Git Commands Reference Guide

Understanding Git & GitHub

Git: Version control software that tracks changes in your files and maintains complete history of your project.

GitHub: Cloud-based service to store Git repositories online, collaborate with others, and share code.

Version Control System: Keeps track of all file changes, allowing you to revert to previous versions anytime.

`git --version`

Check which version of Git is installed on your system.

```bash
git --version
```

Output Example: `git version 2.40.0`

Why use it: Verify Git is installed and check if you need to update it.

`git init`

Initialize a new Git repository in your current directory. This is a **one-time setup per project**.

```bash
git init
```

What happens:

- Creates a hidden `.git` folder in your project
- This folder stores the complete history of all files and changes
- Turns your regular folder into a Git-tracked project

When to use: When starting a new project that you want to track with Git.

`git config`

Configure your Git username and email. Required for commits.

```bash
git config --global user.name "Your Name"
git config --global user.email "your.email@example.com"
```

`--global`: Sets configuration for all repositories on your system.

Without **`--global`**: Sets configuration only for the current repository.

Check your configuration:

```bash
git config --list
```

`git status`

Shows the current state of your working directory and staging area.

```bash
git status
```

What it shows:

- Which files are modified but not staged
- Which files are staged and ready to commit
- Which files are untracked (new files Git doesn't know about)

When to use: Before committing, to see what changes you're about to save.

`git add`

Add file(s) to the staging area, preparing them for commit.

```bash
git add filename.txt        # Add specific file
git add file1.txt file2.txt   # Add multiple files
git add .                   # Add all changed files in current directory
git add *.js                # Add all JavaScript files
```

What staging means: Files are marked to be included in the next commit.

When to use: After making changes and before committing.

`git commit -m "message"`

Save staged changes to the repository with a descriptive message.

```bash
git commit -m "Add user login feature"
```

What it does: Creates a permanent snapshot of staged changes in Git history.

Good commit message: Short, descriptive, explains what changed and why.

Bad commit message: "update", "fix", "changes" (too vague).

When to use: After staging files you want to save permanently.

`git commit -am "message"`

Add and commit in one command. Only works for **already tracked files**, not new files.

```bash
git commit -am "Fix navbar styling"
```

What `-am` means:

- `-a` = automatically stage all modified tracked files
- `-m` = add commit message

Limitation: Does NOT work for new files (untracked files). Use `git add` first for new files.

When to use: Quick commits when you want to skip the `git add` step for existing files.

`git log`

Display detailed history of all commits made in the repository.

```bash
git log
```

What it shows:

- Commit hash (unique ID)
- Author name and email
- Date and time of commit
- Commit message

When to use: When you want to see complete commit history with full details.

`git log --oneline`

Show commit history in condensed, one-line-per-commit format.

```bash
git log --oneline
```

What it shows:

- Shortened commit hash (first 7 characters)
- Commit message only

When to use: Quick overview of recent changes without clutter.

`git diff`

Show line-by-line changes between your working directory and the last commit.

```bash
git diff
```

What it shows:

- Lines **added** → shown with `+` (green)
- Lines **removed** → shown with `-` (red)
- Which files were modified

When to use: Before committing, to review exactly what you changed.

`git diff --staged`

Show changes that are staged (added to staging area) and ready to commit.

```bash
git diff --staged
```

Difference from `git diff`:

- `git diff` shows unstaged changes
- `git diff --staged` shows staged changes

When to use: After `git add`, to verify what will be included in next commit.

`git diff <commit1> <commit2>`

Compare changes between two specific commits.

```bash
git diff abc1234 def5678
```

When to use: To see what changed between two points in history.

`git diff HEAD <filename>`

Compare a specific file's current version with the last committed version.

```bash
git diff HEAD index.html
```

What `HEAD` means: Pointer to your current commit (usually the latest commit on current branch).

When to use: To see changes in one specific file.

`git restore <filename>`

Discard changes in a file and restore it to the last committed version.

```bash
git restore index.html
```

What it does: Throws away all uncommitted changes in the file.

