Git Cheat Sheet

(Beginner To Intermediate)

Starting A Repo

`git init`

Initializes a Git repo in the current directory

`git init <folder>`

Initializes a Git repo in `<folder>` in the current directory. Creates `<folder>` if it doesn't exist then initializes inside.

Information

`git ls-files`

Returns what files Git is tracking.

`git status`

Returns status of changes to files, as well as untracked files (new files).

`git log`

Returns all the commits that are part of the repo.

`git show`

Returns the last commit with a diff showing all the changes. *[Press Q to get out of the show command.]*

`git reflog`

Shows a more detailed history of the repo, including commits made, when you switched branches, resets, etc.

Staging And Committing

`git add <file>`

Adds `<file>` to the staging area.

`git add .`

Adds all files to the staging area. Includes subdirectories but not higher directories.

`git add -A`

Adds all files to the staging area.

`git add -u`

Adds modified and deleted files to the staging area, but not untracked files.

`git commit`

Launches the default editor to write a commit message. Can write a multi-line message. Save the file then exit to complete the commit. *[Ctrl + W = exit]*

`git commit -m "<commit message>"`

Commits files in staging area with a commit message without launching the default editor.

`git commit -am "<commit message>"`

Adds modified files to the staging area, and commits them with the commit message.

`git restore --staged <file>`

Removes `<file>` from the staging area.

`git restore <file>`

Undoes changes made to `<file>`, reverting back to the last known state of that file in the Git repo.

`git switch` *and* `git restore` *were added in Git version 2.23. They're meant to replace common uses of* `git checkout` *and to be more intuitive. You can still use* `git checkout` *as before.*

Renaming & Deleting Files

`git mv <file> <new name>`

Renames the file and stages the renaming.

`git rm <file>`

Deletes the file and stages the deletion.

The benefit of using Git to rename and delete files is Git will automatically recognize and stage them. If you rename a file outside of Git, Git will see the file as being deleted, along with an untracked file with the new name.

Undoing Things

`git commit --amend`

Opens interactive editor where you are able to change the commit message of the last commit and add the files in the staging area to the last commit. ***

`git commit --amend -m "<changed commit message>"`

Changes the commit message of the last commit. ***

`git reset --soft <commit id>`

Resets back to specified commit, keeping changes in the staging area. If commit isn't given, resets to last commit. ***

`git reset --mixed <commit id>`

Resets back to specified commit, keeping changes but removing them from the staging area. ***

`git reset --hard <commit id>`

Resets back to specified commit, getting rid of changes made, but keeps untracked files. ***

`Mixed` is the default if no flag is given. If `<commit id>` isn't given, reset will use the last commit.

*** *These commands alter Git history, so only use them if you have not yet pushed your changes to the remote repo you are working with.*

`git checkout <commit id>`
`git branch <new branch name>`

You can use this to undo resets. Use `git reflog` to find the `<commit id>`. This puts `HEAD` into a detached state on the specified commit, and creates a branch from that commit. Can't do this after Git garbage collector runs.

`git revert <commit id>`

Opens an interactive editor where you create a new commit that reverses the changes made by the specified commit. This doesn't alter Git history, so it can be used with other developers.

`git clean -f`

Removes untracked files in current directory.

`git clean -d`

Removes untracked subdirectories in current directory.

`git clean -x`

Removes untracked files in current directory, including ignored files in the .gitignore file.

`git clean -X`

Removes only untracked ignored files.

`git clean -n`

Shows what files would be removed if the clean command is ran. It is good practice to do this before you actually run clean.

Differences

`git diff`

Shows the difference between what's recently changed versus the last commit on the branch. (More specifically versus where HEAD is.)

`git difftool`

Launches the configured diff tool (e.g. P4Merge) and shows the difference between what's recently changed versus the last commit on the branch. (More specifically versus where HEAD is.)

`git diff <commit id A> <commit id B>`

Shows the differences between two commits (one of them can be HEAD).

`git difftool <commit id A> <commit id B>`

Launches the configured diff tool (e.g. P4Merge) and shows the differences between two commits (one of them can be HEAD).

Branching & Merging

`git branch`

Lists the branches and shows which one you're currently on.

`git branch -a`

Lists all branches, including remotes, and shows which one you're currently on.

`git branch <new branch name>`

Creates a branch.

`git switch <branch name>`

Switches to the specified branch.

`git switch -c <new branch name>`

Creates a new branch then switches to it.

`git merge <branch name>`

Merges the specified branch into the current branch.

`git mergetool`

While in a merging state, launches the configured merge tool (e.g. P4Merge)

Stashing

`git stash`

Stashes changes to tracked files. Does not stash untracked files or ignored files. (Saves the last commit id and message as a default note.)

`git stash save "<note>"`

Stashes changes to tracked files with a note about the stash.

`git stash -u`

Stashes untracked files as well as changes to tracked files.

`git stash -a`

Stashes all changes, including untracked files and ignored files.

`git stash -p`

Starts an interactive mode where you can select which changes you want stashed.

`git stash pop`

Applies the most recent stash to your working copy, and deletes the stash.

`git stash apply`

Applies the most recent stash to your working copy, but doesn't delete the stash.

`git stash drop`

Deletes the most recent stash.

`git stash branch <new branch name>`

Creates a new branch from the commit where the most recent stash was made, and pops the stash onto it.

`git stash <stash command> stash@{<stash number>}`

Executes a stash command for a specific stash (the most recent stash is stash #0). The command can be show, pop, apply, drop, or branch.

`git stash clear`

Deletes all stashes.

