Introduction to Git

In this presentation…

- What is Git?
- Review of Slides for demo
- Demo using Git in the terminal
- Working in a Team: Common workflows
- Other common scenarios
- More things to know

In this presentation…

- We concentrate on the basics and provide insight into what is possible
- We will present a common workflow
- Avoid the more advanced features
- Everyday-use slides have a light green background

Before we begin

- If you’re viewing these slides after lecture, they will be dense, and are intended to serve as a reference
- We build Git from the most basic commands
- We will introduce some less common but useful features too

Introducing Git

- One of the most common VCS in use today
- Efficient for small to medium size files (i.e. code!)
  - Popular alternative: Perforce, often used in game dev
- Operates by adding a subfolder to your repository (the folder containing your code), and storing any data it needs in that subfolder
- Git != GitHub: Git is a tool that manages a local repository. GitHub is a place to store that repository that provides some management tools

Get More Help and Insight

Here’s a recommended*

GitHowTo

where you can learn more about using Git.

*Rahul Suresh

Get Started

- If you have not already done so
- Download and install Git
    - Installation Tutorial and
    - Configuration
- Create a new, empty folder, and name it whatever you want
- Open up a terminal in the folder you created
- Run `git init`

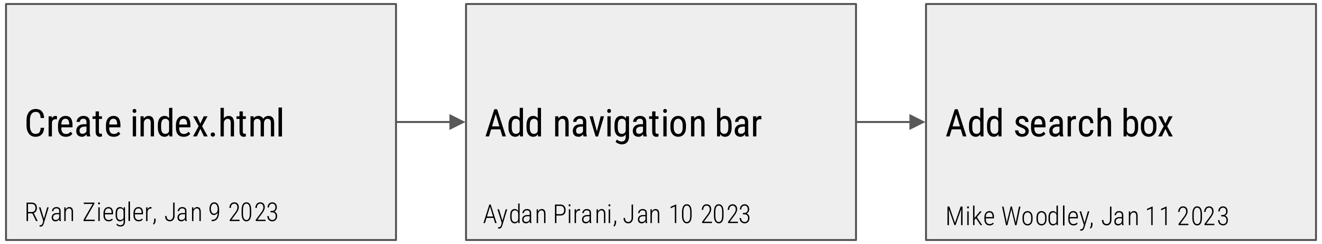
Get Started

- Executing “`git init`” creates a folder called `.git`, we’ll cover what every subfolder does later
  - If you’re in VSCode, you might have to go into settings and remove `.git` from “Files: Exclude” option. (this is a non-destructive change)

```
config
description
HEAD
hooks/
info/
    exclude
objects/
    info/
    pack/
refs/
    heads/
    tags/
```

What is version control?

- Keep track of snapshots of your code: complete replicas at a given point in time
  - Extra information such as author, time, etc
- Why? Easy to undo a change, see when a bug was introduced, etc
- *Facilitates teamwork*: teammates can work in parallel and merge their snapshots together later



What is Git?

- Not just a convenient way for you to turn in homework
- Not just a way to keep backups so that you can return to earlier versions
- Individual users can benefit from the history that is maintained so they can easily "roll back time" to a previous version, if needed
- It IS a tool that tracks changes in source code over time, enabling collaboration and safeguarding code
- Individual users can benefit from a history of their changes so they can easily "roll back time" to an earlier version

What is Git?

- For teams, it allows multiple people to work on the same code simultaneously without overwriting each other’s work
    - If you’ve ever shared documents and found your contributions were changed by a collaborator then you understand how this might be useful
- Detects and alerts to “collisions” (merge conflicts) of changes by different team members

What is Git?

- May have both local and remote “repositories”
- This allows users to work on local copies of a project without disturbing the “best version” of the code
- As you develop new code or repair existing code, you will save intermediate versions of your project ***locally*** as you make changes. This allows team members to work in parallel.
    - commit

What is Git?

