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

Git Cheat Sheet

GitLab

01 Git configuration

`git config --global user.name "Your Name"`	Set the name that will be attached to your commits and tags.
`git config --global user.email "you@example.com"`	Set the e-mail address that will be attached to your commits and tags.
`git config --global color.ui auto`	Enable some colorization of Git output.

02 Starting a project

`git init [project name]`	Create a new local repository in the current directory. If **[project name]** is provided, Git will create a new directory named **[project name]** and will initialize a repository inside it.
`git clone <project url>`	Downloads a project with the entire history from the remote repository.

03 Day-to-day work

`git status`	Displays the status of your working directory. Options include new, staged, and modified files. It will retrieve branch name, current commit identifier, and changes pending commit.
`git add [file]`	Add a file to the **staging** area. Use. in place of the full file path to add all changed files from the **current directory** down into the **directory tree.**
`git diff [file]`	Show changes between **working directory** and **staging area.**
`git diff --staged [file]`	Shows any changes between the **staging area** and the **repository.**
`git checkout -- [file]`	Discard changes in **working directory.** This operation is **unrecoverable**.
`git reset [<path>...]`	Revert some paths in the index (or the whole index) to their state in **HEAD.**
`git commit`	Create a new commit from changes added to the **staging area.** The **commit** must have a message!
`git rm [file]`	Remove file from **working directory** and **staging area.**

04 Storing your work

`git stash`	Put current changes in your **working directory** into **stash** for later use.
`git stash pop`	Apply stored **stash** content into **working directory,** and clear **stash.**
`git stash drop`	Delete a specific **stash** from all your previous **stashes.**

05 Git branching model

`git branch [-a]`	List all local branches in repository. With **-a**: show all branches (with remote).
`git branch [branch_name]`	Create new branch, referencing the current **HEAD**.
`git rebase [branch_name]`	Apply commits of the current working branch and apply them to the HEAD of [branch] to make the history of your branch more linear.
`git checkout [-b] [branch_name]`	Switch working directory to the specified branch. With **-b**: Git will create the specified branch if it does not exist.
`git merge [branch_name]`	Join specified **[branch_name]** branch into your current branch (the one you are on currently).
`git branch -d [branch_name]`	Remove selected branch, if it is already merged into any other. **-D** instead of **-d** forces deletion.

Commit	a state of the code base
Branch	a reference to a commit; can have a **tracked upstream**
Tag	a reference (standard) or an object (annotated)
HEAD	a place where your **working directory** is now

06 Inspect history

`git log [-n count]`	List commit history of current branch. **-n count** limits list to last **n** commits.
`git log --oneline --graph --decorate`	An overview with reference labels and history graph. One commit per line.
`git log ref..`	List commits that are present on the current branch and not merged into **ref**. A **ref** can be a branch name or a tag name.
`git log ..ref`	List commit that are present on **ref** and not merged into current branch.
`git reflog`	List operations (e.g. checkouts or commits) made on local repository.

07 Tagging commits

`git tag`	List all tags.
`git tag [name] [commit sha]`	Create a tag reference named **name** for current commit. Add **commit sha** to tag a specific commit instead of current one.
`git tag -a [name] [commit sha]`	Create a tag object named **name** for current commit.
`git tag -d [name]`	Remove a tag from local repository.

08 Reverting changes

`git reset [--hard] [target reference]`	Switches the current branch to the **target reference**, leaving a difference as an uncommitted change. When **--hard** is used, all changes are discarded. It's easy to lose uncommitted changes with **--hard.**
`git revert [commit sha]`	Create a new commit, reverting changes from the specified commit. It generates an **inversion** of changes.

09 Synchronizing repositories

`git fetch [remote]`	Fetch changes from the **remote**, but not update tracking branches.
`git fetch --prune [remote]`	Delete remote Refs that were removed from the **remote** repository.
`git pull [remote]`	Fetch changes from the **remote** and merge current branch with its upstream.
`git push [--tags] [remote]`	Push local changes to the **remote**. Use **--tags** to push tags.
`git push -u [remote] [branch]`	Push local branch to **remote** repository. Set its copy as an upstream.