Stashes stay on your local repo, they do not transfer to remote repos when you push.

Tags

`git tag <new tag name>`

Creates a lightweight tag at the current commit.

`git tag <new tag name> <branch name>`

Creates a lightweight tag at the last commit on the specified branch.

`git tag -d <tag name>`

Deletes the specified tag.

`git tag -a <new tag name> -m "<annotation>"`

Creates an annotated tag at the current commit.

`git tag -a <new tag name> -m "<annotation>" <commit id>`

Creates an annotated tag at the specified commit.

`git tag --list`

Lists tags.

`git show <tag name>`

Shows the details of the commit with the specified tag, and the annotation if there is one.

`git tag -f <tag name> <commit id>`

Updates the tag to be at the specified commit. Not including `<commit id>` will put the tag on the last commit (More specifically where HEAD is.)

`git push <remote name> <tag name>`

Pushes the tag to the remote repo.

`git push --force <remote name> <tag name>`

Forces the push of the updated tag.

`git push <remote name> --tags`

Pushes all the tags in the local repo to the remote repo.

`git push <remote name> :<tag name>`

Tells the remote repo to delete the specified tag.

Remote Repos

`git remote -v`

Lists remote repos.

`git remote add <remote name> <remote url>`

Adds reference to the remote repo located at the url, and labels it `<remote name>`. (You can make `<remote name>` anything, but it is conventionally called `origin`.)

`git remote set-url <remote name> <remote url>`

Updates the url for the specified remote reference.

`git remote show <remote name>`

Returns info about the remote.

`git clone <remote url>`

Clones the remote repo into the current directory in a new folder with the same name as the remote repo. (Git labels this remote as `origin` by default.)

`git clone <remote url> <remote name>`

Clones the remote repo into the current directory in a new folder labeled `<remote name>`. (Git labels this remote as `origin` by default.)

Fetching & Pulling

`git fetch <remote url>`

Git goes out to the remote repo and updates its local information about what's in the remote repo.

`git fetch -p <remote url>`

Fetches updates and looks for any dead branches and removes those references (-p stands for prune).

`git pull <remote name> <local branch>`

Pulls changes from the corresponding branch in the remote repo to your `<local branch>`.

`git pull <remote name>`

Pulls changes from the corresponding branch in the remote repo to the branch you are currently on in your local repo.

`git pull` *is a 2-in-1 command:* `git fetch` *then* `git merge`*. Updates are first fetched from the remote repo, then those changes are merged into the local repo.*

Pushing

It is good practice to execute `git pull` *before* `git push`*.*

`git push <remote name> <local branch>`

Pushes the `<local branch>` to the corresponding branch in the remote repo.

`git push <remote name> <local branch> --tags`

Pushes `<local branch>` to the remote repo, plus all tags in the local repo.

`git push -u <remote name> <local branch>`

Pushes the `<local branch>` to the corresponding branch in the remote repo. The `-u` flag sets up an upstream relationship with the branch on the local repo and the corresponding branch on the remote repo. This means when you are on that local branch, you only have to type `git push`, and Git will know where to push it to.

`git push <remote name> :<branch name>`

Tells the remote repo to delete the branch after the colon. (Don't need to use the prune flag to delete dead branch this way.)

Forks

Steps to sync a remote fork with its remote parent repo:

Step 1:

`git remote add parent <remote url>`

Add a new remote reference, referencing the parent repo.

Step 2:

`git pull parent master`

Pull changes from the parent repo to your local repo. Local repo is now synced with parent repo.

Step 3:

`git push origin master`

Push changes to the remote forked repo (usually called `origin`).

Step 4:

`git branch -d <branch name>`

Delete any branches that need to be deleted.

Step 5:

`git fetch -p`

Prune the deleted branches from the local references.

Miscellaneous

`git help <command>`

Returns the documentation for the command.

`git config --global --list`

Lists the variables and their values found in the .gitconfig file.

`git config --global alias.<new command> "<what the new command executes>"`

Creates a new Git command. You can append commands after the alias like with regular Git commands.

The .gitignore file:

- Git will ignore any files listed here.
- Syntax: One pattern or expression per line.
- Target a specific file
  e.g. application.log
- Target file types
  e.g. all log files: *.log

A Crash In Git Bash

`.`

Current directory

`..`

Parent directory

`~`

Home directory

`clear`

Clears the screen.

`exit`

Closes the terminal.

`pwd`

Returns the current working directory.

`ls`

Returns a list of the files and folders in the current directory.

`ls -l`

Returns a detailed list of the files and folders in the current directory.

`ls -a`

Returns a list of the files and folders in the current directory, including hidden files and folders.

`cd <directory>`

Changes the current directory.

`cat <file>`

Returns the contents of the file.

`touch <file name>`

Creates a file in the current directory.

`mv <file name> <new file name>`

Renames a file.

`mv <file> <directory>`

Moves the file to the specified directory.

`rm <file>`

Deletes the file.

`WARNING:` `rm` *can be a very dangerous command, so be sure you know what you are doing if you use it!*

`mkdir <folder name>`

Creates a new folder in the current directory.

`echo "<text>" >> <file>`

Writes some text into the file. Creates the file if it doesn't exit. If there are contents in the file, appends the text to the end of the file.

`echo "<text>" > <file>`

Overwrites the contents of the file with the text given.

`alias <new command>="<what the new command executes>"`

Creates a new command.

Created by Michael Rodeman

Light and dark mode versions of this cheat sheet in DOCX, ODF, PDF, and PNG formats are available at this GitHub repo:

https://github.com/MikeRodeman/Git-Cheat-Sheet