1/ git status

command syntax: git status

gives information on the current status of a git repository and its contents

2/ git init

command syntax: git init

Use git init to create a new git repository

Before we can do anything git-related, we must initialize a repo first

3/ The basic git workflow

Work On Stuff (make new files, edit files, delete files, etc)

-> Add Changes (Group specific changes together in preparation of committing)

--> Commit (Commit everything that was previously added)

3a/ git add

command syntax: git add file1 file2

command syntax: git add .

We use the git add command to stage changes to be committed.

It's a way of telling Git, "please include this change in our next commit"

Use git add to add specific files to the staging area. Separate files with spaces to add multiple at once.

use git add . to stage all changes at once

3b/ git commit

command syntax: git commit -m "my message"

Running git commit will commit all staged changes. It also opens up a text editor and prompts you for a commit message

We use the git commit command to actually commit changes from the staging area

When making a commit, we need to provide a commit message that summarizes the changes and work snapshotted in the commit

3c/ Working Directory -> git add -> Staging Area -> git commit -> Repository

4/ Atomic Commits

When possible, a commit should encompass a single feature, change or fix. In other words, try to keep each commit focused on a single thing

This makes it much easier to undo or rollback changes later on. It also makes your code or project easier to review

5/ Amending Commits

Suppose you just made a commit and then realized that you forgot to include a file! Or may be, you made a typo in the commit mesage that you want to correct.

Rather than making a brand new separate commit, you can "redo" the PREVIOUS commit using the --amend option.

(JUST ONE COMMIT AGO - PREVIOUS COMMIT)

Command syntax: git commit -m 'some commit'

git add forgotten\_file

git commit --amend

6/ Ignoring files

We can tell Git which files and directories to ignore in a given repository, using a .gitignore file.

This is useful for files you know you never want to commit, including:

+ Secrets, API Keys, Credentials, etc...

+ Operating System Files

+ Log files

+ Dependencies and packages

.gitignore

Create a file called .gitignore in the root of a repository. Inside the file, we can write patterns to tell Git which files and folders to ignore:

+ .DS\_Store will ignore files named .DS\_Store

+ folderName/ will ignore an entire directory

+ \*.log will ignore any files with the .log extension

7/ Branches

Think of branches as alternative timelines for a project

They enable us to create separate contexts where we can try new things, or even work on multiple ideas in parallel.

If we make changes on one branch, they do not impact the other branches (unless we merge the changes)

7a/ The Master branch

In git, we are always working on a branch. The default branch name is master

It doesn't do anything special or have fancy powers. Its just like any other branch

7b/ HEAD

HEAD is simply a pointer that refers to the current "location" in your repository. It points to a particular branch reference.

So far, HEAD always point to the latest commit you made on the master branch, but some we'll see that we can move around and HEAD will change!

7c/ Viewing Branches

Use git branch to view your existing branches. The default branch in every git repo is master, though you can configure this.

Look for the \* which indicates the branch you are currently on.

Command Syntax: git branch

7d/ Creating Branches

Use git branch <branch-name> to make a new branch based upon the current HEAD.

This just creates the branch. It does not switch you to that branch (the HEAD stays the same)

Command Syntax: git branch <branch-name>

7e/ Switching Branches

Once you have created a new branch, use git switch <branch-name> to switch to it

7f/ Another way of switching??

Historically, we used git checkout <branch-name> to switch branches. This still works.

The checkout command does a million additional things, so the decision was made to add a standalone switch command which is much simplier.

You will see older tutorials and docs using checkout rather than switch. Both now work.

Command syntax: git checkout <branch-name>

7j/ Creating and switching

Use git switch with the -c flag to create a new branch AND switch to it all in one go.

Remember -c as short for "create"

Command Syntax: git switch -c <branch-name>

7h/ Delete Branches

Command Syntax: git branch -d <branch-name>

Force Delete Branches

Command Syntax: git branch -D <branch-name>

8/ Merging

Branching makes it super easy to work within self-contained contexts, but often we want to incorporate changes from one branch into another!

