Basic Git Commands PromptSheet

- **git version** : displays the installed Git version on the system; helps you verify if Git is installed.
- **git init** : initializes/creates a new Git repository. Make sure you are not already inside of a Git repository when you initialize one.
- **git status** : gives information on the current status of a Git repository.
- **git config --global user.name *your_name*** : sets (if you provide a value after *user.name*) or displays (if you run it with no arguments) the global Git username.
- **git config --global user.email *your_email*** : sets (if you provide a value after *user.email*) or displays (if you run it with no arguments) the global Git email address.
- ***git add*** : stages the changes for committing --- tells git to include updates to specific files, folders, etc. in the upcoming commit.
  - **git add *filename(s)*** : stages specific files for committing.
  - **git add .** : stages all changes in the current directory for committing. It does not stage the removal of files.
  - **git add -A** or **git add -all** : stages all changes in the current directory including new files, and removals. It's a more inclusive command than **git add .**
- **git commit** : commits/records the changes that you have staged using ***git add*** with a commit message. If you directly run this command it opens your default text editor for you to write a commit message (flexible for multi-line messages).
  - **git commit -m "*commit_message*"** : commits changes in a concise way with a commit message directly from the command line.
  - **git commit --amend -m "*commit_message*>"** : modifies the last commit by allowing you to basically 'redo' the previous commit. It combines staged changes with the previous commit, effectively allowing you to add more changes or modify the commit message.
  - **git commit -a -m "*commit_message*"** : commits all changes, including modifications and deletions for already tracked files, without the need for explicit ***git add***. Note that it only skips the explicit **git add *modified_file(s)*** for modified and deleted files. It won't handle new, untracked files---you still need to use **git add** for that.
- **git log** : displays the commit history of a Git repository by showing a chronological list of commits, starting from the most recent and going backward in time.
  - **git log --oneline** : displays each commit in a condensed, one-line format providing a concise overview of the commit history.
- **git branch** : lists all the existing branches in the repository. The branch with an asterisk is the currently active branch (the one you are currently on).
  - **git branch -v** : lists all the existing branches in the repository alongside the last commit on each branch with its commit message and hash.
  - **git branch -r** : lists all the existing remote branches our local repository knows about.
  - **git branch *new_branch_name*** : creates a new branch based upon the current HEAD. This just creates the branch---it does not switch you to that branch (the HEAD stays the same).
  - **git switch *branch_name*/*remote_branch_name*** : switches to a specified existing branch in a Git repository by updating the HEAD pointer to point to the specified branch, effectively switching your working directory to the branch's latest commit. It is an alternative to the older **git checkout *branch_name*** command. However, **git switch *branch_name*** is a standalone switch command and much simpler because the latter does a million additional things. If you instead provide a remote branch name as argument, it will make a local branch of the same name and sets it up to track the remote branch then switches to the local branch it created. It then works as an alternative to the older **git checkout --track *remote_name*/*remote_branch_name*** command.