10 Git installation

For GNU/Linux distributions, Git should be available in the standard system repository. For example, in Debian/Ubuntu please type inthe terminal:

```
sudo apt-get install git
```

If you need to install Git from source, you can get it from **git-scm.com/downloads**.

An excellent Git course can be found in the great Pro Git book by Scott Chacon and Ben Straub. The book is available online for free at **git-scm.com/book.**

11 Ignoring files

```
cat   <<EOF >  .gitignore
/logs/*
!logs/.gitkeep
/tmp
*.swp
EOF
```

To ignore files, create a .gitignore file in your repository with a line for each pattern. File ignoring will work for the current and sub directories where .gitignore file is placed. In this example, all files are ignored in the logs directory (excluding the .gitkeep file), whole tmp directory and all files *.swp.

datacamp

Git Cheat Sheet

Learn Git online at www.DataCamp.com

What is Version Control?

Version control systems are tools that manage changes made to files and directories in a project. They allow you to keep track of what you did when, undo any changes you decide you don't want, and collaborate at scale with others. This cheat sheet focuses on one of the most popular one, Git.

> Key Definitions

Throughout this cheat sheet, you'll find git-specific terms and jargon being used. Here's a run-down of all the terms you may encounter

Basic definitions

- **Local repo or repository:** A local directory containing code and files for the project
- **Remote repository:** An online version of the local repository hosted on services like GitHub, GitLab, and BitBucket
- **Cloning:** The act of making a clone or copy of a repository in a new directory
- **Commit:** A snapshot of the project you can come back to
- **Branch:** A copy of the project used for working in an isolated environment without affecting the main project
- **Git merge:** The process of combining two branches together

More advanced definitions

- `.gitignore` **file:** A file that lists other files you want git not to track (e.g. large data folders, private info, and any local files that shouldn't be seen by the public.)
- **Staging area:** a cache that holds changes you want to commit next.
- **Commit ID or hash:** a unique identifier for each commit, used for switching to different save points.
- **HEAD** *(always capitalized letters)***:** a reference name for the latest commit, to save you having to type Commit IDs. `HEAD~n` syntax is used to refer to older commits (e.g. `HEAD~2` refers to the second-to-last commit).

> Installing Git

On OS X — Using an installer

1. Download the installer for Mac
2. Follow the prompts

On OS X — Using Homebrew

```
$ brew install git
```

On Linux

```
$ sudo apt-get install git
```

On Windows

1. Download the latest Git For Windows installer
2. Follow the prompts

Check if installation successful (On any platform)

```
$ git --version
```

> Setting Up Git

If you are working in a team on a single repo, it is important for others to know who made certain changes to the code. So, Git allows you to set user credentials such as name, email, etc..

Set your basic information

- Configure your email
  `$ git config user.email [your.email@domain.com]`
- Configure your name
  `$ git config user.name [your-name]`

Important tags to determine the scope of configurations

Git lets you use tags to determine the scope of the information you're using during setup

- Local directory, single project (this is the default tag)
  `$ git config --local user.email "my_email@example.com"`
- All git projects under the current user
  `$ git config --global user.email "my_email@example.com"`
- For all users on the current machine
  `$ git config --system user.email "my_email@example.com"`

Other useful configuration commands

- List all key-value configurations
  `$ git config --list`
- Get the value of a single key
  `$ git config --get <key>`

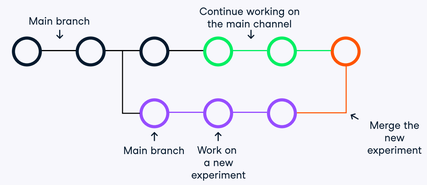
Setting aliases for common commands

If you find yourself using a command frequently, git lets you set an alias for that command to surface it more quickly

- Create an alias named gc for the "git commit" command
  `$ git config --global alias.gc commit`
  `$ gc -m "New commit"`
- Create an alias named ga for the "git add" command
  `$ git config --global alias.ga add`

> What is a Branch?

