Git and GitHub

Made By:
Aryan Khurana

INDEX

Git Cheatsheet at the end

↳ SOURCE: GITHUB

↳ Pg. 12-15

Introduction

Imagine that you have an application or a project folder and you added a new feature which results in the application not working. Through git and github, you can access your previous codebase whenever you want. You can also collaborate towards a project where thousands of other people are collaborating.

GIT → VERSION CONTROL SOFTWARE

1. Keeps track of changes to code.
2. Synchronizes code between different people.
3. Tests changes to code without losing the original code.
4. Reverts back to old versions of code.

GITHUB → A WEBSITE WHICH HOSTS GIT REPOSITORIES

Folder where code is stored ←

Downloading Git

Go to "git-scm.com" and install git.

Basic Linux Commands

1. pwd – When you first open the terminal, you are in the home directory of your user. To know which directory you are in, you can use the **"pwd"** command. It gives us the absolute path, which means the path that starts from the root. The root is the base of the Linux file system. It is denoted by a forward slash(/). The user directory is usually something like "/home/username".

```
nayso@Alok-Aspire:~$ pwd
/home/nayso
```

2. ls — Use the "**ls**" command to know what files are in the directory you are in. You can see all the hidden files by using the command **"ls -a"**.

```
nayso@Alok-Aspire:~$ ls
Desktop            itsuserguide.desktop  reset-settings     VCD_Copy
Documents          Music                 School_Resources   Videos
Downloads          Pictures              Students_Works_10
examples.desktop   Public                Templates
GplatesProject     Qgis Projects         TuxPaint-Pictures
```

3. cd — Use the "**cd**" command to go to a directory. For example, if you are in the home folder, and you want to go to the downloads folder, then you can type in **"cd Downloads"**. Remember, this command is case sensitive, and you have to type in the name of the folder exactly as it is. But there is a problem with these commands. Imagine you have a folder named "Raspberry Pi". In this case, when you type in **"cd Raspberry Pi"**, the shell will take the second argument of the command as a different one, so you will get an error saying that the directory does not exist. Here, you can use a backward slash. That is, you can use **"cd Raspberry\ Pi"** in this case. Spaces are denoted like this: If you just type **"cd"** and press enter, it takes you to the home directory. To go back from a folder to the folder before that, you can type "**cd** .." . The two dots represent back.

```
nayso@Alok-Aspire:~$ cd Downloads
nayso@Alok-Aspire:~/Downloads$ cd
nayso@Alok-Aspire:~$ cd Raspberry\ Pi
nayso@Alok-Aspire:~/Raspberry Pi$ cd ..
nayso@Alok-Aspire:~$
```

4. mkdir & rmdir — Use the **mkdir** command when you need to create a folder or a directory. For example, if you want to make a directory called "DIY", then you can type **"mkdir DIY"**. Remember, as told before, if you want to create a directory named "DIY Hacking", then you can type "mkdir **DIY\ Hacking**". Use **rmdir** to delete a directory. But **rmdir** can only be used to delete an empty directory. To delete a directory containing files, use **rm**.

```
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$ mkdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
DIY
nayso@Alok-Aspire:~/Desktop$ rmdir DIY
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

5. rm - Use the **rm** command to delete files and directories. Use "**rm -r**" to delete just the directory. It deletes both the folder and the files it contains when using only the **rm** command.

```
nayso@Alok-Aspire:~/Desktop$ ls
newer.py  New Folder
nayso@Alok-Aspire:~/Desktop$ rm newer.py
nayso@Alok-Aspire:~/Desktop$ ls
New Folder
nayso@Alok-Aspire:~/Desktop$ rm -r New\ Folder
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$
```

6. touch – The **touch** command is used to create a file. It can be anything, from an empty txt file to an empty zip file. For example, "**touch new.txt**".

```
nayso@Alok-Aspire:~/Desktop$ ls
nayso@Alok-Aspire:~/Desktop$ touch new.txt
nayso@Alok-Aspire:~/Desktop$ ls
new.txt
```

7. man & --help – To know more about a command and how to use it, use the **man** command. It shows the manual pages of the command. For example, "**man cd**" shows the manual pages of the **cd** command. Typing in the command name and the argument helps it show which ways the command can be used (e.g., **cd –help**).

```
TOUCH(1)                        User Commands                        TOUCH(1)

NAME
       touch - change file timestamps

SYNOPSIS
       touch [OPTION]... FILE...