    - **git switch -c *new_branch_name*** : creates a new branch and immediately switches to it---a concise way to perform the two-step process (of creating and switching to a branch) in a single one. It is an equivalent to the older **git checkout -b *new_branch_name*** command.
    - **git switch -** : switches back to the previously active branch. It is an equivalent to **git checkout -**.
    - **git branch -d *branch_name*** or **git branch --delete *branch_name*** : deletes the specified branch if it is not currently checked-out. Note that if the branch has changes that are not yet merged into the current branch, Git will prevent you from deleting it using *-d* or *--delete*. In such cases, you might need to force delete the branch using *-D* or *--delete --force*. Also note that deleting a branch does not delete the commits; it only removes the branch reference.
    - **git branch -m *new branch_name*** or **git branch --move *new_branch_name*** : renames the currently checked-out branch to a new name. Note that if you attempt to rename a branch to the same name it currently has, Git won't stop you, and the command will go through. However, the operation is essentially a no-op (it will result in no changes). Also note that if you attempt to rename a branch to a new name that already exists (as the name of some other branch), Git will prevent you from renaming it using *-m* or *--move*. In such cases, you might need to force rename the branch using *-M* or *--move --force*. This will overwrite any existing branch with the same name and the commit history of the branch you overwrite becomes part of the commit history of the newly named branch as its branch reference is lost.
- **git merge *source_branch_name* -->** : integrates changes from the specified source branch into the currently checked-out (destination/target) branch. In Git, this command fundamentally performs two types of merging:-
    1. Fast-Forward Merge: This occurs when the branch being merged into has no new commits since the branch being merged in. When you run the **git merge *source_branch_name*** command, it moves the branch pointer forward of the branch being merged into (target branch) to point the same commit as the branch being merged in (the specified source branch) without creating an extra merge commit. It essentially "fast-forwards" the branch pointer of the target branch to catch up on the commits of the source branch. This results in a linear commit history on the target branch which appears as a direct continuation of the changes made on the source branch.
    2. Three-Way (Regular) Merge/Creation of a Merge Commit: This occurs when there are divergent changes on both branches and the branch pointer can't be simply "fast-forwarded" i.e. the target branch and the source branch have both seen new commits since they diverged. When you run the **git merge *source_branch_name*** command you might as well run it with the *-m* flag before mentioning the source branch name to enter a new commit message otherwise it will open your default text editor with a default message for you to do so: **git merge -m "*commit_message*" *source_branch_name***. This is because this type of merge involves creating a new merge commit that has two parent commits, indicating the point where the two branches were merged or integrated. One parent is the tip (latest commit) of the target branch and the other parent is that of the source branch. If there are conflicting changes (changes to the same lines of code in the same file, etc. on both branches), Git prompts you to resolve them manually, essentially leaving for you to decide which change will prevail over another.
- **git diff** : shows changes between different commits, branches, files, our working directory, and more. Without any additional options, the command compares the working directory and the staging area (index) and displays all the changes in our working directory that are not staged for the next commit. In other words, the differences are what you could tell Git to further add to the index but you still haven't.
    - **git diff HEAD** : compares the changes in your current working directory with the state of the repository at the latest commit (HEAD) and lists all changes made in the working tree since your last commit (HEAD). This includes both staged and unstaged changes.
    - **git diff --staged** or **git diff --cached** : compares the staging area (index) with your last commit (HEAD) and shows differences between them.
    - **git diff branch1..branch2** or **git diff branch1 branch2** : compares the tips of the two branches in the order specified and lists differences between them.
    - **git diff commit1..commit2** or **git diff commit1 commit2** : compares the two commits in the order specified and displays changes between them. Note that you provide commit hashes as arguments.

- **git diff tag1..tag2** or **git diff tag1 tag2** : compares the two tags in the order specified and displays differences between them.

  > If you want to use the above diff commands with specific file(s) only, you can provide its/their filename(s) or path(s) after them:-

  ```
  - **git diff _file1 file2 ..._ ** or
  - **git diff HEAD _file1 file2 ..._** or
  - **git diff --staged _file1 file2 ..._** or
  - **git diff branch1..branch2 _file1 file2 ..._** or
  - **git diff commit1..commit2 _file1 file2 ..._** or
  - **git diff tag1..tag2 _file1 file2 ..._**
  ```

- **git stash** : saves changes that you are not ready to commit yet. It essentially allows you to temporarily save your working directory and index allowing you to switch branches or perform other tasks without committing your changes. If you directly run this command or you run it with the *save* argument: **git stash save**, it will take all uncommitted changes (staged or unstaged) and stash/save them, reverting the working directory and index to the state of the last commit. You can also include a stash message to describe the changes you're stashing by writing it within quotation marks after the stash command with the save option: **git stash save "*stash_message*"**.
  - **git stash list** : displays a stash history of a Git repository by showing a chronological list of stashes, starting from the most recent and going backward in time.
  - **git stash apply** : applies the changes from the most recent stash to your working directory and index without removing the stash itself. This can be useful if you want to apply stashed changes to multiple branches.
  - **git stash drop** : drops or removes the most recent stash from the stash list permanently.
  - **git stash clear** : clears out the entire stash list permanently.
  - **git stash pop** : applies the most recent stash and removes it from the stash list. It's a combination of two different actions performed by two different commands:-
    1. Applying or restoring the most recent stash to your working directory and index. This action is performed by **git stash apply**.
    2. Dropping or removing the most recent or applied stash from the stash list. This action is typically performed by **git stash drop**.