⚠️ **Warning**: This **permanently deletes** unsaved work! Cannot be undone.

When to use: When you made mistakes and want to go back to the last saved version.

`git restore --staged <filename>`

Unstage a file (remove from staging area) without losing changes.

```bash
git restore --staged script.js
```

What it does: Moves file from staging area back to working directory. Changes are NOT lost.

When to use: When you accidentally staged a file you don't want to commit yet.

`git reset HEAD~1`

Undo the last commit but keep changes in your working directory.

```bash
git reset HEAD~1
```

What it does:

- Removes the last commit from history
- Moves changes back to working directory (unstaged)
- You can modify and recommit

`HEAD~1`: One commit before current HEAD. Use `HEAD~2` for two commits back, etc.

When to use: When you want to undo last commit but keep the work you did.

`git reset --hard HEAD~1`

Undo the last commit and **permanently delete** all changes.

```bash
git reset --hard HEAD~1
```

What it does:

- Removes last commit from history
- **Deletes all changes permanently**
- No way to recover unless you use `git reflog`

⚠️ **Warning**: This is destructive! Use only when you're absolutely sure.

When to use: When you want to completely remove last commit and its changes.

`git reflog`

Show a log of all HEAD movements - every commit, checkout, reset, rebase, merge, etc.

```bash
git reflog
```

What it shows:

- Complete history of where your HEAD has been
- Includes commits that might not show in `git log` (like deleted commits)
- Each entry has a reference like `HEAD@{0}`, `HEAD@{1}`, etc.

When to use:

- **Recovery tool** when you accidentally delete commits or branches
- To see "where you came from" in your Git journey
- To undo mistakes by going back to a previous state

Example recovery:

```bash
git reflog                  # Find the commit hash you want
git checkout abc1234        # Go to that commit
git switch -c recovery-branch # Create branch to save it
```

`git branch`

List all branches in your repository. Current branch is marked with `*`.

```bash
git branch
```

Output example:

```
  feature-login
* main
  feature-signup
```

When to use: To see which branches exist and which one you're currently on.

---

git branch <branch-name>

Create a new branch.

```bash
git branch feature-login
```

What it does: Creates a new branch but does NOT switch to it.

When to use: When you want to work on a new feature without affecting main code.

---

git branch -d <branch-name>

Delete a branch safely (only if it has been merged).

```bash
git branch -d feature-login
```

What it does: Deletes the branch only if its changes have been merged into another branch.

When to use: Clean up after merging a feature branch into main.

---

git branch -D <branch-name>

Force delete a branch even if it hasn't been merged.

```bash
git branch -D feature-login
```

What it does: Deletes the branch regardless of merge status.

⚠️ **Warning**: You'll lose all commits unique to that branch if not merged elsewhere.

When to use: When you want to discard a branch and all its work.

`git checkout <branch-name>`

Switch to a different branch.

```bash
git checkout feature-login
```

What it does:

- Changes your working directory to match that branch
- Updates HEAD to point to that branch
- Any new commits will be added to that branch

When to use: When you want to work on a different branch.

`git checkout -b <branch-name>`

Create a new branch and switch to it immediately.

```bash
git checkout -b feature-signup
```

What it does: Combines `git branch` and `git checkout` in one command.

When to use: Shortcut for creating and switching to branches in one step.

`git switch <branch-name>`

Switch to a different branch (modern, clearer alternative to `git checkout`).

```bash
git switch main
```

What it does: Same as `git checkout <branch>` but only for switching branches (no other checkout functions).

When to use: When you want to switch branches. More intuitive than `checkout`.

`git switch -c <branch-name>`

Create a new branch and switch to it (modern alternative).

```bash
git switch -c feature-payment
```

What it does: Same as `git checkout -b` but with clearer syntax.

When to use: Modern way to create and switch to a new branch.