We can do this using the git merge command

The merge command can sometimes confuse students early on. Remember these two merging concepts:

+ We merge branches, not specific commits

+ We always merge to the current HEAD branch

8a/ To merge, follow these basic steps:

+ Switch or checkout the branch you want to merge the changes into (the receiving branch)

+ Use the git merge command to merge changes from a specific branch into the current branch

Command syntax: To merge the bugfix branch into master:

+ git switch master

+ git merge bugfix

Depending on the specific changes you are trying to merge, Git may not be able to automatically merge.

This results in merge conflicts, which you need to manually resolve.

When you encouter a merge conflict, Git warns you in the console that it could not automatically merge.

It also changes the contents of your files to indicate the conflicts that it wants you to resolve.

Conflict Markers:

The content from your currenty HEAD (the branch you are trying to merge content into) is displayed between the <<<<HEAD and =====

The content from the bra

nch you are trying to merge from is displayed betwwen the ====== and >>>> symbols.

==> Resolving Conflicts:

Whenever you encouter merge conflicts, follow these steps to resolve them:

+ Open up the file(s) with merge conflicts

+ Edit the file(s) to remove the conflicts. Decide which branch's content you want to keep in each conflict. Or keep the content from both.

+ Remove the conflict "markers" in the document

+ Add your changes and then make a commit.

9/ Git Diff

9a/ git diff

We can use git diff command to view changes between commits, branches, files, our working directory and more!

We often use git dif alongside commands like git status and git log, to get a better picture of a repository and how it has changed over time.

Command Syntax: git diff

(Compares staging area and working directory)

Without additional options, git diff lists all the changes in our working directory that are NOT staged for the next commit.

9b/ git diff HEAD

git diff HEAD lists all changes in the working tree since your last commit.

Command Syntax: git diff HEAD

9c/ git diff --staged

git diff --cached

git diff --stage or --cached will list the changes between the staging area and our last commit.

"Show me what will be included in my commit if i run git commit right now"

Command Syntax:

+ git diff --staged

+ git diff --cached

9d/ Diff-ing Specific Files

We can view the changes within a specific file by providing git diff with a filename

Command Syntax:

+ git diff HEAD [filename]

+ git diff --staged [filename]

9e/ Comparing Branches

git diff branch1..branch2 will list the changes between the tips of branch1 and branch2

Command Syntax: git diff branch1..branch2

9f/ Comparing commits

To compare two commits, provide git diff with the commit hashes of the commits in question.

Command Syntax: git diff commit1..commit2

10/ Stashing

Git provides an easy way of stashing these uncommitted changes so that we can return to them later, without having to make unnecessary commits.

10a/ git stash save

git stash is super useful command that helps you save changes that you are not yet ready to commit. You can stash changes and then come back to them later.

Running git stash will take all uncommitted changes (staged and unstaged) and stash them, reverting the changes in your working copy.

Command syntax: git stash

You can also use git stash save instead

10b/ git stash pop

Use git stash pop to remove the most recently stashed changes in your stash and re-apply them to your working copy.

Command Syntax: git stash pop

10c/ git stash apply

You can use git stash apply to apply whatever is stashed away, without removing it fromn the stash.

This can be useful if you want to apply stashed changes to multiple branches.

Command Syntax: git stash apply

10d/ Stashing Multiple times

You can add multiple stashes onto the stack of stashes. They will all be stashed in the order you added them.

Command Syntax:

+ git stash

do some other stuff

+ git stash

do some other stuff

+ git stash

10e/ View stashes

run git stash list to view all stashes

Command Syntax: git stash list

10f/ Applying specific stashes

git assumes you want to apply the most recent stash when you run git stash apply, but you can also specify a particular stash like git stash apply stash@{2}

Command Syntax: git stash apply stash@{2}

10j/ Dropping stashes

To delete a particular stash, you can use git stash drop <stash-id>

Command Syntax: git stash drop stash@{2}

10h/ Clearing the stash

To clear out all stashes, run git stash clear

Command Syntax: git stash clear

11/ Undoing Changes and Time Traveling

11a/ Checkout

We can use checkout to create branches, switch to new branches, restore files, and undo history!