  > The stash commands to apply, drop or pop stashes can be used with specific stashes only, too:-

  ```
  1. **git stash apply stash@{_stash_index_}** or
  2. **git stash drop stash@{_stash_index_}** or
  3. **git stash pop stash@{_stash_index_}**
  ```

- **git checkout** : a versatile or an overloaded command (depending on which side you prefer) used for various purposes:-
  - **git checkout *branch_name*** : checks-out/switches to a specified existing branch in a Git repository by updating the HEAD pointer to point to the specified branch, effectively switching your working directory to the branch's latest commit. It is an alternative to the newer **git switch *branch_name*** command.
  - **git checkout -b *new_branch_name*** : creates a new branch and immediately switches to it---a concise way to perform the two-step process (of creating and switching to a branch) in a single one. It is an equivalent to the newer **git switch -c *new_branch_name*** command.
  - **git checkout *remote_name/remote_branch_name*** : switches to the specified existing remote branch by putting you in a detached HEAD state.
  - **git checkout --track *remote_name/remote_branch_name*** : creates a local branch of the same name as the specified remote branch and sets it up to track the remote branch then switches to the local

branch created. This works as an alternative to the newer **git switch *remote_branch_name*** command.

- **git checkout *commit_hash*** : switches to the specified commit in your Git commit history, essentially by putting your repository in a "detached HEAD" state.
  A detached HEAD state means that you are no longer on a branch, but directly on the specified commit. You are essentially "detaching" the HEAD pointer from a branch reference (which always points at its tip i.e. last commit) to point it at some specific commit. In this detached HEAD state you can look around, make experimental changes and commit them, and you can discard any commits you make in this state. Any changes you make in this state won't be associated with any branch i.e. you will notice all of it hasn't impacted any of your branches when you switch back to them.
  If you finish examining the contents of your old commit and now if you want to revert to where you were before, simply switch back to the branch you were on to "re-attach" the HEAD pointer to point back at the branch reference. You can even branch off of the commit you went back to review to continue make and save changes from there on by creating and switching to a new branch. This also "re-attaches" your HEAD pointer, but now it points at your new branch's reference. And really you can switch to any branch in a detached HEAD state to "re-attach" it.
- **git checkout *HEAD~1 or HEAD~2 or ...*** : references previous commits relative to a particular commit. When you use *HEAD~1* you are instructing Git to move the HEAD pointer to the commit one step back in the commit history (parent of the commit HEAD is pointing at). Similarly, *HEAD~2* refers to two commits before HEAD (grandparent of the commit HEAD is pointing at) and so on. Note that switching to commits this way also puts your repository in a detached HEAD state.
- **git checkout -** : switches back to the previously active branch. It is an equivalent to **git switch -**.
- **git checkout HEAD *filename*** or **git checkout -- *filename*** : discards local changes or modifications made to a specific file in your working directory which you haven't yet committed and reverts the file to the version stored in the last commit (HEAD) or the file's previously committed version.
- **git checkout HEAD .** or **git checkout -- .** : discards uncommitted local changes in all files of your working directory and reverts them to their previously committed versions.
- **git checkout *tag_name*** : checks out code at a specific tag in your Git history. Note that checking out a tag results in a detached HEAD.

- **git restore *filename*** : discards uncommitted local changes in a specific file of your working directory and restores/reverts it back to its contents in the HEAD i.e. back to its previously committed version. It is an equivalent to the traditional **git checkout -- *filename*** or **git checkout HEAD *filename*** command.
  - **git restore --source *HEAD~1 or HEAD~2 or .../commit_hash filename*** : works alike **git restore** but the *--source* option is used to specify the source from which to restore changes. *HEAD~1* restores the specified file to the state of it at the commit one step back from HEAD i.e. one commit back. *HEAD~2* restores the file to its state two commits back and so on. Basically without the *source* option, the default source is just HEAD. You can also specify a commit hash instead if the commit you want to provide as the source is too far behind than where your HEAD currently is and you don't want to count those steps.
  - **git restore --staged *filename(s)*** : unstages changes for a specific file in the staging area (index). Note that the changes made on the file previously added to the index will be removed/unstaged from it but still retained in your working directory. It essentially is used to undo the action of ***git add*** on specific files in the index.
  - **git restore --staged .** : unstages changes for all files in the index. Note that the changes made on the files previously added to the index will be removed/unstaged from it but still retained in your working directory. It essentially is used to undo the action of ***git add*** on all files in the index.
- **git reset *HEAD~1 or HEAD~2 or .../commit_hash*** : resets the current branch to the specified commit or the commit relative to HEAD (if you use the HEAD~X syntax). If you run it directly without any options or with the default *--mixed* option, it only resets the staging area to the specified commit and keeps the changes in your working directory. Basically, it will clear your commit history after the commit whose commit hash you specified while keeping their changes in your working directory.
  - **git reset --hard *HEAD~1 or HEAD~2 or .../commit_hash*** : resets both the staging area and the working directory to match the commit specified or the commit relative to HEAD (if you use the HEAD~X syntax). It effectively discards all changes made after the specified commit in your current branch.