`git checkout <commit-hash>`

Go back to a specific commit. Creates **detached HEAD** state.

```bash
git checkout abc1234
```

What "Detached HEAD" means:

- You're viewing an old commit
- You're not on any branch
- Any new commits won't be saved unless you create a new branch

When to use: To inspect old code without making changes.

To return to normal:

```bash
git switch main
```

`git checkout HEAD~2`

Go back 2 commits before the current one.

```bash
```

```
git checkout HEAD~2
```

Syntax:

- `HEAD~1` = 1 commit back
- `HEAD~2` = 2 commits back
- `HEAD~5` = 5 commits back

When to use: Quick way to look at recent history without typing commit hashes.

`git merge <branch-name>`

Merge another branch into your current branch.

```bash
git checkout main
git merge feature-login
```

What it does:

- Combines changes from `feature-login` into `main`
- Creates a **merge commit** that records the merge
- Preserves the complete history of both branches

When to use: When a feature is complete and you want to add it to main branch.

Merge conflict: If both branches modified the same lines, Git can't auto-merge. You must manually resolve conflicts.

`git rebase <branch-name>`

Rewrite commit history by moving your commits to the tip of another branch.

```bash
git checkout feature-branch
git rebase main
```

What it does:

- Takes your commits from `feature-branch`
- Replays them **on top of** the latest `main` branch
- Creates a clean, linear history (no merge commits)
- Changes commit hashes

When to use: To keep your branch updated with main without creating merge commits.

⚠️ **CRITICAL RULE**:

- **NEVER rebase commits that have been pushed to a shared repository**
- **Only rebase local, unpushed commits**
- Rebasing changes commit hashes which causes conflicts for collaborators

Safe usage: Rebase your local feature branch before pushing for the first time.

`git rebase -i HEAD~5`

Interactive rebase - edit, reorder, squash, or delete the last 5 commits.

```bash
git rebase -i HEAD~5
```

What it does:

- Opens an editor showing last 5 commits (in **reverse order** - oldest first)
- You can modify commits before they're replayed

Options:

- `pick` = use commit as is
- `reword` = change commit message
- `edit` = stop to modify commit
- `squash` = combine with previous commit
- `fixup` = like squash but discard commit message

- `drop` = delete commit
- Reorder by moving lines up/down

When to use:

- Clean up messy commit history before pushing
- Combine multiple small commits into one meaningful commit
- Fix typos in commit messages
- Remove debug commits

⚠️ **Warning**: Only do this on **local, unpushed commits**! Rebasing shared commits causes issues.

Merge vs Rebase Comparison

`git merge main`

What it does:

- Combines your branch with main
- Creates a **merge commit** that shows the merge happened
- Preserves the complete history of both branches
- History shows when branches diverged and merged

Result:

```
  A---B---C feature
 /         \
D---E---F---G main
```

Pros:

- Preserves exact history
- Safe for shared branches
- Easy to understand what happened

Cons:

- Creates extra merge commits
- History can become cluttered

`git rebase main`

What it does:

- Moves your commits to the tip of main
- Rewrites history to make it **linear**
- No merge commit created
- Changes commit hashes

Result:

```
D---E---F---A'---B'---C' main (with feature commits)
```

Pros:

- Clean, linear history
- Easier to read `git log`
- No merge commit clutter

Cons:

- Rewrites history (dangerous for shared branches)
- Harder to track when features were merged

`git stash`

Temporarily save uncommitted changes so you can switch branches cleanly.

```bash
git stash
```

What it does:

- Saves your modified files in a "stash" (temporary storage)
- Reverts your working directory to the last commit (clean state)
- Allows you to switch branches without committing incomplete work

When to use: When you need to switch branches but aren't ready to commit yet.

Example scenario: You're working on a feature but need to urgently fix a bug on main. Stash your work, switch to main, fix bug, come back and restore stash.

---

git stash pop

Bring back stashed changes and remove them from the stash.

```bash
git stash pop
```

What it does:

- Applies most recent stash to your working directory
- Deletes that stash after applying

When to use: After switching back to your branch, to continue working on saved changes.