- When you join an existing project, you can easily get an entire copy of the project in its current form
  - clone
- As your project progresses, you will be able to update your local copy to the current state of the project or of any previous version
  - Pull - merges changes
  - Fetch - does not merge changes
- When you have a completed feature or repair, you will integrate finished changes into the remote (shared) repository
  - Push - integrates your code via a Git command
  - Pull request - is not a direct Git command. Your team manages this process with aid of GitHub.

Git Basics

- **`git init`**: Initialize a new Git repository.
- **`git clone`**: Clone an existing repository from a remote source like GitHub.
- **`git add`**: Stage changes for the next commit.
- **`git commit`**: Save a snapshot of the project with a descriptive message.
- **`git status`**: Check the status of changes in the working directory and staging area.
- **`git log`**: View the commit history.

Slide Review Followed by Demo

I will review the slides for the live demo and then go to the terminal and show you all of the steps in the slides.

Live Terminal: Get Started

- After installing Git, open a terminal
- Create a project directory and make it your default
  - mkdir proj_git
  - cd proj_git

- Create a text file containing anything. Remember “echo” and “>”.
  - echo this is my first file for this project>file.txt

- Try doing anything without first running “git init” to see what happens.
  - git hash-object -w file.txt

*Note error message! Must use “git init” first.

Live Terminal: Get Started

- Let’s do it right now.
    - git init
    - git hash-object -w file.txt

We now have

1) Created a project directory;
2) Made the project directory our working directory;
3) “Made a change”;
4) created a Git object;
5) calculated the SHA-1 hash of a file. The -w option write the file to Git’s object database.

*for most new GIt users, you do not need to know how to manually create objects

Live Terminal: Your first Git object

- Objects are snapshots of files or folders
- Let’s modify **file.txt** by appending something to it.
- (this creates a new object and new subdirectory in .git/objects)
  - echo append this to file.txt>>file.txt
  - git hash-object -w file.txt

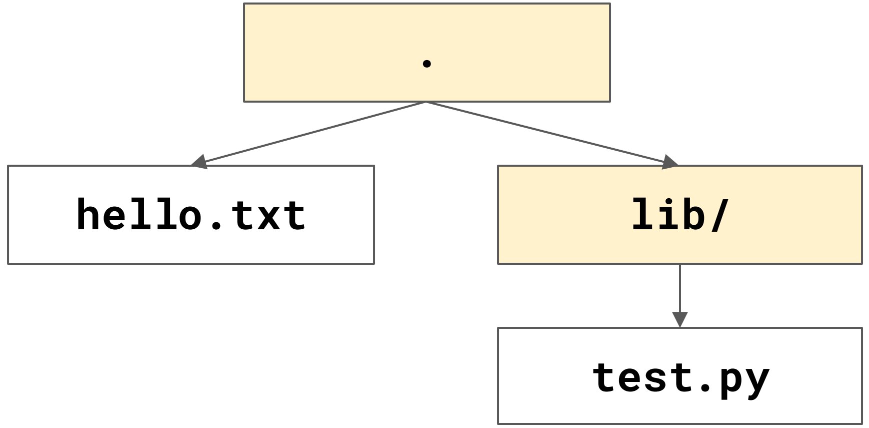
- Note the difference in the hashes
- (shortcut: you really only need the first 4+ chars of the hash)
  - git cat-file [hash1] -p * -p is pretty-print
  - git cat-file [hash2] -p

- Now look in .git/objects
  - ls .git/objects *notice first two chars of directory names

Supporting subfolders

- GIt would be much more useful if it did more than manage files
- What *is* a folder? /Users/mwoodley/Desktop
  - A tree! Recall we can define a tree recursively: a tree is either a leaf, or a non-leaf node with at least one child tree
  - Nodes correspond to folders, and leaves to files

```
hello.txt
lib/
    test.py
```



Let’s do something useful

- When working on a project, we have a workflow that includes making changes and then integrating those changes.
- When using Git, we “commit” to keep a running set of snapshots as our code progresses.
- Best Practice: Commit often.
    - How often? Any change you don’t want to have to reconstruct
    - You can clean up things quite easily at a later time.
- So let’s commit something.