- **git revert *commit hash*** : creates a new commit that reverts/undoes the changes made in a previous commit. It's a safer way to undo commits compared to **git reset** command and its options because it doesn't modify the existing commit history but instead creates a new commit that reverses the changes. Because it results in a new commit, you will be prompted to enter a commit message.
- **git clone *repo_URL*** : creates a copy of/clones an existing remote Git repository on your local machine. Make sure you are not already inside of a Git repository when you clone one.
- **git remote** : manages connections to remote repositories (repository that is hosted on a server or another location outside of your local machine). If you run it directly without any options, it displays a list of the names of remotes (if added any).
  - **git remote -v** : works alike **git remote** ran directly i.e. lists added remotes but the *-v* option standing for "verbose" displays a list of the names of added remotes with more information---including the URL of each remote repository.
  - **git remote add *name/label_to_a_remote(conventionally, 'origin') remote_repo_URL*** : adds a new remote repository to your Git configuration.
  - **git remote rename *old_remote_name new_remote_name*** : renames an existing remote repository from its current name to its desired name.
  - **git remote remove *remote_name*** : removes an existing remote repository from your Git configuration.
- **git push *remote_name branch_name*** : sends/pushes changes from your local repository's specified branch to the remote repository's branch of the same name (if it doesn't exist, GitHub creates one).
  - **git push *remote_name local_branch_name*:*remote_branch_name*** : pushes changes from your local repository's specified branch to the remote repository's specified branch.
  - **git push -u *remote_name branch_name*** : works alike **git push *remote_name branch_name*** but the *-u* option allows you to set the upstream of the branch you're pushing. You can think of this as a link connecting our local branch to a remote branch. After you run this command, it sets up tracking between the specified local branch and the remote branch of the same name (if it doesn't exist, a new one is created) which allows subsequent pushes with just **git push** without specifying the remote and branch.
- **git fetch *remote_name*** : retrieves/fetches updates from a specific remote repository without integrating or merging them into your current branch. It does not modify your working directory or staging area, it only updates remote tracking branches. Note that if no remote name is specified and you run the command directly without any arguments, the remote name defaults to 'origin'. We can also fetch latest changes from a specific branch of a remote using the syntax: **git fetch *remote_name remote_branch_name***. After you've fetched updates from a remote or a remote branch, you'll have those changes on your machine, but if you want to see them you have to checkout the remote branch your local branch is behind from. Your local branch is untouched.
- **git pull *remote_name remote_branch_name*** : fetches and merges changes from a specific remote branch of the specific remote repository into your current branch. It immediately updates your local repository with the changes it retrieves. The command is equivalent to running **git fetch** to fetch changes to update the remote tracking branch with the latest changes from the remote and then **git merge** to integrate those changes into your current branch to update it with the changes fetched on the remote tracking branch. Note that the second action it performs (merging) can result in merge conflicts which you will be prompted to resolve. Also, if we simply run **git pull** without specifying a particular remote and remote branch to pull from, Git assumes the remote to be 'origin' by default and the remote branch to be the one with a tracking connection configured for your current branch.
- **git rebase *branch_name*** : incorporates changes from one branch into another by rewriting commit history making it linear. The command takes the changes from your current branch (the one you're rebasing) and applies them on top of the specified branch. When you rebase a branch, it is moved entirely so that it begins at the tip of the branch you're rebasing onto. All of the work and changes are still there, but the history is re-written. It can be used as an alternative to the **git merge *branch_name*** command to avoid clogging of the commit history by the merge commits. Since, instead of using merge commits, rebasing integrates changes by rewriting history by creating new commits for each of the original commits of the branch you're rebasing. Note that in case of the **git merge *branch_name*** command, the specified branch is merged into the

currently active branch, whereas, in case of the **git rebase *branch_name*** command, the currently active branch is rebased onto the specified branch. But the use case is the same, you switch to the branch you want to rebase and then run the command with the branch name you want your current branch to be rebased onto.

- **git rebase -i *HEAD~1 or HEAD~2 or ...*** : initiates an interactive rebase, allowing you to edit commits, add files, drop commits, etc. for the last specified number of commits i.e. you specify how far back you want to rewrite commits. Notice that you are not rebasing onto another branch. Instead, you are rebasing a series of commits onto the HEAD they currently are based on. When you run this command, Git will open an interactive text editor displaying a list of the last specified number of commits in your branch. Each commit is prefixed with the word "pick", indicating the default action, which is to keep the commit as is. Then you will have options to perform various operations on each commit:-

  > *pick* : keep the commit as it is.
  > *reword* : keep the commit but allow changing the commit message.
  > *edit* : Pause rebase to allow amending the commit (e.g., changing files, adding more changes).
  > *squash* or *fixup* : Combine the commit with the previous one. "squash" retains commit messages, while "fixup" discards them.
  > *drop* : remove the commit from history.

