**Git Cheat Sheet**

**What is Git?**

Git is like a time machine for your code! It helps you save different versions of your work, work with friends on the same project, and keep track of all the changes you make.

**Setting Up Git**

**Create a new Git project**

git init

**What it does:** Turns your regular folder into a Git project so you can start tracking your files.

**Example:**

cd my-cool-project

git init

This creates a hidden .git folder that stores all the magic Git information!

**Clone (download) an existing project**

git clone [url]

**What it does:** Makes a copy of someone else's Git project on your computer.

**Example:**

git clone https://github.com/username/awesome-game.git

This downloads the "awesome-game" project to your computer!

**Saving Your Changes**

**Check what files have changed**

git status

**What it does:** Shows you which files you've changed but haven't saved to Git yet.

**Example:**

git status

You might see something like:

Changes not staged for commit:

modified: game.py

Untracked files:

new\_level.py

**Add files to be saved**

git add [file]

**What it does:** Prepares specific files to be saved in Git.

**Examples:**

git add game.py

This tells Git you want to save changes to the game.py file.

git add .

The dot (.) means "add ALL changed files" - be careful with this!

**Save your changes**

git commit -m "Your message here"

**What it does:** Takes a snapshot of your files, saving them in Git history with a message explaining what you did.

**Example:**

git commit -m "Added a cool new jumping feature"

Always write clear messages that explain what you changed!

**Viewing History**

**See the history of saves**

git log

**What it does:** Shows you a list of all the commits (saves) you've made.

**Example:**

git log

This shows all your commits, with the newest ones at the top.

**See a simplified history (one line per commit)**

git log --oneline

**What it does:** Shows a shorter version of the commit history with each commit on a single line.

**Example:**

git log --oneline

You might see something like:

a1b2c3d Added sound effects to jumping

e5f6g7h Fixed bug in enemy movement

i9j0k1l Initial game setup

This makes it much easier to see a quick summary of what changes have been made. It's perfect when you want to get a quick overview without all the extra details!

**See what changed in a file**

git diff [file]

**What it does:** Shows exactly what lines you added or removed in a file.

**Example:**

git diff game.py

Lines with a + were added, lines with a - were removed.

**Working with Branches**

**Create a new branch**

git branch [branch-name]

**What it does:** Creates a separate timeline where you can try out new ideas without messing up your main code.

**Example:**

git branch add-multiplayer

This creates a new branch named "add-multiplayer".

**Switch to a different branch**

git checkout [branch-name]

**What it does:** Switches you to a different branch so you can work on that version of the code.

**Example:**

git checkout add-multiplayer

Now you're working on the "add-multiplayer" branch!

**Create and switch to a new branch in one command**

git checkout -b [branch-name]

**What it does:** Creates a new branch and switches to it right away - it's a shortcut!

**Example:**

git checkout -b fix-bugs

This creates a "fix-bugs" branch and switches to it.

**Combine branches together**

git merge [branch-name]

**What it does:** Takes the changes from another branch and adds them to your current branch.

**Example:**

git checkout main

git merge add-multiplayer

This takes all the cool multiplayer features you built and adds them to your main branch!

**Understanding Merging and Conflicts**

**What is Merging?**

Merging is like combining two different versions of your work together. Imagine you and your friend both colored different parts of the same drawing - merging is putting those parts together into one complete picture!

**Types of Merges:**

1. **Fast-forward merge** - The simplest merge, when no new changes were made to the main branch while you were working on your feature branch.
2. **Automatic merge** - Git automatically combines changes from both branches when they modified different parts of the same file.
3. **Conflict merge** - When you and someone else changed the same lines in the same file, Git needs your help to decide which changes to keep.

**What Are Merge Conflicts?**

Conflicts happen when you and someone else changed the same lines of code in different ways. Git doesn't know which version to keep, so it asks for your help!

**Why Do Conflicts Happen?**

* Two people changed the same line of code
* One person deleted a file while another person modified it
* Two people added different code to the exact same place

**How to Spot a Conflict**

Git will tell you there's a conflict during a merge or pull. The conflicted files will contain special markers that look like this:

<<<<<<< HEAD

This is your code

=======

This is the other person's code

>>>>>>> feature-branch

**How to Fix Conflicts**

1. **Find the conflicts**: Use git status to see which files have conflicts
2. **Open each file**: Look for the <<<<<<<, =======, and >>>>>>> markers
3. **Choose what to keep**: Edit the file to keep the correct code and remove the markers
4. **Save your changes**: Save the file after fixing each conflict
5. **Finish the merge**: After fixing all conflicts, use git add to mark them as resolved, then git commit to complete the merge

**Example of resolving a conflict:**

Original file (game.py):

def player\_jump():

player.y += 10

After you try to merge, you see:

def player\_jump():

<<<<<<< HEAD

player.y += 10

=======

player.y += 15

play\_sound("jump.wav")

>>>>>>> feature-branch

To fix it, edit the file to keep what you want:

def player\_jump():

player.y += 15

play\_sound("jump.wav")

Then:

git add game.py

git commit -m "Merge branch 'feature-branch' and resolve jump height conflict"

**Tips for Preventing Conflicts:**

* Pull changes often from the main branch
* Communicate with your team about who is working on what
* Keep your commits small and focused
* Try to work on different files or different parts of the code
* Use branches for new features

**Working with Remote Repositories (GitHub, etc.)**

**Send your changes to the internet**

git push [remote] [branch]

**What it does:** Uploads your commits to a remote repository (like GitHub) so others can see them.

**Example:**

git push origin main

This pushes your changes on the main branch to the remote called "origin" (usually GitHub).