DESCRIPTION
       Update  the  access  and modification times of each FILE to the current
       time.

       A FILE argument that does not exist is created empty, unless -c  or  -h
       is supplied.

       A  FILE  argument  string of - is handled specially and causes touch to
       change the times of the file associated with standard output.

       Mandatory arguments to long options are  mandatory  for  short  options
       too.

       -a     change only the access time

Manual page touch(1) line 1 (press h for help or q to quit)
```

8. cp — Use the **cp** command to copy files through the command line. It takes two arguments: The first is the location of the file to be copied, the second is where to copy.

```
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ cp new.txt /home/nayso/Music/
nayso@Alok-Aspire:~/Desktop$ ls /home/nayso/Music/
new.txt
```

9. mv — Use the **mv** command to move files through the command line. We can also use the **mv** command to rename a file. For example, if we want to rename the file "**text**" to "**new**", we can use "**mv text new**". It takes the two arguments, just like the **cp** command.

```
nayso@Alok-Aspire:~/Desktop$ ls
new.txt
nayso@Alok-Aspire:~/Desktop$ mv new.txt newer.txt
nayso@Alok-Aspire:~/Desktop$ ls
newer.txt
```

10. locate — The **locate** command is used to locate a file in a Linux system, just like the search command in Windows. This command is useful when you don't know where a file is saved or the actual name of the file. Using the -i argument with the command helps to ignore the case (it doesn't matter if it is uppercase or lowercase). So, if you want a file that has the word "hello", it gives the list of all the files in your Linux system containing the word "hello" when you type in "**locate -i hello**". If you remember two words, you can separate them using an asterisk (*). For example, to locate a file containing the words "hello" and "this", you can use the command "**locate -i *hello*this".**