Workflow for commit

1. Make changes
2. Stage changes
3. Review changes
4. Commit changes

1: Make Changes (and use subfolders)

- Live Terminal:
- Make a subfolder and put something in it
  - mkdir lib
  - echo data file in a subfolder>lib/data.txt

2: Stage changes

- Live Terminal:
- Staging is preparing changes to be included in the next commit
- The staging area is where you put files when you’re ready to take a snapshot. You can put as many files as you want here.
    - `git add file.txt`
    - `git add lib/data.txt`

3: Reviewing changes

- Live Terminal:
- See what files are staged:
  - `git status`

- Review changes:
  - `git diff —-cached`

4: Committing

- Live Terminal:
- Let’s make a commitment (commit)
    - git commit -m "added text files to project"

Altogether now (and let’s see the DIFFerence)

- Live Terminal:
- Let’s make a commitment (commit)
  - echo append this to file.txt>>file.txt
  - git add file.txt
- See what files are staged:
  - git status
- Review changes:
  - git diff –cached
- Commit:
  - git commit -m "appended text to file.txt to demonstrate diff"

Now that you know the basics …

Of working with git at the command line, let’s look at some example workflows you will be encountering while working on your project.

Working in a team: Example Workflows

- Clone
- Checkout
- Pull
- Checkout vs pull
- Branch
- Merge

Working in a team: Existing repo (clone)

- Get the URL for your project repository
- Clone the repository
  - git clone https://github.com/username/repository.git
- Navigate into the project directory
  - cd repository
- Pull the latest changes
  - git pull origin main
- Make changes, stage, and commit them
- Follow team procedure for integrating changes
  - Pull request, code review, etc.

Working in a team: Existing repo (checkout)

working on the main branch and want to sync your local version with the latest changes from the remote main branch:

- Fetch the latest changes:
  - git fetch origin
- Switch to the main branch:
  - git checkout main
- Pull the latest changes from the remote main branch:
  - git pull origin main
- If there are conflicts, resolve them, then commit the changes:
  - git add <file-with-conflicts>
  - git commit
- Check the status to ensure everything is synced:

Working in a team: Syncing with existing repo (pull)

- Navigate to your project directory
  - cd /path/to/your/local/repository
- Check the current branch
  - git branch
- Fetch the latest changes from the remote*
  - git fetch origin
- Pull the latest changes and merge them
  - git pull origin main

*fetch followed by pull allows you to review the changes before merging

Working in a team: checkout vs pull

Feature	`git pull`	`git checkout`
Main Purpose	Fetches and merges changes from the remote repository	Switches branches, restores files, or reverts to a commit
When to Use	Syncing with remote changes made by others	Switching branches or working with older commits/files
Scope	Only affects the current branch	Can switch between branches or revert individual files
Involves Remote Repository	Yes, updates your local branch with the remote version	No, works only locally unless switching to a remote branch
Merges Changes	Yes, automatically merges remote changes into local	No, does not merge or fetch new changes

Working in a team: Going your own way (Branch)

- Check current branch
  - git branch
- Pull latest changes from the current branch (main)
  - git pull origin main
- Create a new branch
  - git branch feature-login
- Switch to the new branch
  - git switch feature-login
- Verify the branch was created and switched
  - git branch

*we don’t always create a new branch. Branches are for starting something new: feature, significant refactor, etc.

Working in a team: Coming back together (merge)

- Switch to the branch you want to merge to
  - git switch main
- Verify the branch you’re using
  - git branch
- Merge the previous branch into current branch
  - git merge newBranch
- Manually resolve conflicts, then
  - git add <resolved files>
  - git commit

Working in a team: A new standard (setting the HEAD)

In Git, **setting the HEAD** means changing what Git considers the current commit and branch. Normally, HEAD points to the latest commit on the current branch, but there are scenarios where you might want or need to move HEAD manually.

- **git checkout**: Switches HEAD to a branch or specific commit.
- **git reset**: Moves HEAD to a specific commit, optionally resetting the working directory and/or index.
- **git rebase**: Adjusts where HEAD points during the rebase process.
- **git symbolic-ref**: Manually sets HEAD to point to a branch.
- **git update-ref**: Manually sets HEAD to point to a hash.