Branches are special "copies" of the code base which allow you to work on different parts of a project and new features in an isolated environment. Changes made to the files in a branch won't affect the "main branch" which is the main project development channel.



> Git Basics

What is a repository?

A repository or a repo is any location that stores code and the necessary files that allow it to run without errors. A repo can be both local and remote. A local repo is typically a directory on your machine while a remote repo is hosted on servers like GitHub

Creating local repositories

- Clone a repository from remote hosts (GitHub, GitLab, DagsHub, etc.)
  `$ git clone <remote_repo_url>`
- Initialize git tracking inside the current directory
  `$ git init`
- Create a git-tracked repository inside a new directory
  `$ git init [dir_name]`
- Clone only a specific branch
  `$ git clone -branch <branch_name> <repo_url>`
- Cloning into a specified directory
  `$ git clone <repo_url> <dir_name>`

A note on cloning

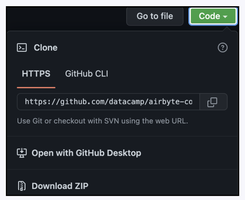
There are two primary methods of cloning a repository - HTTPS syntax and SSH syntax. While SSH cloning is generally considered a bit more secure because you have to use an SSH key for authentication, HTTPS cloning is much simpler and the recommended cloning option by GitHub.

HTTPS

```
$ git clone https://github.com/your_username/repo_name.git
```

SSH

```
$ git clone git@github.com:user_name/repo_name.git
```



Managing remote repositories

- List remote repos
  `$ git remote`
- Create a new connection called <remote> to a remote repository on servers like GitHub, GitLab, DagsHub, etc.
  `$ git remote add <remote> <url_to_remote>`
- Remove a connection to a remote repo called <remote>
  `$ git remote rm <remote>`
- Rename a remote connection
  `$ git remote rename <old_name> <new_name>`

> Working With Files

Adding and removing files

- Add a file or directory to git for tracking
  `$ git add <filename_or_dir>`
- Add all untracked and tracked files inside the current directory to git
  `$ git add .`
- Remove a file from a working directory or staging area
  `$ git rm <<filename_or_dir>`

Saving and working with changes

- See changes in the local repository
  `$ git status`
- Saving a snapshot of the staged changes with a custom message
  `$ git commit -m "[Commit message]"`
- Staging changes in all tracked files and committing with a message
  `$ git add -am "[Commit message]"`
- Editing the message of the latest commit
  `$ git commit --amend -m "[New commit message]"`
- Saving staged and unstaged changes to stash for a later use (see below for the explanation of a stash)
  `$ git stash`
- Stashing staged, unstaged and untracked files as well
  `$ git stash -u`
- Stashing everything (including ignored files)
  `$ git stash --all`
- Reapply previously stashed changes and empty the stash
  `$ git stash pop`
- Reapply previously stashed changes and keep the stash
  `$ git stash apply`
- Dropping changes in the stash
  `$ git stash drop`
- Show uncommitted changes since the last commit
  `$ git diff`
- Show the differences between two commits (should provide the commit IDs)
  `$ git diff <id_1> <id_2>`

A note on stashes

Git stash allows you to temporarily save edits you've made to your working copy so you can return to your work later. Stashing is especially useful when you are not yet ready to commit changes you've done, but would like to revisit them at a later time.