---

git stash apply

Apply stashed changes but keep them in the stash for reuse.

```bash
git stash apply
```

What it does:

- Applies most recent stash to your working directory
- Keeps the stash saved for later use

When to use: When you want to apply the same changes to multiple branches.

---

git stash list

View all stashed changes.

```bash
git stash list
```

Output example:

```
stash@{0}: WIP on feature-login: abc1234 Add login form
stash@{1}: WIP on main: def5678 Update README
```

When to use: To see what changes you've stashed.

---

git stash drop

Delete the most recent stash.

```bash
git stash drop
```

When to use: When you no longer need the stashed changes.

---

git stash clear

Delete all stashes permanently.

```bash
git stash clear
```

⚠️ **Warning**: All stashed changes will be lost!

When to use: Clean up when you're sure you don't need any stashed changes.

`git clone <url>`

Download a repository from GitHub (or other remote) to your local machine.

```bash
git clone https://github.com/username/repo.git
```

What it does:

- Downloads entire repository including all history
- Creates a folder with the repository name
- Automatically sets up remote connection to origin

When to use: When you want to work on an existing project from GitHub.

`git remote add origin <url>`

Connect your local repository to a remote repository on GitHub.

```bash
git remote add origin https://github.com/username/repo.git
```

What `origin` means: Default name for the remote repository (you can use any name).

When to use: After creating a repo locally with `git init`, to connect it to GitHub.

`git remote -v`

View all remote repositories connected to your local repo.

```bash
git remote -v
```

Output example:

```
origin  https://github.com/username/repo.git (fetch)
origin  https://github.com/username/repo.git (push)
```

When to use: To verify which remote repositories you're connected to.

`git push`

Upload your commits to the remote repository (GitHub).

```bash
git push
```

What it does: Sends your local commits to the remote branch.

First time pushing a new branch: Use `git push -u origin <branch-name>` to set up tracking.

When to use: After committing changes you want to share or backup on GitHub.

`git push origin <branch-name>`

Upload commits from a specific branch to GitHub.

```bash
git push origin main
git push origin feature-login
```

When to use: When you need to explicitly specify which branch to push.

`git push -u origin <branch-name>`

Push and set upstream tracking for the branch.

```bash
git push -u origin main
```

What `-u` does:

- Pushes your branch to remote
- Sets up tracking so future `git push` and `git pull` commands know which remote branch to use
- You only need to use `-u` once per branch

After using `-u`: You can just use `git push` without specifying branch name.

When to use: First time pushing a new branch to GitHub.

`git push --force`

Force push - overwrites remote history with your local history.

```bash
git push --force
```

What it does: Replaces remote branch history with your local branch history, even if they conflict.

⚠️ **EXTREME WARNING**:

- This can **delete other people's work**
- Never use on shared branches
- Only use when you're absolutely sure and working alone

When to use: After rebasing local commits or fixing history (only if you're working solo or after team agreement).

Safer alternative: `git push --force-with-lease` (fails if remote has changes you don't have).

`git pull`

Download changes from GitHub and merge them into your current branch.

```bash
git pull
```

What it does:

- Fetches changes from remote
- Automatically merges them into your current branch

Equivalent to: `git fetch` + `git merge`

When to use: To update your local branch with latest changes from GitHub.

`git fetch`

Download changes from GitHub but don't merge them.

```bash
git fetch
```

What it does:

- Downloads all new commits from remote
- Updates your remote tracking branches
- Does NOT modify your working directory

When to use: When you want to see what's new on remote before merging.

After fetching: Use `git merge origin/main` to merge changes manually.

.gitignore

A special file that tells Git which files or folders to ignore and not track.

Create file: Create a file named `.gitignore` in your project root.

