Microsoft Student Partners

Sir Syed University of Engineering & Technology
Society

GitHub Ultimate Program

Git Commands Manual

Microsoft

GitHub

GitHub is a Git repository hosting service, but it adds many of its own features. While Git is a command line tool, **GitHub** provides a Web-based graphical interface. It also provides access control and several collaboration features, such as a wikis and basic task management tools for every project.

Git

Git is a distributed version-control system for tracking changes in source code during software development. It is designed for coordinating work among programmers, but it can be used to track changes in any set of files. Its goals include speed, data integrity, and support for distributed, non-linear workflows.

Famous Linux Commands

1. pwd

pwd stands for Print Work directory and does exactly what you think – it shows the directory you're currently in. This is one of the handiest Linux terminal commands that aims to make new user's life peaceful by ensuring they don't get lost in that seemingly cryptic terminal window.

2. ls

The ls command is probably one of the most widely used commands in the Unix world. It presents to you the contents of a particular directory – both files and directories. You will use this command alongside pwd to navigate your ways inside the mighty Unix filesystem.

3. cd

Short for Change Directory, the cd command is behind your movement from one directory to another. It's one of the few Linux commands that you're *bound* to use throughout your stint with the Linux system. This command makes life in front of the terminal less scary for beginners while providing a standard method to browse the entire filesystem of your device.

4. mkdir

Want to create a new folder through the terminal? The mkdir command is created for just this specific purpose. It lets you create folders anywhere you like in your Linux system – given you have got the necessary permission, of course!

5. rmdir

The archrival of the mkdir command, the rmdir command allows you to delete specific folders from your system without any hassles. Although many utilize the rm command for this purpose, screwing up parameters or even a single character with rm can do things you wouldn't even dream. So, stick with rmdir for now.

6. lsblk

Often you will find the need to list the available block devices of your Linux system. The lsblk is one of the most used Linux commands for this purpose. This handy terminal command will present you with a tree structure of your block devices and is used heavily by professional users.

7. mount

Contrary to Windows, whenever you plug in an SD card or a USB, chances are your distro won't show them directly at the start. You need to mount it with your existing filesystem using the mount command. This Linux command is one of the most powerful terminal commands out there.

8. touch

The touch command is an essential Linux command for creating a valid empty file. You can create files on the go in your terminal and fill them up later or in real-time – based on your requirements. It's also the go-to command for changing the timestamps.

9. mv

Short for a move, it's a supplement to the cut operation you perform in the GUI. Just like cp, you can use the mv command to move either single or multiple files from one location to another. You can force this Linux command to transfer large files with the -f parameter.

10. sudo

The sudo command is the holy grail of Linux commands. It lets non-privileged users access and modify files that require low-level permissions. Often you will use this command to access root from your regular user account.

Setting Up Git

Make sure you have already installed Git and GitHub Desktop in your computer. Following the below commands, you have to setup your profile.

```
git config --global user.name "<Your-Full-Name>"
git config --global user.email "<Your-EMAIL-Address>"
```

Make sure that Git console's output is in colored.

```
git config --global color.ui auto
```

Displays the original state in a conflict.

```
git config --global merge.conflictstyle diff3
git config –list
```

In the project directory make first commit and push

```
git init
git remote add origin <https://github.com/........>
git status
git add .
git config --global user.email "<Your-EMAIL-Address>"
git config --global user.name "<Your-Full-Name>"
git commit -m "Initial commit"
git push origin master or git push origin master –f
```

For changing remote origin location / URL

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

For cloning any repository

```
git clone <path-of-repository-to-clone>
```

For checking status

```
git status
```

For displaying information of commits

```
git log
git log ls
git log ls –L
git log –oneline
```

For seeing the number of lines that have been added or deleted

```
git log --stat
```

This below command has a -p flag that can be used to display the actual changes made to a file; how many lines added and how many removed

```
git log --patch
```

```
git log -p
```
-p stands for patch, they both are same
```
git log -p –stat              # same output
git log -p fdf5493            # display only info of that commit
```

The git show command will show only one commit
```
git show
```
It will display the commit, the author, the date, the commit message & the patch information

For staging file for tracking we use
```
git add <file>
git add .          #contains all file
```

The following command is not like the shell's rm command.
```
git rm --cached
```
This will not destroy any of your work it just removes it from the Staging Index.

