CS 45, Lecture 8

Version Control

Akshay Srivatsan, Ayelet Drazen, Jonathan Kula

Winter 2023

Outline

Contents

1 Review

Computer Networks

Last lecture, we saw:

- How computers can use a network to talk to each other
- How information gets sent from one place to another over the internet

In this lecture, we will see:

- How to safely store your files (code or text)
- How to collaborate on files with others over the internet
- *How to avoid losing all your homework!*

Files

- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files

- As you change these files over time, you'll eventually want some way to keep track of different "versions" of the file.
- What we need is a "version control system".

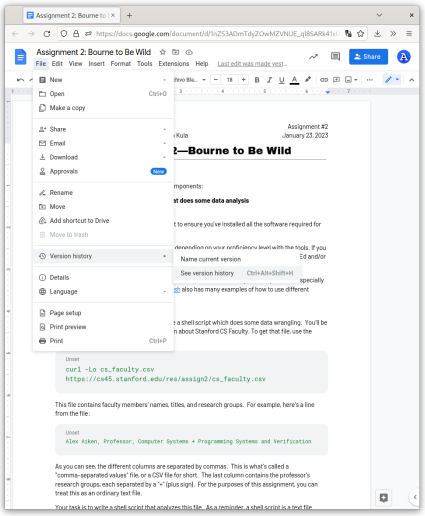
2 Version Control

2.1 Version Control Systems

Version Control Systems

- A VERSION CONTROL SYSTEM (VCS) is a piece of software which manages different versions of your files and folders for you.
- A good VCS will let you look at old versions of files and restore files (or information) which you might have accidentally deleted.
- You've seen these before!

Version Control Systems



Assignment #2
January 23, 2023

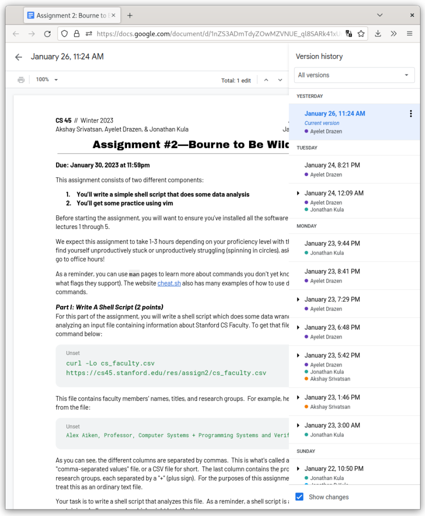
2—Bourne to Be Wild

```
curl -Lo cs_faculty.csv
https://cs45.stanford.edu/res/assign2/cs_faculty.csv
```

This file contains faculty members' names, titles, and research groups. For example, here's a line from the file:

```
Alex Aiken, Professor, Computer Systems + Programming Systems and Verification
```

As you can see, the different columns are separated by commas. This is what's called a "comma-separated values" file, or a CSV file for short. The last column contains the professor's research groups, each separated by a "+" (plus sign). For the purposes of this assignment, you can treat this as an ordinary text file.



CS 45 // Winter 2023
Akshay Srivatsan, Ayelet Drazen, & Jonathan Kula

Assignment #2—Bourne to Be Wilc

Due: January 30, 2023 at 11:59pm

This assignment consists of two different components:

1. You'll write a simple shell script that does some data analysis
2. You'll get some practice using vim

Before starting the assignment, you will want to ensure you've installed all the software lectures 1 through 5.

We expect this assignment to take 1-3 hours depending on your proficiency level with tl find yourself unproductively stuck or unproductively struggling (spinning in circles), ask go to office hours!

As a reminder, you can use `man` pages to learn more about commands you don't yet knc what flags they support). The website cheat.sh also has many examples of how to use d commands.

Part I: Write A Shell Script (2 points)
For this part of the assignment, you will write a shell script which does some data wranc analyzing an input file containing information about Stanford CS Faculty. To get that file command below:

```
curl -Lo cs_faculty.csv
https://cs45.stanford.edu/res/assign2/cs_faculty.csv
```

This file contains faculty members' names, titles, and research groups. For example, he from the file:

```
Alex Aiken, Professor, Computer Systems + Programming Systems and Verif
```

As you can see, the different columns are separated by commas. This is what's called a "comma-separated values" file, or a CSV file for short. The last column contains the prc research groups, each separated by a "+" (plus sign). For the purposes of this assignme treat this as an ordinary text file.

Version Control Systems

A good version control system:

- Will store many versions of your files
- Will let you "revert" a file (or a part of a file) to an older version
- Will track the order of different versions
- Will ensure each "version" is neither too big nor too small

A great version control system:

- Will let you collaborate on files with other people
- Will combine the different versions of the files produced by different people sanely

2.2 Comparison of VCSs

There are many different ways to set up a version control system. Let's see the pros and cons of some of the more common ones.

Google Docs

Google Docs automatically keeps track of file history in a basic VCS.