*Read about: detached HEAD state

Other Types of Workflows: Common Scenarios

- Undo commit
- Rebasing
- .gitignore
- Blame

Oops! Undo! How to undo a commit.

- Undo last commit but keep changes in staging
  - git reset –soft HEAD~1
- Undo and discard changes
  - git reset –hard HEAD~1

There are a number of ways to undo a commit depending on your situation.

Rebasing

Is a way of integrating changes from one branch to another by reapplying the commits from one branch on top of another.

It’s different from merging in that it rewrites the commit history.

Avoiding oversharing

.gitignore

- Setting up a **.gitignore** file to prevent unwanted files (like compiled binaries, temporary files, or IDE configs) from being included in the project. This ensures that only the necessary files are shared in the team.
- .gitignore is a plain text file so you can read it and simply add to it
  a. NEVER share anything confidential (passwords, etc)
  b. Things that change often where the exact content is not pertinent to the project

.gitignore - example

```
# Ignore node_modules directory
node_modules/
# Ignore environment files (often contain sensitive data)
.env
# Ignore log files
*.log
# Ignore build directories and files
/build/
/dist/
```

.gitignore: is the existence of .gitignore sufficient?

- No, simply having a .gitignore file is not enough to ignore files that have already been tracked by Git.
- The .gitignore file works in conjunction with Git to prevent certain files from being **added** to the repository in the first place, but if a file is already being tracked by Git, it will not be automatically ignored just because you add it to .gitignore.
- You’ll need to take additional steps to stop tracking files that have already been committed.

Laying the Blame

Blame

- **Git Blame** (git blame): Find out which commits modified specific lines of a file, which can help trace bugs or understand the history of a piece of code.

Other Likely Scenarios and what you should read about

Accidental Push to the Wrong Branch

- **git cherry-pick**
- **git rebase**.

Unfinished Work Needing Temporary Storage

- **it stash**

Forgot to Pull Before Pushing

- **git pull**
- **git fetch**

Other Likely Scenarios and what you should read about

Want to Combine or Edit Commits Before Pushing

- **git rebase -i**
- **squash**
- **edit**

Need to Apply Changes from Another Branch Without Merging

- **git cherry-pick**

Accidentally Deleted or Lost Work

- **git reflog**

Other Likely Scenarios and what you should read about

Collaborating with Forked Repositories

- **forks**
- **pull requests**
- **git remote add**
- **git fetch**

Maintaining a Clean History

- **git rebase**
- **git squash**

Switching Between Multiple Versions of a Project

- **git bisect**.

Tagging Releases or Milestones

- **git tag**.

Digression: Preparing for a “commit” - Manual Staging

- You can explicitly add or remove files from the staging area using `update-index`
  - `git update-index --add hello.txt`
  - `git hash-object -w lib/test.py`
  - `git write-tree`
    - note the hash it returned

Digression: “Advanced Stages of GIt” - Manually Staging

Previously, we saw that we could manually create an object. We can also manually set up our staging.

This could be useful if you’re working with Git’s index directly in more advanced or custom situations, such as:

- **Custom Scripting**
- **Advanced Index Manipulation**

Digression: Shaking the trees - Exploring the tree snapshot

- Here we are using the hash from the tree from the previous step
  - `git cat-file [hash from the last slide] -p`
    - The output will look like:

      ```
      100644 blob [hash_1] hello.txt
      040000 tree [hash_2] lib
      ```
  - `git cat-file [hash_2] -p`
    - The output will look like:

      ```
      100644 blob [hash_3] test.py
      ```
- We just followed pointers! The hashes either correspond to files (leaves), or to nodes in the directory tree (which, unsurprisingly, Git calls trees)