Example `.gitignore`:

```
# API keys and secrets
.env
config.json
api_keys.txt

# Dependencies
node_modules/
venv/
vendor/

# Build files
dist/
build/
*.log

# Operating system files
.DS_Store
Thumbs.db
*.swp

# IDE settings
.vscode/
.idea/
*.sublime-workspace
```

Syntax:

- `filename.txt` = ignore specific file
- `*.log` = ignore all .log files
- `folder/` = ignore entire folder
- `!important.log` = don't ignore this file (exception)
- `#` = comment

When to use:

- Prevent sensitive data (API keys, passwords) from being committed
- Exclude large dependency folders (node_modules, venv)
- Ignore auto-generated files
- Keep personal IDE settings private

Already committed file?: If you already committed a file and then add it to `.gitignore`, you need to remove it from Git:

```bash
git rm --cached filename.txt
git commit -m "Remove file from tracking"
```

Why "Master" Became "Main"

Historical context: When Git was created, the default branch was called "master" and other branches were sometimes referred to as "slave" branches.

The problem: Many people found this terminology offensive due to its historical association with slavery.

The change: In 2020, the tech community adopted more inclusive language:

- "master" → "main"
- "slave" → "other branches" or specific feature names

Current state:

- GitHub, GitLab, and other platforms now default to "main" for new repositories
- Older repositories may still use "master" (both work technically)
- "main" is the modern standard

Note: This is purely a naming convention - both "master" and "main" function identically.

Quick Command Cheat Sheet

```bash
```

```
# Setup
git init                        # Initialize repository
git config --global user.name   # Set username
git config --global user.email  # Set email

# Basic workflow
git status                      # Check status
git add .                       # Stage all changes
git commit -m "message"         # Commit changes
git push                        # Upload to GitHub

# Viewing changes
git log                         # View commit history
git log --oneline               # View short history
git diff                        # See unstaged changes
git diff --staged               # See staged changes

# Branching
git branch                      # List branches
git branch feature-name         # Create branch
git checkout feature-name       # Switch branch
git checkout -b feature-name    # Create and switch
git merge feature-name          # Merge branch

# Undoing
git restore file.txt            # Discard changes
git reset HEAD~1                # Undo last commit (keep changes)
git reset --hard HEAD~1         # Undo last commit (delete changes)

# Stashing
git stash                       # Save changes temporarily
git stash pop                   # Restore stashed changes

# Remote
git clone url                   # Download repository
git pull                        # Download and merge changes
git push                        # Upload changes
git fetch                       # Download without merging

# Recovery
git reflog                      # View all history (recovery tool)
```

Emergency Recovery Scenarios

Lost a commit after reset?

```bash
git reflog                    # Find lost commit hash
git checkout abc1234          # Go to that commit
git switch -c recovery-branch # Create branch to save it
```

Accidentally committed to wrong branch?

```bash
git log --oneline             # Find commit hash
git checkout correct-branch   # Switch to correct branch
git cherry-pick abc1234       # Copy commit here
```

Want to undo last commit but keep changes?

```bash
git reset HEAD~1              # Moves commit to working directory
```

Want to undo last commit and delete changes permanently?

```bash
git reset --hard HEAD~1       # Deletes everything
```

Accidentally deleted a branch?

```bash
git reflog                    # Find branch's last commit
git checkout abc1234          # Go to that commit
git switch -c recovered-branch # Recreate branch
```

Best Practices

1. **Commit often with clear messages -** Small, focused commits are easier to understand and revert if

needed.

2. **Never rebase shared commits -** Only rebase local, unpushed commits to avoid conflicts with collaborators.
3. **Use `.gitignore` for sensitive files -** Prevent API keys, passwords, and secrets from being committed.
4. **Pull before you push -** Always sync with remote before pushing to avoid merge conflicts.
5. **Review changes before committing -** Use `git diff` and `git status` to verify what you're committing.
6. **Create branches for features -** Keep main branch stable and working. Develop features in separate branches.
7. **Write meaningful commit messages -** Future you (and your teammates) will appreciate clear explanations.
8. **Don't use `git push --force` on shared branches -** This can delete other people's work.
9. **Commit complete features -** Each commit should represent a logical unit of work that doesn't break the project.
10. **Use `git reflog` as safety net -** If you mess up, reflog can help you recover "lost" commits.