Pros:

- Great for rich text
- Allows real-time collaboration
- Saved on the cloud automatically[1]

Cons:

- Bad for plain text (especially code)
- Requires an internet connection
- Only supports a single "current" version of a single file

You might wonder why you'd ever want multiple "current" versions of a file. One example is when we are planning out this class—we have multiple ideas for each part of an assignment, and try to flesh them out enough to see what works and what doesn't. It would be nice if we could have multiple, equally-valid, versions of the assignment at the same time, but right now the only way to do that is to copy the file.

Copying Files

You can make a bunch of copies of files or folders with `cp` as a simple form of version control. You can compare versions with `diff`.

Pros:

- Works on either rich or plain text (or anything else)
- It's simple and makes it easy to move data between versions

Cons:

- It's messy and a lot of manual work
- It's hard to tell what the relationship between different versions is
- It takes a lot of hard drive space

Zip Files

Instead of just `cp`ing folders, we could bundle them up into a Zip file (a single file which can be "unzipped" into a folder).

Pros:

- Tracks versions for an entire folder at once
- Easy to share a version with someone else (email)

Cons:

[1]Stay tuned for our lecture on the cloud!

- It's still a lot of manual work
- It's hard to tell what the relationship between different versions is
- It's hard to extract a single file from an old version

Zip Files++

- What if we had a tool which did all this zip file stuff automatically?
- We could tell it to take a "snapshot" of a directory, and it would save all the changes in it.[2]
- We could ask it to recover an old version of a specific file, or to reset everything to an old version to "undo" our work.
- The tool could track the relationships between different versions, so we can have multiple "current" versions at the same time.
- If we want to combine different versions, the tool can automatically do it for us (instead of us copying and pasting the parts together).

Git

`git` is a version control system which tracks "commits" (snapshots) of files in a REPOSITORY.

- Git stores old versions of files in a hidden folder (`.git`), and automatically manages them.
- We can tell Git to keep track of certain files, and tell it when to take a snapshot.
- We can ask Git to go back to an old snapshot (even for a single file).
- We can ask Git to keep track of who's working on what, so multiple people can work on different things without conflicting.
- If we want to combine multiple people's work, we can ask Git to automatically merge them together. If it can't for some reason, it'll ask us to manually merge them.

There are several other version control systems in use, but Git is by far the most popular. Different companies sometimes decide to use different VCSes for various reasons; for example, Facebook/Meta uses a version of Mercurial (another VCS) they call "Sapling". However, these different VCSes are conceptually very similar, so moving from one to another is pretty easy.

3 Git

3.1 Linear History

Basic Workflow

The simplest way to use git is the "linear" workflow, which is the same way you'd use Google Docs:

1. `git init` to enable Git in a certain directory
2. `git add` any files you want Git to "track"
3. `git commit` the currently "staged" changes to save a snapshot
4. make changes to your files
5. `git add` the changed files to "stage" them again

[2]Or, even better, we could tell it *which* files to save in the snapshot. Everything else stays as it was in the previous snapshot.

6. Repeat from 3

You can use `git log` to see your commit history, and use `git status` to see the current state of staged/unstaged/untracked changes.

Before you do anything in Git, you should tell Git who you are:

```
git config --global user.name "<YOUR FULL NAME>"
git config --global user.email "<YOUR PUBLIC EMAIL ADDRESS>"
```

Note that these configuration values will be permanently baked into any commits you make, so other people may be able to see them. Use a pseudonym or a pseudonymous email address if you don't want to publish your information; see the GitHub Docs for more info.

Basic Workflow

Demo

Let's practice how to:

- Create a new Git repository
- Commit a new file
- Commit changes to files
- Revert commits
- Look at an old version of a file
- Compare two versions of files
- See your commit history

3.2 Branching Workflow

Branching Workflow

We can also split our "repo" into multiple BRANCHES, which are like alternate versions of a folder. This means different people can work on different things without interfering with one another.

1. Make sure your repository is "clean" (i.e., you have no uncommitted changes).
2. `git checkout -b <branch>` to create a new branch and move to it; at this point, the new branch will be identical to the old one.
3. Make changes, `git add`, `git commit` as usual
4. `git checkout` to switch between branches

3.3 Combining Branches

Branching Workflow

Combining Branches

Now that we have multiple branches, we probably want to join them back together at some point.

There are several ways to do this:

- `git merge` two branches into one
- `git merge --fast-forward` a long branch onto a shorter version of itself

- `git rebase` one branch onto another branch
- `git cherry-pick` a specific commit from one branch to another

Branching Workflow

Fast Forwarding

The simplest case of merging is called fast-forwarding.

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] \arrow[dr] & X \arrow[r] & Y \arrow[r] & F \\
& & & X \arrow[r] & Y &
\end{tikzcd}$$

Assume there was a main branch containing A and B. We then created a new branch based on B, containing X and Y. Merging the main branch and our new branch is easy—our new branch is just a longer version of the main branch, so we can “fast-forward” the main branch to catch up. We can then commit F on the main branch, based on Y.

Branching Workflow

Merging

Merging (in general) creates a merge commit to join the two branches.