We can use git checkout commit <commit-hash> to view a previous commit

Remember, you can use the git log command to view commit hashes. We just need the first 7 digits of a commit hash.

Command Syntax: git checkout d8194d6

\*\*\*

Detached HEAD

You have a couple options:

a/ Stay in detached HEAD to examine the contants of the old commit. Poke around, view the files, etc.

b/ Leave and go back to wherever you were before - reattach the HEAD.

c/ Create a new branch and switch to it. You can now make and save changes, since HEAD is no longer detached.

\*\*\*

11b/ Checkout

git checkout supports a slightly odd syntax for referencing previous commits relative to a particular commit.

HEAD~1 refers to the commit before HEAD (parent)

HEAD~2 refers to 2 commits before HEAD (grandparent)

This is not essential, but i want to mention it because it's quite weird looking if you've never seen it

Command Syntax: git checkout HEAD~1

11c/ Discarding changes

Suppose you've made some changes to a file but do not want to keep them. To revert the file back to whatever it looked like when you last committed, you can use:

git checkout HEAD <filename> to discard any changes in that file, reverting back to the HEAD.

Command Syntax: git checkout HEAD <file>

11d/ Another Option

Here's another shorter option to revert a file..

Rather than typing HEAD, you can substitute -- followed by the file(s) you want to restore

Command syntax: git checkout -- <file>

11e/ Restore

git restore is a brand new Git command that helps with undoing operations.

Because it is so new, most of the existing Git tutorials and books do not mention it, but it is worth knowing!

Recall that git checkout does a million different things, which many git users find very confusing.

git restore was introduced alongside git switch as alternatives to some of the uses for checkout.

11e1/ Unmodifying files with restore

Suppose you've made some changes to a file since your last commit. You've saved the file but then realize you definitely do NOT want those changes anymore!

To restore the file to the contents in the HEAD, use git restore <file-name>

Command Syntax: git restore <file-name>

NOTE: The above command is not "undoable" if you have uncommited changes in the file, they will be lost!

11e2/ Unmodifying Files with Restore

git restore <file-name> restores using HEAD as the default source, but we can change that using the --source option.\

For example, git restore --source HEAD~1 home.html will restore the contents of home.html to its state from the commit prior to HEAD. You can also use a particular commit hash as the source

Command Syntax: git restore --source HEAD~1 app.js

11e3/ Unstaging Files with Restore

If you have accidentally added a file to your staging area with git add and you do not wish to include it in the next commit, you can use git restore to remove it from staging.

Use the --staged option like this: git restore --staged app.js

Command Syntax: git restore --staged <file-name>

12/ Undoing commits with git reset

12a/ Git Reset

Suppose you've just made a couple of commits on the master branch, but you actually meant to make them on a separate branch instead.

To undo those commits, you can use git reset.

git reset <commit-hash> will reset the repo back to a specific commit. The commits are gone.

Command syntax: git reset <commit-hash>

12b/ Reset --hard

If you want to undo both the commits AND the actual changes in your files, you can use the --hard option.

for example, git reset --hard HEAD~1 will delete the last commit and associated changes.

Command syntax: git reset --hard <commit-hash>

13/ Reverting Commits with Git Revert

13a/ Git Revert

git revert is similar to git reset in that they both "undo" changes, but they accomplish it in different ways.

git reset actually moves the branch pointer backwwards, eliminating commits.

git revert instead creates a brand new commit which reverses/undo the changes from a commit. Because it results in a new commit, you will be prompted to enter a commit message.

Command syntax: git revert <commit-hash>

Note: Which one should i use?

Both git reset and git revert help us reverse changes, but there is a significant difference when it comes to collaboration.

+ If you want to reverse some commits that other people already have on their machine, you should you revert.

+ If you want to reverse commits that you haven't shared with others, use reset and no one will ever know!