```
nayso@Alok-Aspire:~$ locate newer.txt
/home/nayso/Desktop/newer.txt
nayso@Alok-Aspire:~$ locate *DIY*Hacking*
/home/nayso/DIY Hacking
```

Basic Git Commands

- **git init** → Initialises a git repository

This command creates a repository in the project folder where all the project history is stored. The folder is named ".git". This folder is hidden and can be accessed by "ls -a" command.

- **git status** → Provides information about changes which are currently not in the history of your project.
- **git add** → Used to stage files which are untracked in order to store them in the history.

1. git add . ⇒ Used to stage all the files
2. git add file.txt ⇒ Used to stage specific files

- **git commit** → Used to store changes permanently in the history.

You can use this command in the following way:

git commit -m "Names.txt file added"

↳ Used to add messages to the history.

- **git restore** → Used to unstage files that are already tracked.

To use this command: git restore --staged names.txt

- **git log** → Used to view the history.

History consists of all the changes that were made. The name, date and time are also tracked and stored in the history.

- **git reset** → Allows the user to access previous commits and reset latest ones.

Copy the commit below all the commits you want to remove.

Usage: git reset <commit hash>

The commits which are removed from the log will turn into unstaged commits.

- **git stash** → Used to stash unstaged changes

After using this command the working tree in the "git status" will be cleared and the unstaged changes will be stashed.

Stashing → If you do not want to delete the unstaged commits, you can stash them.

- git stash pop → Brings the stashed changes back to the working tree.
- git stash clear → Clears and deletes all the changes forever.

Using Git and GitHub

To start off, we need to create a new repository (Folder) in GitHub. After creation, we get a URL. We want this URL to be attached to the local project. In order to do so, we use the following command.

git remote add origin https://github.com/aryan-khurana/proj.git

- remote means you are working with URLs.
- add means you are adding a URL.
- origin is the name of the URL by convention

→ git remote -v → Shows all the URLs that are attached to the folder.

Now, if we go back to GitHub, we still cannot see the changes because we have not shared the changes on the URL. In order to do so we use:

- **git push** → Allows the user to push the changes to a URL and a branch

To use this command, we have to specify the name of the URL (origin) and the branch (master) → git push origin master

You can also clone an existing GitHub repository on your local computer.

- **git clone** → Takes a repository located somewhere on github servers and copies it to your computer.

Use: git clone <url>

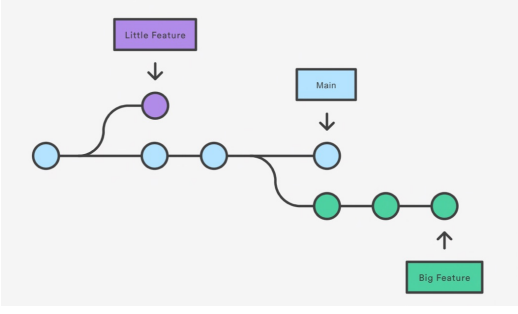
- **git pull** → Takes changes from your server and downloads them on your computer.

Git Branching

Introduction

Branching is a feature available in most modern version control systems. Branching in other VCS's can be an expensive operation in both time and disk space. In Git, branches are a part of your everyday development process. Git branches are effectively a pointer to a snapshot of your changes. When you want to add a new feature or fix a bug—no matter how big or how small—you spawn a new branch to encapsulate your changes. This makes it harder for unstable code to get merged into the main code base, and it gives you the chance to clean up your future's history before merging it into the main branch.

The diagram here visualizes a repository with two isolated lines of development, one for a little feature, and one for a longer-running feature. By developing them in branches, it's not only possible to work on both of them in parallel, but it also keeps the main branch free from questionable code.



Commands

- **git branch** → Creates a new branch

This command can be used in two ways:

1. git branch ➔ Displays all the branches you have in your project and highlights the current one.
2. git branch <branchname> ➔ Creates a new branch but does not make it the current one.

- **git checkout** → Used to switch branches

This command can be used in two ways:

1. git checkout -b <branchname> ➔ Creates a new branch and makes it the current one.
2. git checkout <branch name> ➔ Used to switch branches.

- **git merge** → Used to merge branches to the main branch

Usage: git merge <branch name>

↳ You can only use this command once you checkout to the main branch.

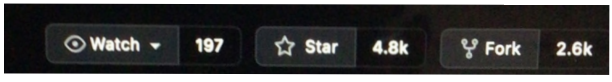
Forking and Pull Requests

Q.) What is forking?

A) Forking is a git clone operation executed on a server copy of a projects repo. You basically create a new feature branch in your local repo. Work is done to complete the new feature and git commit is executed to save the new changes. You can then push the new feature branch to your remote forked repo.

Q.) What is the point of forking?