This is used to configure code editor that you are going to use. Hint: Use notepad
```
git config --global core.editor <your-editor's-config-went-here>
```

To make a commit in Git you use the git commit command
```
git commit -m "Initial commit"
```
or
```
git commit -m "Version 1.0"
```
or any version you are working on.

The following command can be used to see changes that have been made but haven't been committed, yet.
```
git diff
```

Now create a .gitignore file to ignore those files which u don’t want to track.
```
touch .gitignore
```
In the gitignore file you can write "samples/*.jpg" this will ignore all the jpg files

For tag verification, we use
```
git tag
git tag -a v1.0
```
-a provides the detail such as who made the tag plus time and message for tag
```
git tag -d v1.0
```
#A Git tag can be deleted with the -d flag (for delete!) and the name of the tag

When viewing information of commits, the --decorate flag will show us some details that are hidden from the default view.
```
git log --decorate
```

Branching & Merging

Branching is the practice of creating copies of programs or objects in development to work in parallel versions, retaining the original and working on the **branch** or making different changes to each.

Merging is **Git's** way of putting a forked history back together again. The **git merge** command lets you take the independent lines of development created by **git** branch and integrate them into a single branch. Note that all of the commands presented below **merge** into the current branch.

For listing all branch names in the repository

```
git branch
```

Creates a branch

```
git branch sidebar
```

Remember that when a commit is made that it will be added to the current branch. So even though we created the new sidebar, no new commits will be added to it since we haven't switched to it, yet. If we made a commit right now, that commit would be added to the master branch, not the sidebar branch. We've already seen this, but to switch between branches, we need to use Git's #checkout command.

```
git log --oneline --decorate
git branch alt-sidebar-loc 42a69f
```

Branch will be created on this commit

One thing to note is that you can't delete a branch that you're currently on. So, to delete the sidebar branch, you'd have to switch to either the master branch or create and switch to a new branch this deletes the branch.

And for the force deletion we use

```
git branch -D sidebar
```

Did you know that the git checkout command can actually create a new branch, too? If you provide the -b flag, you can create a branch and switch to it all in one command. Let's use this new feature of the git checkout command to create our new footer branch and have this footer branch starts at the same location as the master branch:

```
git checkout sidebar
```

switches to sidebar branch

```
git checkout -b richards-branch-for-awesome-changes
```

For resetting the changes to last commit

```
git reset --hard HEAD^
```

The merges we just did were able to merge successfully. Git is able to intelligently combine lots of work on different branches. However, there are times when it can't combine branches together. When a merge is performed and fails, that is called a merge conflict.
As you've learned, Git tracks lines in files. A merge conflict will happen when the exact same line(s) are changed in separate branches. For example, if you're on a alternate-sidebar-style branch and change the sidebar's heading to "About Me" but then on a different branch and change the sidebar's heading to "Information About Me", which heading should Git choose? You've changed the heading on both branches, so there's no way Git will know which one you actually want to keep. And it sure isn't going to just randomly pick for you! Remember that a merge conflict occurs when Git isn't sure which line(s) you want to use from the branches that are being merged. So, we need to edit the same line on two different branches...and then try to merge them.

```
git merge <name-of-branch-to-merge-in>
git merge footer
git merge sidebar
```

After the conflict
```
git status
```

Merge Conflict Indicators Explanation

The editor has the following merge conflict indicators:

<<<<<<< HEAD everything below this line (until the next indicator) shows you what's on the current branch
||||||| merged common ancestors everything below this line (until the next indicator) shows you what the original lines were
======= is the end of the original lines, everything that follows (until the next indicator) is what's on the branch that's being merged in
>>>>>>> heading-update is the ending indicator of what's on the branch that's being merged in (in this case, the heading-update branch)

The following command will update the most recent commit, if you forget to add.
```
git commit --amend
```

For undoing the last changings in the most recent commit
```
git revert <SHA-of-commit-to-revert>
```

To erase the commit, you made.
```
git reset <reference-to-commit>
```

Before doing any resetting, you should create a backup branch on the most – recent commit so that you could get back to the commits if I make a mistake
```
git branch backup
```