- **git tag** : prints a list of all the tags in the current repository if you run it directly or with the optional *-l* or *--list* option. Tags are basically pointers that refer to a particular point in Git history. You can think of tags as branch references that do not change. Once a tag is created, it always refers to the same commit like a label for it. Note that this command lists tags in alphabetical or numeric order, the order in which they are displayed has no real importance. You can also search for tags that match a particular pattern in case your Git repository contains a lot of tags. If you're interested only in looking at the tags that include "beta" in their name, you can run this: **git tag -l "*beta*"**---the asterisks in the pattern indicate "some character before" and "some character afterwards" respectively, you may or may not include them.
  - **git tag *new_tag_name*** : creates a lighweight tag at the latest commit i.e. referring to the commit that HEAD is referencing. Lightweight tags are basically just a name/label pointers to a specific commit in your Git history. If you want to tag a specific commit, you can provide the commit hash: **git tag *new_tag_name commit_hash***. Note that when you create a lightweight tag, it's local to your repository. To have the tag on the remote you have to run: **git push origin *tag_name*** explicitly.
  - **git tag -a *new_tag_name* -m "*tag_message*"** : creates an annotated tag at the latest commit i.e. referring to the commit that HEAD is referencing. Annotated tags are tags associated with additional information or extra meta data including the tagger's/author's name, tagger's/author's email, the date and time, and a tag message. Note that the *-m* option in this command is used to specify the tag message directly on the command line. If you run the command without it, Git will open your default text editor for you to enter the tag message. If you want to create an annotated tag at a specific commit, you can provide the commit hash: **git tag -a *new_tag_name* -m "*tag_message*" *commit_hash***. When you want to see the additional information employed by an annotated tag, you can run: **git show *commit_hash/tag_name***. Also note that when you create an annotated tag, it's local to your repository. To have the tag on the remote you have to run: **git push origin *tag_name*** explicitly.
  - **git tag -f *tag_name*** : updates the reference of an existing tag forcefully to the latest commit. This can be useful if you need to move a tag to point to a different commit. If you want to move the tag reference to a specific commit, you can provide a commit hash: **git tag -f *tag_name commit_hash***. Since, the changed reference of the tag on your local repository won't reflect on your remote automatically, you will have to run: **git push origin -f *tag_name*** explicitly.
  - **git tag -d *tag_name*** : deletes the specified tag from your Git history. Note that if you have already pushed the tag to a remote repository, the local deletion won't automatically propagate to the remote and even if you force push after you delete a tag locally, it won't reflect on the remote. You will need to use: **git push origin --delete *tag_name*** explicitly.

    > You can push all local tags to the remote repository at once by running **git push --tags** command.

- **git reflog** : displays the reference log (reflog) of the HEAD reference if you run it directly without any options. A reflog in Git is basically a record of changes to branch references, such as commits, branch creations, checkouts, and other operations. It helps you recover lost commits or branches and provides a history of recent actions. This command accepts subcommands *show, expire, delete* and *exits*. The commonly used variant and the default subcommand is *show*. If you simply run **git reflog** or run **git reflog show**, it will show the log of a specific reference. The reference defaults to HEAD, but if you want to view the logs for the tip of a branch you can run **git reflog show *branch_reference***. After viewing your reflogs, you can simply hard reset into any commits that seem lost and are not appearing in git log to retrieve your "lost work". Note that Git only keeps reflogs on your local activity, they are not shared with collaborators and reflogs do expire after around 90 days (the expiration time can be configured).

  > We can access specific references using the syntax: *name@{qualifier}* and can pass them to commands like *checkout, reset, diff, merge* including **git reflog show *reference***. This syntax comes in handy in case of **git reflog** command if you want to filter reflogs entries by time/date. You can do this by using qualifiers like:-
  > *name_of_ref@{1.day.ago}*
  > *name_of_ref@{2.days.ago}*
  > *name_of_ref@{3.minutes.ago}*
  > *name_of_ref@{yesterday}*
  > *name_of_ref@{'Sat Feb 24 00:11:40 2024 +0530'}*

- **git config --global alias.*name git_command*** : creates a custom alias of the specified name for the specified git command.