Branches

- List all branches
  `$ git branch`
  `$ git branch --list`
  `$ git branch -a` *(shows remote branches as well)*
- Create a new local branch named `new_branch` without checking out that branch
  `$ git branch <new_branch>`
- Switch into an existing branch named <branch>
  `$ git checkout <branch>`
- Create a new local branch and switch into it
  `$ git checkout -b <new_branch>`
- Safe delete a local branch (prevents deleting unmerged changes)
  `$ git branch -d <branch>`
- Force delete a local branch (whether merged or unmerged)
  `$ git branch -D <branch>`
- Rename the current branch to <new_name>
  `$ git branch -m <new_name>`
- Push a copy of local branch named branch to the remote repo
  `$ git push <remote_repo> branch~`
- Delete a remote branch named branch *(-d tag only works locally)*
  `$ git push <remote_repo> :branch`
  `$ git push <remote_repo> --delete branch`
- Merging a branch into the main branch
  `$ git checkout main`
  `$ git merge <other_branch>`
- Merging a branch and creating a commit message
  `$ git merge --no-ff <other_branch>`
- Compare the differences between two branches
  `$ git diff <branch_1> <branch_2>`
- Compare a single <file> between two branches
  `$ git diff <branch_1> <branch_2> <file>`

Pulling changes

- Download all commits and branches from the <remote> without applying them on the local repo
  `$ git fetch <remote>`
- Only download the specified <branch> from the <remote>
  `$ git fetch <remote> <branch>`
- Merge the fetched changes if accepted
  `$ git merge <remote>/<branch>`
- A more aggressive version of fetch which calls fetch and merge simultaneously
  `$ git pull <remote>`

Logging and reviewing work

- List all commits with their author, commit ID, date and message
  `$ git log`
- List one commit per line *(-n tag can be used to limit the number of commits displayed (e.g. -5))*
  `$ git log --oneline [-n]`
- Log all commits with diff information:
  `$ git log --stat`
- Log commits after some date *(A sample value can be 4th of October, 2020 - "2020-10-04" or keywords such as "yesterday", "last month", etc.)*
  `$ git log --oneline --after="YYYY-MM-DD"`
- Log commits before some date (Both `--after` and `--before` tags can be used for date ranges)
  `$ git log --oneline --before="last year"`

Reversing changes

- Checking out (switching to) older commits
  `$ git checkout HEAD~3`
- Checks out the third-to-last commit.
  `$ git checkout <commit_id>`
- Undo the latest commit but leave the working directory unchanged
  `$ git reset HEAD~1`

You can undo as many commits as you want by changing the number after the tilde.

- Discard all changes of the latest commit (no easy recovery)
  `$ git reset --hard HEAD~1`

Instead of `HEAD~n`, you can provide commit hash as well. Changes after that commit will be destroyed.

- Undo a single given commit, without modifying commits that come after it (a safe reset)
  `$ git revert [commit_id]`

May result in revert conflicts

Git Cheat Sheet

GIT BASICS

`git init <directory>`	Create empty Git repo in specified directory. Run with no arguments to initialize the current directory as a git repository.
`git clone <repo>`	Clone repo located at `<repo>` onto local machine. Original repo can be located on the local filesystem or on a remote machine via `HTTP` or `SSH`.
`git config user.name <name>`	Define author name to be used for all commits in current repo. Devs commonly use `--global` flag to set config options for current user.
`git add <directory>`	Stage all changes in `<directory>` for the next commit. Replace `<directory>` with a `<file>` to change a specific file.
`git commit -m "<message>"`	Commit the staged snapshot, but instead of launching a text editor, use `<message>` as the commit message.
`git status`	List which files are staged, unstaged, and untracked.
`git log`	Display the entire commit history using the default format. For customization see additional options.
`git diff`	Show unstaged changes between your index and working directory.

UNDOING CHANGES

`git revert <commit>`	Create new commit that undoes all of the changes made in `<commit>`, then apply it to the current branch.
`git reset <file>`	Remove `<file>` from the staging area, but leave the working directory unchanged. This unstages a file without overwriting any changes.
`git clean -n`	Shows which files would be removed from working directory. Use the `-f` flag in place of the `-n` flag to execute the clean.

REWRITING GIT HISTORY

`git commit --amend`	Replace the last commit with the staged changes and last commit combined. Use with nothing staged to edit the last commit's message.
`git rebase <base>`	Rebase the current branch onto `<base>`. `<base>` can be a commit ID, branch name, a tag, or a relative reference to `HEAD`.
`git reflog`	Show a log of changes to the local repository's `HEAD`. Add `--relative-date` flag to show date info or `--all` to show all refs.