Using the sample repo above with HEAD pointing to master on commit 9ec05ca, running

```
git reset --mixed HEAD^
```

will take the changes made in recent commit and move them to the working directory.
Running git reset --soft HEAD^ will take the changes made in commit 9ec05ca and move them directly to the Staging Index.

```
git reset --soft HEAD^
```

And when you need to permanently remove it, hit following command

```
git reset --hard HEAD^
```

A **remote** in **Git** is a common repository that all team members use to exchange their changes. In most cases, such a **remote** repository is stored on a code hosting service like GitHub or on an internal server. In contrast to a local repository, a **remote** typically does not provide a file tree of the project's current state.

```
git remote
git remote -v
```

Give path of cloned repository

```
git remote add origin https://github.com/HelloWorld.git
git clone https://github.com/owen/my-travel-plans.git
```

```
git log --oneline --graph --decorate --all
git push <remote-shortname> <branch>
```

The local commits end at commit 5a010d1 while the remote has two extra commits - commit 4b81b2a and commit b847434. Also, notice that in our local repository when we did the git log the origin/master branch is still pointing to commit 5a010d1.
Remember that the origin/master branch is not a live mapping of where the remote's master branch is located. If the remote's master moves, the local origin/master branch stays the same. To update this branch, we need to sync the two together. If you don't want to automatically merge the local branch with the tracking branch, then you wouldn't use git pull you would use a different command called git fetch. You might want to #do this if there are commits on the repository that you don't have but there are also commits #on the local repository that the remote one doesn't have either.

```
git pull origin master
```

When git fetch is run, the following things happen:

- The commit(s) on the remote branch are copied to the local repository
- The local tracking branch (e.g. origin/master) is moved to point to the most recent commit
- The important thing to note is that the local branch does not change at all.
- You can think of git fetch as half of a git pull. The other half of git pull is the merging aspect.

```
git fetch origin master
```

One main point when you want to use git fetch rather than git pull is if your remote branch and your local branch both have changes that neither of the other ones has. In this case, you want to #fetch the remote changes to get them in your local branch and then perform a merge #manually. Then you can push that new merge commit back to the remote
This is very important command for seeing the changes in head and all the commits and links between remote and local branches in the repository world forking is the split but an identical split

```
git log --oneline --graph --decorate –all
```

```
git clone <repository url>
git push origin include-richards-destintation
```

This is not a massive project, but it does have well over 1,000 commits. A quick way that we can see how many commits each contributor has added to the repository is to use the git shortlog command

```
git shortlog
```

git shortlog displays an alphabetical list of names and the commit messages that go along with them. If we just want to see just the number of commits that each developer has made, we can add a couple of flags: -s to show just the number of commits (rather than each commit's message) and -n to sort them numerically (rather than alphabetically by author name).

```
git log --author=AuthorName
```

#Filter By Author

For merging remote branch with local branch after fetching

```
git merge origin/master
```

The following is very imp command for seeing the changes in head and all the commits and links between remote and local branches

```
git log --oneline --graph --decorate –all
```

To display how many people did how many commits in the project, we use the following command

For filtering every commit with <word> will be displayed by using the following command

```
git log --grep=<word>
```

For adding changes back to original source
A pull request is a request for the source repository to pull in your commits and merge them with their project. To create a pull request, a couple of things need to happen:

- you must fork the source repository
- clone your fork down to your machine
- make some commits (ideally on a topic branch!)
- push the commits back to your fork
- create a new pull request and choose the branch that has your new commits

```
git remote add upstream https://github.com/udacity/course-collaboration-travel-plans.git
```

Using the git remote rename command to rename origin to mine and upstream to source-repo.

```
git remote rename mine origin
git fetch upstream master
```

To make sure I'm on the correct branch for merging hit following command in the Git console

```
git checkout master
```

`git merge upstream` or `git merge master`

For sending someone's changes to *my* remote

```
git push origin master
```

When you have to merge ***n*** commits recently added, you have to use the following command.

```
git rebase -i HEAD~3
```

Here 3 is the value of ***n,*** or number of commits.

==========

Prepared by
Furqan Ali
AI Mechanic

Reviewed by
Muhammad Huzaifa Shahbaz
Microsoft Student Partner