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] \arrow[dr] & D \arrow[r] & E \arrow[r] & M \arrow[r] & F \\
& & & X \arrow[r] & Y \arrow[ur] & &
\end{tikzcd}$$

In this case, assume as before that we had a main branch containing A and B. We created a new branch and added X and Y. However, while we were doing that, someone else added D and E to the main branch. Our branch has now diverged from the main branch; we both have different, possibly contradictory ideas of what the “real” state is. We fix this by creating a merge commit M, which contains a combination of E and Y; if E and Y were contradictory, Git will ask the user to resolve the merge conflict by picking which version to keep. Now we’re back to a single “real” version M, so we can add F on the end.

Branching Workflow

Rebasing

Rebasing moves the “base” of a branch to be a different commit. *Rebasing edits Git’s history to make fast-forwarding possible.*

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] \arrow[dr] & D \arrow[r] & E \arrow[r] \arrow[dr] & X' \arrow[r] & Y' \arrow[r] & F \\
& & & X \arrow[r] & Y & X' \arrow[r] & Y' &
\end{tikzcd}$$

Again assume that we had a main branch containing A and B, from which we created a new branch and added X and Y. Once again, someone else added D and E to the main branch while we weren’t looking. However, this time, we want to maintain the illusion of “linear history”; the idea that every version has exactly one predecessor and one successor. This is obviously not true in this case—the commit C has two successors—so we “rebase” our branch. This creates a new commit X’ which combines X and E, and a new commit Y’ which combines Y and X’. However, if you look at the Git history, you never see a reference to the original X or Y; for all practical purposes, X and Y never existed. Now we’re back to the fast-forward case; our branch is a longer version of `main`, so we can fast-forward `main` to include X’ and Y’. Now we can commit a new F based on Y'; if anyone in the future ever looks at the repository, it'll look like everything up to F was a linear, non-branching, sequence of changes.

Branching Workflow

Cherry-Picking

CHERRY-PICKING copies a *single commit* from one branch to another branch. *CHERRY-PICKING and rebasing is a good way to move a single commit from one branch to another.*

$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] \arrow[dr] & D \arrow[r] & E \arrow[r] & Y' \arrow[r] \arrow[dr] & F & \\
 & & & X \arrow[r] & Y \arrow[r] & Z & X' \arrow[r] & Z'
\end{tikzcd}$$

Sometimes we want to grab a specific change from another branch, without merging or rebasing the entire branch. This is a use case for "cherry-picking", selecting a specific commit to copy into another branch. In this case, assume a similar setup to the previous cases—a main branch containing A, B, C, D, and E, and a new branch containing A, B, X, and Y, and Z. We're on the main branch at E, and we realize we need something that was added in commit Y, but we **don't** want to include commit X or commit Z. We can't literally just copy Y onto our main branch, since it depends on X and doesn't include anything from D or E, so we create a new commit Y'. Y' is like a mashup of Y and E, but without any of the stuff from X. Now we can commit F based on Y', and F will contain no references to X or Z.

Note that this can get weird if you later do a merge—Y and Y' will both be in your history! If you cherry-pick commits like this, you probably want to do a rebase to make your history less confusing; in this case we could rebase the new branch onto Y', creating a new X' and Z', which would effectively erase the original X, Y, and Z from the new branch.

Branching Workflow

When to merge/rebase/cherry-pick?

- **fast-forward** when possible (`git merge --ff-only`).
- **rebase and then fast-forward** if possible, i.e., if you're the only one working on the branch; **never** rebase a branch other people are using (`git rebase` and `git merge --ff-only`).
- **merge** if neither of the above are possible (`git merge`).
- **cherry-pick** if you want to copy a specific commit to another branch (`git cherry-pick`)[3].

Some projects insist upon having linear history in `main`, which means any merge commits will be rejected. In this case, you should first `rebase` your changes onto the most recent `main`, then `merge --ff-only` to fast-forward `main` to include your changes. Most projects are okay with merge commits, so you can just use `merge` and forget that `rebase` ever existed.

Branching Workflow

Branching Demo

Let's practice how to:

- Split our repository into two branches
- Switch between branches
- Make commits on either branch

[3]This is pretty rare, I've only used it a handful of times.

- Merge two branches together

To Be Continued...

We'll pick back up with merge conflict resolution and collaboration in Lecture 9.

Some commands which came up during class:

`git checkout:` essentially means "move to a different commit"; doesn't change your git history

`git reset:` "resets" the entire repository to the way it was in an old commit (and changes git history to match)

`git revert:` "undoes" a specific old commit by creating a new commit that does the opposite

Note that, even though Git commits are technically versions, Git's commands often operate on the *changes* between versions.

You can view all your active branches with `git branch`, and delete branches with `git branch -d`. Remember that a "branch" is basically just a named version of a file; "main" is typically the official version, while other branches are so-called "feature branches" where different people can experiment with adding different things. Generally a feature branch should be "owned" by a single person to avoid conflicts (only that person should ever change that branch).

One last thing: git has separate man pages for each subcommand, since they're each so complicated they need their own docs. For example, the man page for `git commit` can be viewed using `man git-commit`.