A) Most commonly, forks are either used to propose changes to someone else's project or to use someone else's project as a starting point for your own idea. You can fork a repository to create a copy of the repository and make changes without affecting the upstream repository. [the original repo]
You can fork a repository by clicking the "fork" option in the top right corner of any repo.



Q.) What is a pull request?

A.) A pull request is a method of submitting contributions to a project. A pull request occurs when a developer asks for changes committed to an external repository to be considered for inclusion in a projects main repo.

Note: You cannot push your changes/features to the upstream as you do not have access to it, but you can push changes to the origin.

Note: If you want to remove a commit from the pull request, you have to force push: `git push origin <branchname> -f`

Note: If the upstream is updated but the fork is not, you can resolve this in the following three ways.

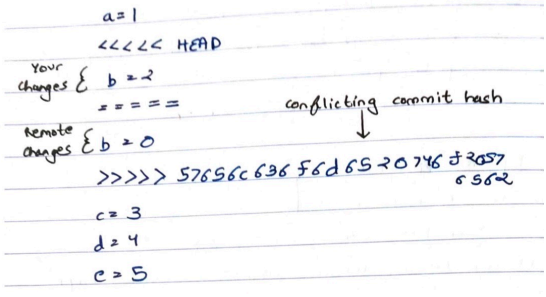
1. Click on fetch upstream option in your fork.
2. Use a combination of the following commands.
   - → `git checkout main`
   - → `git fetch --all --prune` ← To get all the branches / To fetch the deleted branches
   - → `git reset --hard upstream/main`
   - → `git push origin main`
3. Pull the changes
   - → `git pull upstream main`

Intermediate Concepts

Merge Conflicts

When working in Git, users can combine commits from two different branches through an action known as merging. Files are automatically merged unless there are conflicting sets of changes (i.e. the commits update the same line of code differently). A merge conflict is an event that occurs when Git is unable to automatically resolve differences in code between two commits. When all the changes in the code occur on different lines or in different files, Git will successfully merge commits without your help. However, when there are conflicting changes on the same lines, a "merge conflict" occurs because Git doesn't know which code to keep and which to discard.

The file, on being opened:



To resolve a merge conflict, remove the markers. Decide how the file should look. Push the changes. All of this is done manually.

Squashing commits

If you have a lot of commits, you can merge them into one commit.

Usage: `git rebase -i <commit hash below all the commits to be squashed>`

- After using this command you can either pick or squash the commits. Replace "pick" with "s" to do so.
- All the commits that are squashed will be merged with the previous picked commit.
- Press esc + colon (:) + x to add a message for the merged commits.

P.T.O for the cheatsheet provided by GitHub

Git & GitHub 101

Basic CLI Commands

1. To list all files or folder in a folder

   `ls`

2. Make a new folder

   `mkdir folder_name`

3. Go inside a folder

   `cd folder_name`

4. To delete a whole non-empty directory/folder

   `rm directory_name -rf`

5. Write a file in Git Bash Vim

   `vim file_name`

   1. use insert key to enable the writing mode in any file
   2. then after finishing edits, press the left-right arrow key to disable the writing mode and then write `:x` to exit out

6. Copy + Paste in CLI

   1. Use the insert key to paste in CLI or highlight the statement then right click and copy that statemen and then right-click on CLI shows the options.

Basic Git Commands

1. To make a new file

   `touch names.txt`

2. To check if git is installed in your PC

   `git`

3. To initialize an empty Git repository in your folder

   `git init`

4. To view the changes or the untracked files in the project that's not been saved yet

   `git status`

5. Staging the files

   `git add file_name` or `git add .` (to stage everything in the current folder)

6. Committing the files

Working with Existing Projects on GitHub

Use Git Bash for Windows.

You can't directly change the contents of a repo unless you have access to it. To solve this, you create a copy (fork) of this project in your own account. In our own copy, we can do anything we want with it. After forking, we:

1. Cloning the forked project to local machine

   `git clone forked_repo_url`

2. The public repo that we forked out local copy from is known as the upstream url. We can save it as

   `git remote add upstream insert_upstream_url`

3. Creating a new branch

   `git branch branch_name`

4. Then shift the head to the above branch using the checkout command

5. Then stage. Then commit.

6. Then push. We can't push to upstream (no access). Can push to our forked repo though (origin)

   `git push origin your_branch_name`

7. Always make different branches for different pull requests if you're working on different features. 1 branch = 1 pull request (never commit on main (2))

8. To remove a commit

   1. we can remove a commit with the reset command Now it's unstaged.
   2. then add. to stage the remaining files
   3. then we can use the stash command to stash it elsewhere
   4. then, we'll have to force push this branch since the online repo contains a commit which the local repo does not

   `git push origin your_branch_name -f`

9. To make forked project even (updated) with the main project