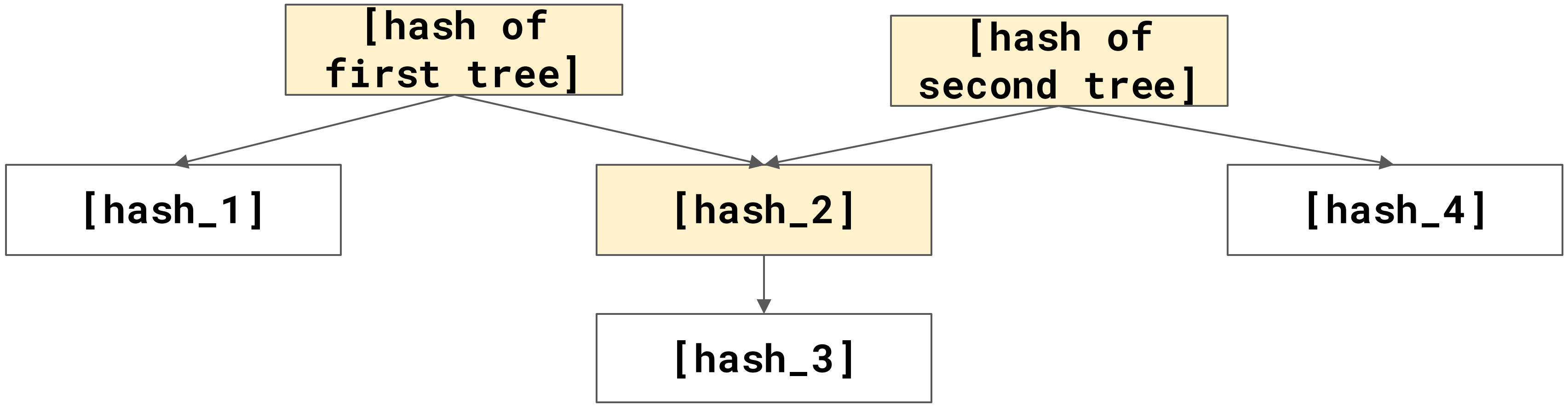
Digression: Modifying the tree

- Update the contents of **hello.txt**, and run
  - git hash-file -w hello.txt
  - `git update-index --add hello.txt`
  - `git update-index --add lib/test.py`
  - `git write-tree`
- the hash returned by write-tree is different than the one it returned before
  - `git cat-file [hash from new write-tree] -p`

has the same structure as before, and *only* the hash for `hello.txt` changed, not the hash for lib!

Digression: Modifying the tree

- `hash_2` -> `lib/`
- `hash_3` -> `test.py`
- `hash_1` -> first version of `hello.txt`
- `hash_4` -> second version of `hello.txt`



