Git Cheat Sheet

Setup

Set the name and email that will be attached to your commits and tags

```
$ git config --global user.name "Danny Adams"
$ git config --global user.email "my-email@gmail.com"
```

Start a Project

Create a local repo (omit <directory> to initialise the current directory as a git repo

```
$ git init <directory>
```

Download a remote repo

```
$ git clone <url>
```

Make a Change

Add a file to staging

```
$ git add <file>
```

Stage all files

```
$ git add .
```

Commit all staged files to git

```
$ git commit -m "commit message"
```

Add all changes made to tracked files & commit

```
$ git commit -am "commit message"
```

Basic Concepts

main: default development branch
origin: default upstream repo
HEAD: current branch
HEAD^: parent of HEAD
HEAD~4: great-great grandparent of HEAD

By @DoableDanny

Branches

List all local branches. Add -r flag to show all remote branches. -a flag for all branches.

```
$ git branch
```

Create a new branch

```
$ git branch <new-branch>
```

Switch to a branch & update the working directory

```
$ git checkout <branch>
```

Create a new branch and switch to it

```
$ git checkout -b <new-branch>
```

Delete a merged branch

```
$ git branch -d <branch>
```

Delete a branch, whether merged or not

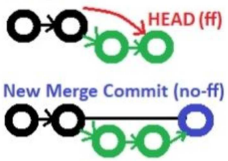
```
$ git branch -D <branch>
```

Add a tag to current commit (often used for new version releases)

```
$ git tag <tag-name>
```

Merging

Merge branch a into branch b. Add --no-ff option for no-fast-forward merge



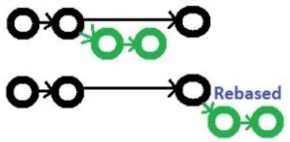

```
$ git checkout b
$ git merge a
```

Merge & squash all commits into one new commit

```
$ git merge --squash a
```

Rebasing

Rebase feature branch onto main (to incorporate new changes made to main). Prevents unnecessary merge commits into feature, keeping history clean




```
$ git checkout feature
$ git rebase main
```

Interatively clean up a branches commits before rebasing onto main

```
$ git rebase -i main
```

Interatively rebase the last 3 commits on current branch

```
$ git rebase -i Head~3
```

Undoing Things

Move (&/or rename) a file & stage move

```
$ git mv <existing_path> <new_path>
```

Remove a file from working directory & staging area, then stage the removal

```
$ git rm <file>
```

Remove from staging area only

```
$ git rm --cached <file>
```

View a previous commit (READ only)

```
$ git checkout <commit_ID>
```

Create a new commit, reverting the changes from a specified commit

```
$ git revert <commit_ID>
```

Go back to a previous commit & delete all commits ahead of it (revert is safer). Add --hard flag to also delete workspace changes (BE VERY CAREFUL)

```
$ git reset <commit_ID>
```

Review your Repo

List new or modified files not yet committed

```
$ git status
```

List commit history, with respective IDs

```
$ git log --oneline
```

Show changes to unstaged files. For changes to staged files, add --cached option

```
$ git diff
```

Show changes between two commits

```
$ git diff commit1_ID commit2_ID
```

Stashing

Store modified & staged changes. To include untracked files, add -u flag. For untracked & ignored files, add -a flag.

```
$ git stash
```

As above, but add a comment.

```
$ git stash save "comment"
```

Partial stash. Stash just a single file, a collection of files, or individual changes from within files

```
$ git stash -p
```

List all stashes

```
$ git stash list
```

Re-apply the stash without deleting it

```
$ git stash apply
```

Re-apply the stash at index 2, then delete it from the stash list. Omit stash@{n} to pop the most recent stash.

```
$ git stash pop stash@{2}
```

Show the diff summary of stash 1. Pass the -p flag to see the full diff.

```
$ git stash show stash@{1}
```

Delete stash at index 1. Omit stash@{n} to delete last stash made

```
$ git stash drop stash@{1}
```

Delete all stashes

```
$ git stash clear
```

Synchronizing

Add a remote repo

```
$ git remote add <alias> <url>
```

View all remote connections. Add -v flag to view urls.

```
$ git remote
```

Remove a connection

```
$ git remote remove <alias>
```

Rename a connection

```
$ git remote rename <old> <new>
```

Fetch all branches from remote repo (no merge)

```
$ git fetch <alias>
```

Fetch a specific branch

```
$ git fetch <alias> <branch>
```

Fetch the remote repo's copy of the current branch, then merge

```
$ git pull
```

Move (rebase) your local changes onto the top of new changes made to the remote repo (for clean, linear history)

```
$ git pull --rebase <alias>
```

Upload local content to remote repo

```
$ git push <alias>
```

Upload to a branch (can then pull request)

```
$ git push <alias> <branch>
```

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.
- **Git stash:** another type of cache that holds unwanted changes you may want to come back later
- **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>`

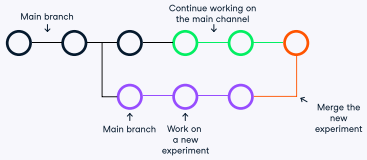
Setting aliases for common commands

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

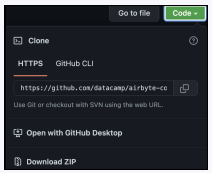
Creating local repositories
- Clone a repository from remote hosts (GitHub, GitLab, DagsHub, etc.)
  `$ git clone <remote_repo_url>`
- Initialize git tracking inside the current directory
  `$ git init`
- Create a git-tracked repository inside a new directory
  `$ git init [dir_name]`
- Clone only a 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