reset --hard sha

# Reset the staging area and the working directory to match the most recent commit. In addition to unstaging changes, the --hard flag tells Git to overwrite all changes in the working directory, too.

$ git reset --hard

**Git remote**

# Show where 'origin' is pointing to and also tracked branches

$ git remote show origin

# Show where 'origin' is pointing to

$ git remote -v

# Change the 'origin' remote's URL

$ git remote set-url origin https://github.com/user/repo.git

# Add a new 'origin'

# Usually use to 'rebase' from forks

$ git remote add [NAME] https://github.com/user/fork-repo.git

**Git grep**

# 'Searches' for parts of strings in a directory

$ git grep 'something'

# 'Searches' for parts of strings in a directory and the -n prints out the line numbers where git has found matches

$ git grep -n 'something'

# 'Searches' for parts of string in a context (some lines before and some after the grepped term)

$ git grep -C<number of lines> 'something'

# 'Searches' for parts of string and also shows lines BEFORE the grepped term

$ git grep -B<number of lines> 'something'

# 'Searches' for parts of string and also shows lines AFTER the grepped term

$ git grep -A<number of lines> 'something'

**Git blame**

# Show alteration history of a file with the name of the author

$ git blame [filename]

# Show alteration history of a file with the name of the author && SHA

$ git blame [filename] -l

**Git log**

# Show a list of all commits in a repository. This command shows everything about a commit, such as commit ID, author, date and commit message.

$ git log

# List of commits showing commit messages and changes

$ git log -p

# List of commits with the particular expression you are looking for

$ git log -S 'something'

# List of commits by author

$ git log --author 'Author Name'

# Show a list of commits in a repository in a more summarised way. This shows a shorter version of the commit ID and the commit message.

$ git log --oneline

# Show a list of commits in a repository since yesterday

$ git log --since=yesterday

# Shows log by author and searching for specific term inside the commit message

$ git log --grep "term" --author "name"

**Checking what you are committing**

# See all (non-staged) changes done to a local repo

$ git diff

# See all (staged) changes done to a local repo

$ git diff --cached

# Check what the changes between the files you've committed and the live repo

$ git diff --stat origin/master

**Useful commands**

# Check if a sha is in production

$ git tag --contains [sha]

# Number of commits by author

$ git shortlog -s --author 'Author Name'

# List of authors and commits to a repository sorted alphabetically

$ git shortlog -s -n

# List of commit comments by author

$ git shortlog -n --author 'Author Name'

# This also shows the total number of commits by the author

# Number of commits by contributors

$ git shortlog -s -n

# Undo local changes to a File

$ git checkout -- filename

# Shows more detailed info about a commit

$ git cat-file sha -p

# Show number of lines added and removed from a repository by an author since some time in the past.

$ git log --author="Author name" --pretty=tformat: --numstat --since=month | awk '{ add += $1; subs += $2; loc += $1 - $2 } END { printf "added lines: %s, removed lines: %s, total lines: %s\n", add, subs, loc }'

**Useful alias**

To add an alias simply open your .gitconfig file on your home directory and include the alias code

# Shows the log in a more consisted way with the graph for branching and merging

lg = log --color --graph --pretty=format:'%Cred%h%Creset -%C(yellow)%d%Creset %s %Cgreen(%cr) %C(bold blue)<%an>%Creset' --abbrev-commit

**Contributing**

1. Fork it!
2. Create your feature branch: git checkout -b my-new-feature
3. Commit your changes: git commit -m 'Add some feature'
4. Push to the branch: git push -u origin my-new-feature
5. Submit a pull request - cheers!