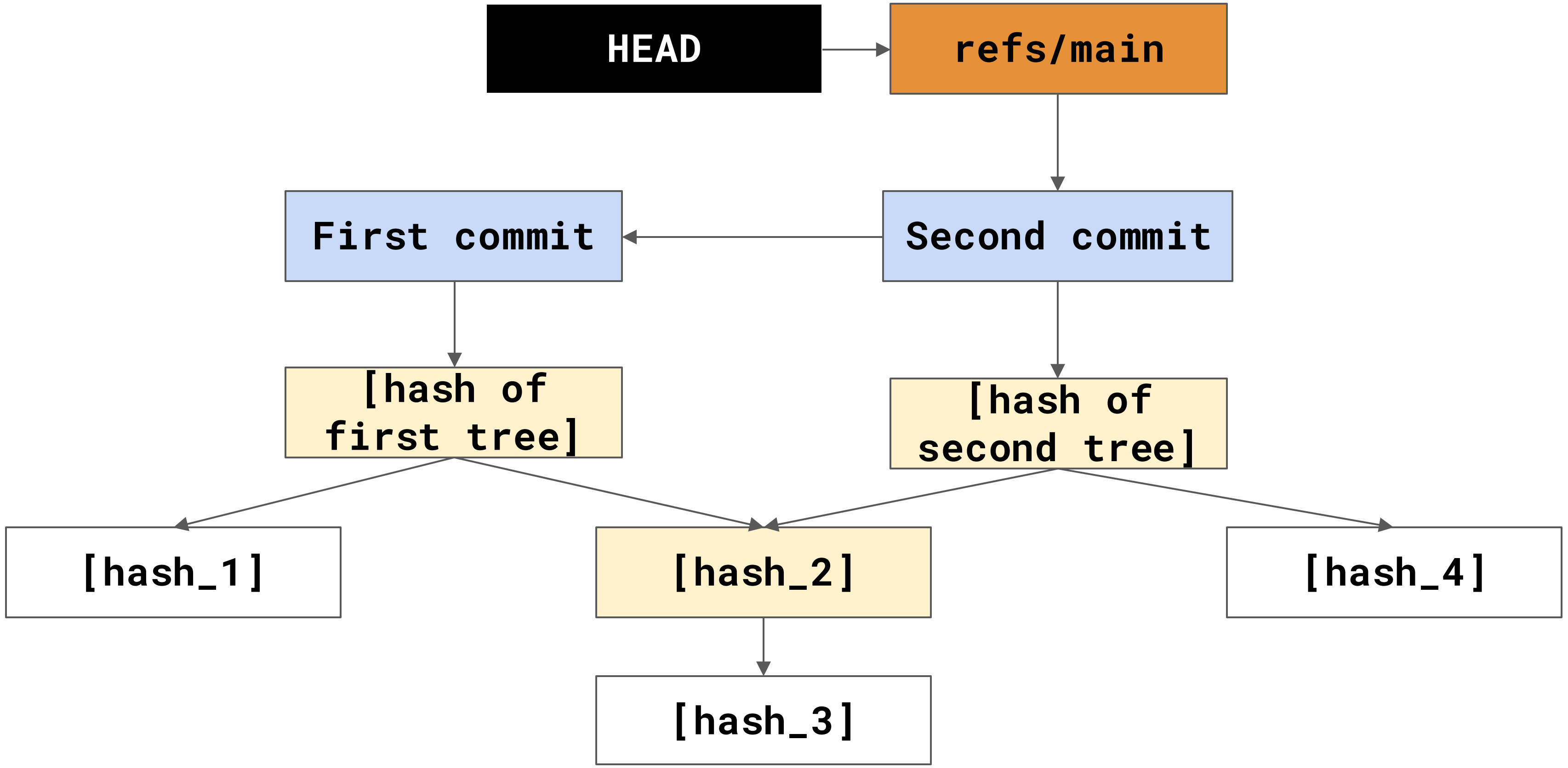
Digression: Packing

- We saw that the tree structure saves space when files aren’t changed, but what happens when a file is changed only a little bit?
    - Don’t want to make a new object, this will duplicate almost everything!
- Solution: packfiles! Every so often (and whenever you send your code over the network) Git will look at objects and compress them into a Packfile
- The Packfile compression includes *deduplication*, meaning that objects that share content will become smaller!
- Packing is transparent to the user, so you don’t have to worry about doing it yourself!
- Packfiles are stored in `.git/objects/pack/` and Git stores any extra metadata in `.git/objects/info/`

Setting the HEAD

- We need to tell Git where the head of our linked list is!
- Run `echo [full hash of second commit] > .git/refs/heads/main`
- This creates a file telling Git that the main reference of our repository is pointing to the second commit (we'll introduce branching and other references later)
- Run `git log main` to see the commit history for the main reference
- If we run git log right now, we'll get an error–Git doesn't know what reference it should be using. Tell it to use the main one by replacing the contents of `.git/HEAD` with the following:
  ```
  ref: refs/heads/main
  ```
- Now run `git log` to see the commit history we just created!
- *There is only one HEAD!*

Setting the HEAD



Undoing commits

- Undoing a commit is as simple as moving pointers around!
- `*HEAD := pointer that HEAD points to`
- `git reset --hard [commit hash]`
  - Make `*HEAD` point to the provided commit, and update all files to match the tree that commit points to
- `git reset --soft [commit hash]`
  - Make `*HEAD` point to the provided commit, but keep files as they are
- `git reset --hard HEAD~`*n*
  - Make `*HEAD` point to the commit *n* behind it, and update all files to match the tree that commit points to (`HEAD~1` will undo the most recent commit)
- `git reset --soft HEAD~`*n*
  - Make `*HEAD` point to the commit *n* behind it, but keep files as they are