**Get the latest changes from the internet**

git pull [remote] [branch]

**What it does:** Downloads the newest changes from a remote repository.

**Example:**

git pull origin main

This gets all the new changes your friends pushed to the main branch.

**Add a new remote**

git remote add [name] [url]

**What it does:** Connects your local Git project to a remote repository.

**Example:**

git remote add origin https://github.com/username/my-awesome-game.git

This connects your project to a GitHub repository.

**Fixing Mistakes**

**Undo changes in a file (before adding)**

git checkout -- [file]

**What it does:** Undoes all changes in a file since your last commit.

**Example:**

git checkout -- game.py

This erases all unsaved changes in game.py - be careful!

**Unstage a file (after adding, before committing)**

git reset [file]

**What it does:** Removes a file from the staging area, so it won't be included in your next commit.

**Example:**

git reset game.py

This takes game.py out of the "ready to commit" list.

**Change your last commit (Amending)**

git commit --amend -m "New message"

**What it does:** Updates your last commit with any new changes and/or a new message.

**Example:**

git commit --amend -m "Fixed jumping AND added double jump"

This changes your last commit message and adds any staged changes.

**More About Amending**

Amending is like an eraser for your last commit! Imagine you just took a photo but then realized your finger was in the way. Instead of taking a whole new photo, you can just retake the same one.

**When to use amend:**

* You forgot to add a file to your last commit
* You made a typo in your commit message
* You found a small bug right after committing

**Example workflow:**

git commit -m "Add new enemy character"

# Oops! You forgot to add the enemy's artwork file!

git add enemy-artwork.png

git commit --amend -m "Add new enemy character with artwork"

**Important:** Only amend commits that you haven't pushed to GitHub yet! Amending commits that others have already seen can cause confusion.

**Go back to a previous state**

git reset --hard [commit-hash]

**What it does:** Resets your entire project back to a previous commit - WARNING: this erases history!

**Example:**

git reset --hard 1a2b3c4

This takes your project back to commit "1a2b3c4" - use with EXTREME caution!

**Stashing Changes**

**Save changes temporarily**

git stash

**What it does:** Saves your changes in a special drawer so you can work on something else and come back to them later.

**Example:**

git stash

This hides all your changes and gives you a clean working directory.

**See your list of stashed changes**

git stash list

**What it does:** Shows all your saved stashes.

**Example:**

git stash list

You might see: stash@{0}: WIP on main: 1a2b3c4 Add jumping feature

**Bring back your stashed changes**

git stash apply

or

git stash pop

**What they do:**

* apply brings back your stashed changes but keeps a copy in the stash
* pop brings back your stashed changes and removes them from the stash

**Example:**

git stash pop

This brings back your most recent stash.

**Apply a specific stash**

git stash apply stash@{n}

**What it does:** Brings back a specific stash from your list.

**Example:**

git stash apply stash@{2}

This brings back the stash at position 2 in your stash list.

**Delete a stash**

git stash drop stash@{n}

**What it does:** Removes a stash from your list.

**Example:**

git stash drop stash@{0}

This deletes the most recent stash.

**When to use stash:**

* You're in the middle of working on something but need to switch to another task
* You want to try an idea but aren't ready to commit it yet
* You need to pull changes but don't want to commit your current work
* You want to save your work without creating a formal commit

**Example workflow:**

# You're working on a new feature

# Boss asks you to fix an urgent bug

git stash save "My half-finished feature"

git checkout main

# Fix the urgent bug

git add .

git commit -m "Fix urgent bug"

git checkout feature-branch

git stash pop

# Continue working on your feature

**Helpful Tips**

1. Use git status often to see what's going on
2. Make small, focused commits - don't try to save too many changes at once
3. Write clear commit messages that describe what you did
4. Create new branches for new features or experiments
5. Pull before you push to avoid conflicts
6. If you get stuck or confused, don't panic! Git can almost always help you recover your work

**Git Workflow Summary**

1. git pull to get the latest changes
2. Make your changes to the code
3. git status to see what files changed
4. git add your changed files
5. git commit -m "Message" to save your changes
6. git push to share your changes

# ⚙️ Configuration

**git config --global user.name "Name"**Sets your name for Git commits globally.  
*Example: git config --global user.name "Alice"*

**git config --global user.email "email"**Sets your email for Git commits globally.  
*Example: git config --global user.email "alice@example.com"*

**git config --list**Lists all the Git configuration settings currently in use.

# 💣 Danger Zone

**git push origin <branch> --force**Forces Git to push even if it would overwrite changes. Use this carefully.

**git push origin --delete <branch>**Deletes the remote branch.  
*Example: git push origin --delete feature-login*

Now you're a Git wizard! 🧙‍♂️