```
git commit -m "your_message_here"
```

7. To unstage or remove a file from the staging level

   `git restore --staged file_name.txt`

8. To view the entire history of the project

   `git log`

9. Removing a commit from the history of a project

   `git reset insert_commit_hash_id_to_which_you_want_to_go_back_to_here` (all the commits or changes before this will go back to the unstaged area now)

10. After you stage a few files but then you want to have a clean codebase or reuse those files later, we can stash those changes to go back to the commit before they were staged

    `git stash`

11. Bringing back those changes or pop them from the stash

    `git stash pop`

12. To clear the changes or files in your stash

    `git stash clear`

How Git works

1. Connecting your Remote Repository to Local Repository

   `git remote add origin insert_https_project_link_here`

2. Pushing local changes to remote repository

   `git push origin master` (we're pushing to the url origin, and the branch master)

3. To view all your remote urls

   `git remote -v`

4. Never commit on the main branch since it's the one used by the people, to prevent any mishaps

5. Shifting the head to a branch (head is the pointer which points to where all you do your changes)

   `git checkout branch_name`

6. Merging your branch to main of project

   `git merge branch_name`

1. Shift the head to your main branch

   `git checkout main`

2. Fetching all the commits/changes from the main project (upstream)

   `git fetch --all --prune` (here prune gets deleted commits too)

3. Reset the main branch of origin (forked) to main branch of upstream (main project)

   `git reset --hard upstream/main`

4. Check and verify your changes

   `git log` *click q for exit from log*

5. Then push all these local changes to your online forked repo

   `git push origin main`

Method 2

1. To fetch all **at once**

   `git pull upstream main`

2. Then push to the origin url or your forked project

   `git push origin main`

Method 3

1. Update using the Fetch Upsteam button on forked repo

10. Squashing all your multiple commits into one commit

    `git rebase -i insert_hash_code_of_commit_above_which_all_your_required_co`

    If there's 4 commits. Keep 1 as the pick and then `s` or squash the other 3 into that one

11. Merge conflicts and how to resolve them

    1. They happen when multiple users edit the same code line and then push it. Git won't know which one to merge and then there'd be a conflict
    2. This has to be resolved manually by repo maintainer

https://s3-us-west-2.amazonaws.com/secure.notion-static.com/94dcc4e7-259a-4c09-bf05-d1ce5eb2d9e3/atlassian-git-cheatsheet.pdf

GitHub
GIT CHEAT SHEET

Git is the free and open source distributed version control system that's responsible for everything GitHub related that happens locally on your computer. This cheat sheet features the most important and commonly used Git commands for easy reference.

INSTALLATION & GUIS

With platform specific installers for Git, GitHub also provides the ease of staying up-to-date with the latest releases of the command line tool while providing a graphical user interface for day-to-day interaction, review, and repository synchronization.

GitHub for Windows
https://windows.github.com

GitHub for Mac
https://mac.github.com

For Linux and Solaris platforms, the latest release is available on the official Git web site.

Git for All Platforms
http://git-scm.com

SETUP

Configuring user information used across all local repositories

`git config --global user.name "[firstname lastname]"`
set a name that is identifiable for credit when review version history
`git config --global user.email "[valid-email]"`
set an email address that will be associated with each history marker
`git config --global color.ui auto`
set automatic command line coloring for Git for easy reviewing

SETUP & INIT

Configuring user information, initializing and cloning repositories

`git init`
initialize an existing directory as a Git repository
`git clone [url]`
retrieve an entire repository from a hosted location via URL

STAGE & SNAPSHOT

Working with snapshots and the Git staging area

`git status`
show modified files in working directory, staged for your next commit
`git add [file]`
add a file as it looks now to your next commit (stage)
`git reset [file]`
unstage a file while retaining the changes in working directory
`git diff`
diff of what is changed but not staged
`git diff --staged`
diff of what is staged but not yet committed
`git commit -m "[descriptive message]"`
commit your staged content as a new commit snapshot

BRANCH & MERGE

Isolating work in branches, changing context, and integrating changes

`git branch`
list your branches. a * will appear next to the currently active branch
`git branch [branch-name]`
create a new branch at the current commit
`git checkout`
switch to another branch and check it out into your working directory
`git merge [branch]`
merge the specified branch's history into the current one
`git log`
show all commits in the current branch's history

INSPECT & COMPARE

Examining logs, diffs and object information

`git log`
show the commit history for the currently active branch
`git log branchB..branchA`
show the commits on branchA that are not on branchB
`git log --follow [file]`
show the commits that changed file, even across renames
`git diff branchB...branchA`
show the diff of what is in branchA that is not in branchB
`git show [SHA]`
show any object in Git in human-readable format

SHARE & UPDATE

Retrieving updates from another repository and updating local repos

`git remote add [alias] [url]`
add a git URL as an alias
`git fetch [alias]`
fetch down all the branches from that Git remote
`git merge [alias]/[branch]`
merge a remote branch into your current branch to bring it up to date
`git push [alias] [branch]`
Transmit local branch commits to the remote repository branch
`git pull`
fetch and merge any commits from the tracking remote branch

TRACKING PATH CHANGES

Versioning file removes and path changes

`git rm [file]`
delete the file from project and stage the removal for commit
`git mv [existing-path] [new-path]`
change an existing file path and stage the move
`git log --stat -M`
show all commit logs with indication of any paths that moved

REWRITE HISTORY

Rewriting branches, updating commits and clearing history

`git rebase [branch]`
apply any commits of current branch ahead of specified one
`git reset --hard [commit]`
clear staging area, rewrite working tree from specified commit

IGNORING PATTERNS

Preventing unintentional staging or commiting of files

```
logs/
*.notes
pattern*/
```

Save a file with desired patterns as .gitignore with either direct string matches or wildcard globs.

`git config --global core.excludesfile [file]`
system wide ignore pattern for all local repositories

TEMPORARY COMMITS

Temporarily store modified, tracked files in order to change branches

`git stash`
Save modified and staged changes
`git stash list`
list stack-order of stashed file changes
`git stash pop`
write working from top of stash stack
`git stash drop`
discard the changes from top of stash stack