Making things easier

- Manually hashing files and committing trees and updating pointers is a lot of work! Fortunately Git gives us commands that do the legwork for us:
  - `git add [space separated paths to files or directories]`
    - Adds the specified files or directories to the staging areas
  - `git commit -m "[message]"`
    - Hash all the files, and the tree, and create a commit object with the message provided, then update the pointer pointed to by `HEAD` to point to the new commit. After all this, unstage any staged files.
- Modify `lib/test.py` and then run `git add lib/test.py` and `git commit -m “update test”`
- Run `git log` again, and you’ll see your new commit

Diffs and some history

- Commits aren’t useful unless we can see what changed between them
  - For example we want to see what specific changes introduced a bug
- Run **`git diff [old commit hash or ref] [new commit hash or ref]`**
- Git started as a VCS built for the Linux kernel development workflow
  - Developers share patches (see example below, which is output from git diff of first two commits) via email, and can apply them to their local repositories

```
diff --git a/hello.txt b/hello.txt
index 6769dd6..9553a1e 100644
--- a/hello.txt
+++ b/hello.txt
@@ -1 +1 @@
-Hello world!
\ No newline at end of file
+Howdy world!
\ No newline at end of file
```

Best practices for commits

- Commits were originally *emails*, and this has inspired many of the best practices in use today
- Commit messages were (and still are, for Linux Kernel developers) email subjects: they should be short, descriptive summaries of the change
    - Good: “Update driver class to support silent command line option”
    - Bad: “Update driver”
- Commits should also be *small*: ideally <250 lines unless you have good reasons not to (such as a project-wide find and replace)
    - If you can’t summarize the commit into a tweet, it should probably be broken up into smaller commits

Undoing commits

- Undoing a commit is as simple as moving pointers around!
- `*HEAD := pointer that HEAD points to`
- `git reset --hard [commit hash]`
    - Make `*HEAD` point to the provided commit, and update all files to match the tree that commit points to
- `git reset --soft [commit hash]`
    - Make `*HEAD` point to the provided commit, but keep files as they are
- `git reset --hard HEAD~`*n*
    - Make `*HEAD` point to the commit *n* behind it, and update all files to match the tree that commit points to (`HEAD~1` will undo the most recent commit)
- `git reset --soft HEAD~`*n*
    - Make `*HEAD` point to the commit *n* behind it, but keep files as they are

Making a new branch

- Run `git checkout -b [name of branch]`
  - In CS222, you should prefix branch names with your name, for example `ryan/[name of branch]`
- Now we see a new file has been created in `.git/refs/heads/ryan/[name of branch]`: if we open the file, it'll point to the same commit that main points to
- We can see that the HEAD has been updated to point to `refs/heads/ryan/[name of branch]`, meaning any commits we make will change what `ryan/[name of branch]` points to, *not* what main points to
- Modify `lib/test.py`, add it to staging, and create a commit
- Run git log: git will tell you that `ryan/[name of branch]` points to the new third commit, but main still points to the second commit

Switching branches

- Run **`git checkout [name of branch]`**. Git will:
  - Update **`HEAD`** to point to **`refs/[name of branch]`**
  - Modify your files to match the tree that **`refs/[name of branch]`** points to
    - If you've modified files but didn't commit the changes, Git will attempt to keep those modifications when you checkout the other branch, and will tell you if it can't. *We recommend that you always commit before switching branches unless you are very experienced!*
- Let's switch back to main by running **`git checkout main`**
- If we run **`git log`**, we'll see that the third commit isn't shown
- Make another new branch, and on it, modify **`hello.txt`** and make a commit with those changes