GIT BRANCHES

`git branch`	List all of the branches in your repo. Add a `<branch>` argument to create a new branch with the name `<branch>`.
`git checkout -b <branch>`	Create and check out a new branch named `<branch>`. Drop the `-b` flag to checkout an existing branch.
`git merge <branch>`	Merge `<branch>` into the current branch.

REMOTE REPOSITORIES

`git remote add <name> <url>`	Create a new connection to a remote repo. After adding a remote, you can use `<name>` as a shortcut for `<url>` in other commands.
`git fetch <remote> <branch>`	Fetches a specific `<branch>`, from the repo. Leave off `<branch>` to fetch all remote refs.
`git pull <remote>`	Fetch the specified remote's copy of current branch and immediately merge it into the local copy.
`git push <remote> <branch>`	Push the branch to `<remote>`, along with necessary commits and objects. Creates named branch in the remote repo if it doesn't exist.

Additional Options +

GIT CONFIG

`git config --global user.name <name>`	Define the author name to be used for all commits by the current user.
`git config --global user.email <email>`	Define the author email to be used for all commits by the current user.
`git config --global alias. <alias-name> <git-command>`	Create shortcut for a Git command. E.g. `alias.glog "log --graph --oneline"` will set `"git glog"` equivalent to `"git log --graph --oneline.`
`git config --system core.editor <editor>`	Set text editor used by commands for all users on the machine. `<editor>` arg should be the command that launches the desired editor (e.g., vi).
`git config --global --edit`	Open the global configuration file in a text editor for manual editing.

GIT LOG

`git log -<limit>`	Limit number of commits by `<limit>`. E.g. `"git log -5"` will limit to 5 commits.
`git log --oneline`	Condense each commit to a single line.
`git log -p`	Display the full diff of each commit.
`git log --stat`	Include which files were altered and the relative number of lines that were added or deleted from each of them.
`git log --author= "<pattern>"`	Search for commits by a particular author.
`git log --grep="<pattern>"`	Search for commits with a commit message that matches `<pattern>`.
`git log <since>..<until>`	Show commits that occur between `<since>` and `<until>`. Args can be a commit ID, branch name, `HEAD`, or any other kind of revision reference.
`git log -- <file>`	Only display commits that have the specified file.
`git log --graph --decorate`	`--graph` flag draws a text based graph of commits on left side of commit msgs. `--decorate` adds names of branches or tags of commits shown.

GIT DIFF

`git diff HEAD`	Show difference between working directory and last commit.
`git diff --cached`	Show difference between staged changes and last commit

GIT RESET

`git reset`	Reset staging area to match most recent commit, but leave the working directory unchanged.
`git reset --hard`	Reset staging area and working directory to match most recent commit and **overwrites all changes** in the working directory.
`git reset <commit>`	Move the current branch tip backward to `<commit>`, reset the staging area to match, but leave the working directory alone.
`git reset --hard <commit>`	Same as previous, but resets both the staging area & working directory to match. **Deletes** uncommitted changes, and **all commits after** `<commit>`.

GIT REBASE

`git rebase -i <base>`	Interactively rebase current branch onto `<base>`. Launches editor to enter commands for how each commit will be transferred to the new base.

GIT PULL

`git pull --rebase <remote>`	Fetch the remote's copy of current branch and rebases it into the local copy. Uses git rebase instead of merge to integrate the branches.

GIT PUSH

`git push <remote> --force`	Forces the `git push` even if it results in a non-fast-forward merge. Do not use the `--force` flag unless you're absolutely sure you know what you're doing.
`git push <remote> --all`	Push all of your local branches to the specified remote.
`git push <remote> --tags`	Tags aren't automatically pushed when you push a branch or use the `--all` flag. The `--tags` flag sends all of your local tags to the remote repo.