Merging branches

- We now have two branches with changes—this is similar to what would happen if two people were working on different parts of the project at once
- Switch to the main branch (`git checkout main`)
- Merge the first branch into main by running `git merge [name of first branch]`, and you'll see the following output:

```
Updating 4f4e5e9..8d6e41d
Fast-forward
 lib/test.py | 2 +-
 1 file changed, 1 insertion(+), 1 deletion(-)
```

- Run `git log`, and we'll see that the main pointer points to the same thing as the `[name of first branch]` pointer

Digression: how merging works

- TLDR: merging is hard, for full detail read the docs: https://git-scm.com/docs/merge-strategies
    - First check for files being moved/renamed
    - Next check in-file changes
        - If same file changed by both branches, attempt to apply both changes
        - If not possible, let user resolve the conflicts
            - This is why we suggest you don’t work on the same file on multiple branches at the same time
    - If there were conflicts, the merge changes will be saved as a singular commit, otherwise it is treated as copying all the commits from source to target

Rebasing branches

- The second branch we created is now “behind” main (1 commit ahead, 1 commit behind)
- Switch to this branch (**`git checkout [name of second branch]`**) then run **`git rebase main`**. Git will:
  - Move the ref for this branch back to the commit where the branch began
  - Apply all commits from main
  - Rewrite all commits on the current branch to be as if they branched off of the most recent commit from main

Finishing up, team workflows

- Switch back to main (`git checkout main`) and merge the second branch (`git merge [name of second branch`])
- Once you're done with a branch, you can delete it with `git branch -D [name of branch]`--this amounts to simply deleting the ref file!
- Generally you aren't working on the same computer as the rest of your team. This is where GitHub comes in: it is a platform that can store the .git folder, and facilitates merging through a feature called Pull Requests, with support for code reviews by teammates.

One more thing: `.gitignore`

- **There are some files you should not track with git: *never ever ever* commit files containing passwords, API keys, etc**
  - Sensitive data should be stored in `.env` files: check language/library docs for details on how to access data from `.env` files
- Best practice to avoid tracking dependencies (such as `node_modules` folder): keeps repository size down
- `.gitignore` file specifies files and folders to ignore

Git tips and tricks

- When working in a team, you should create a new branch for each new bugfix, feature, etc. Branches should be self-contained.
- You should avoid modifying the same file on two different branches at the same time, doing this will cause merge difficulties!
- Commit messages should be short and descriptive
- More commits is better than fewer: save your progress often!
- Experience is your friend: if you’re new to Git, it might be a lot at first, but as you use it you’ll understand more
    - Feel free to ask any Git questions in office hours, we’re here to help!

Summary - part 1

- Set up git for your project by running git init in project folder
- Add files to staging area with **`git add [space separated list of files and/or folders]`**
- Create a commit with **`git commit -m "[message]"`**
    - Messages should be short and descriptive (no longer than a tweet), commits should modify < ~250 lines as a rule of thumb (but there are exceptions, use your best judgement)

Summary - part 2

- Check out a new branch with `git checkout -b [branch name]`
  - All branches in 222 should start with your name
- Check out an existing branch with `git checkout [branch name]`
- See what branch you're currently on with `git branch`
- Compare two commits/ref with `git diff [first thing] [second thing]`
- Merge branch `b` into branch `a` by running `git merge b` on branch `a`
- Rebase branch `b` onto branch `a` by running `git rebase b` on branch `a`
- Delete branch [name] by running `git branch -D [name]`

The folders we didn't cover

- There's a few files and subfolders in `.git` that this presentation doesn't cover because CS222 won't use them
  - `hooks/`: contains files that run when certain git actions occur, for example running a script to make sure tests pass before you make a commit
  - `info/`: contains the exclude file, which has the same function as `.gitignore`. `.gitignore` is most commonly used, so don't use this
  - `description`: file containing a description of the repository. You don't need to fill this out
  - `config`: file containing git configuration options. You probably don't need to edit this

Useful resources

- Git SCM book: https://git-scm.com/book (main reference used to make this guide)
- Git cheatsheet: https://gist.github.com/rzig/a37930960417055cea942dc8ddc18469
- Oh shit git: https://ohshitgit.com/