Learning Goals

In this (and next) 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 your data!

File Versions

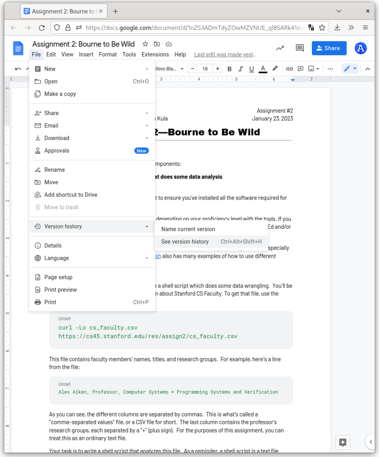
- Many of the files you work with will be text:
  - Source Code
  - Documentation
  - Markup Files
- As you change these files over time, you'll eventually want some way to keep track of different “versions” of the file.
- What we need is a “version control system”.

Outline

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

a Kula

2—Bourne to Be Wild

mponents:

at does some data analysis

t to ensure you've installed all the software required for

depending on your proficiency level with the tools. If you
Ed and/or

specially
sh also has many examples of how to use different

e a shell script which does some data wrangling. You'll be
n about Stanford CS Faculty. To get that file, use the

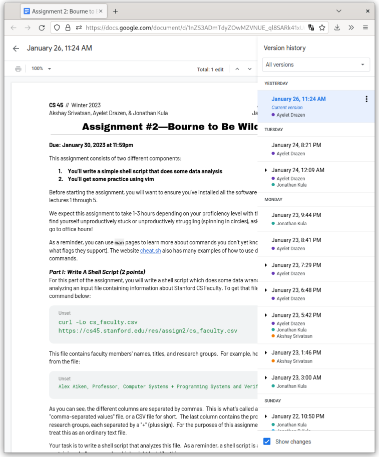
Unset
```
curl -Lo cs_faculty.csv
https://cs45.stanford.edu/res/assign2/cs_faculty.csv
```

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

Unset
```
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.

Your task is to write a shell script that analyzes this file. As a reminder, a shell script is a 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:

Unset
```
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:

Unset
```
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.

Your task is to write a shell script that analyzes this file. As a reminder, a shell script is a

YESTERDAY
- January 26, 11:24 AM — *Current version* — Ayelet Drazen

TUESDAY
- January 24, 8:21 PM — Ayelet Drazen
- January 24, 12:09 AM — Ayelet Drazen, Jonathan Kula

MONDAY
- January 23, 9:44 PM — Jonathan Kula
- January 23, 8:41 PM — Ayelet Drazen
- January 23, 7:29 PM — Ayelet Drazen
- January 23, 6:48 PM — Ayelet Drazen
- January 23, 5:42 PM — Ayelet Drazen, Jonathan Kula, Akshay Srivatsan
- January 23, 1:46 PM — Akshay Srivatsan
- January 23, 3:00 AM — Jonathan Kula

SUNDAY
- January 22, 10:50 PM — Jonathan Kula

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 help you combine “branched” versions of the files produced by different people working independently

Git

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

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.

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.

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).

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.

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.

Outline

Outline

Basic Workflow

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

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

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”

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

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

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

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
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.

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

Outline

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.

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).

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

Outline

Branching Workflow

Combining Branches

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

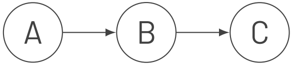
There are several ways to do this:

- `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

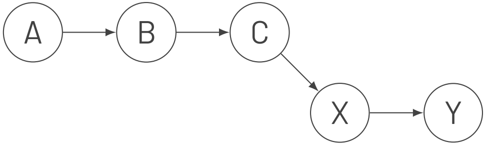
The simplest case of MERGING is called FAST-FORWARDING.


$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

Branching Workflow

Fast Forwarding

The simplest case of MERGING is called FAST-FORWARDING.


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

Branching Workflow

Fast Forwarding

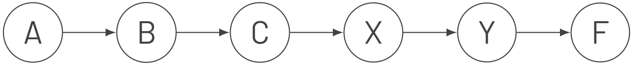
The simplest case of MERGING is called FAST-FORWARDING.

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

Branching Workflow

Fast Forwarding

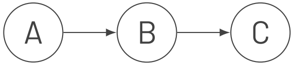
The simplest case of MERGING is called FAST-FORWARDING.


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

Branching Workflow

Merging

MERGING (in general) creates a MERGE COMMIT to join the two branches.

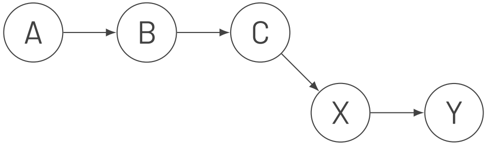


$$\begin{tikzcd}A \arrow[r] & B \arrow[r] & C\end{tikzcd}$$

Branching Workflow

Merging

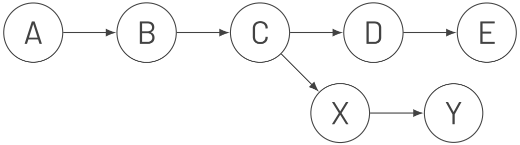
MERGING (in general) creates a MERGE COMMIT to join the two branches.



Branching Workflow

Merging

MERGING (in general) creates a MERGE COMMIT to join the two branches.


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

Branching Workflow

Merging

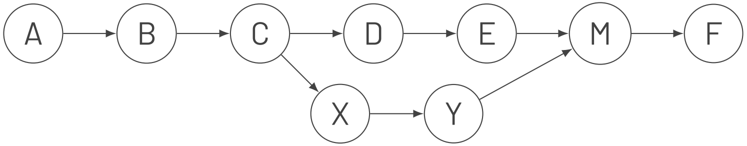
MERGING (in general) creates a MERGE COMMIT to join the two branches.

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

Branching Workflow

Merging

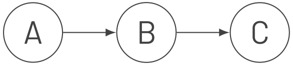
MERGING (in general) creates a MERGE COMMIT to join the two branches.


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

Branching Workflow

Rebasing

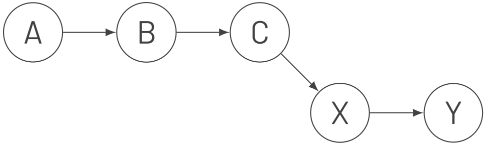
REBASING moves the “base” of a branch to be a different commit.


$$\begin{tikzcd} A \arrow[r] & B \arrow[r] & C \end{tikzcd}$$

Branching Workflow

Rebasing

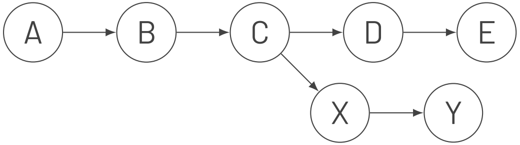
REBASING moves the “base” of a branch to be a different commit.


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

Branching Workflow

Rebasing

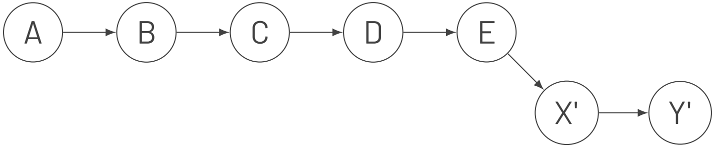
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

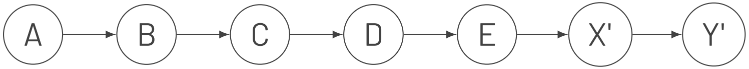
REBASING moves the “base” of a branch to be a different commit.


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

Branching Workflow

Rebasing

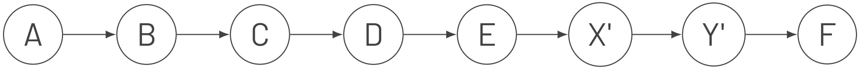
REBASING moves the “base” of a branch to be a different commit.



Branching Workflow

Rebasing

REBASING moves the “base” of a branch to be a different commit.

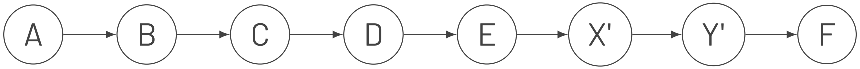


$$\begin{tikzcd}
A \arrow[r] & B \arrow[r] & C \arrow[r] & D \arrow[r] & E \arrow[r] & X' \arrow[r] & Y' \arrow[r] & F
\end{tikzcd}$$

Branching Workflow

Rebasing

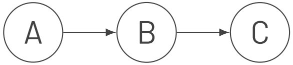
REBASING moves the “base” of a branch to be a different commit. REBASING edits Git's history to make FAST-FORWARDING possible.



Branching Workflow

Cherry-Picking

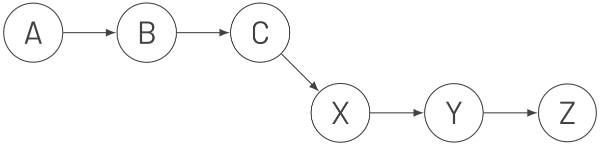
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

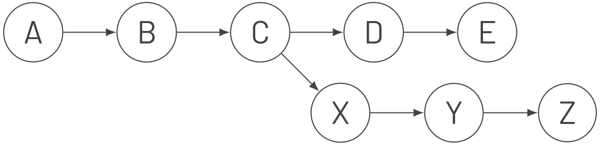
CHERRY-PICKING copies a *single commit* from one branch to another branch.


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

Branching Workflow

Cherry-Picking

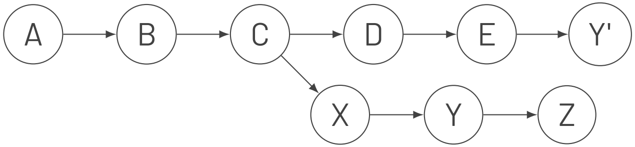
CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

CHERRY-PICKING copies a *single commit* from one branch to another branch.



Branching Workflow

Cherry-Picking

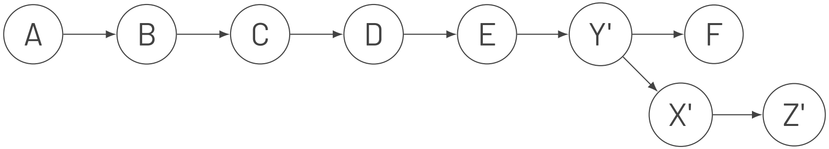
CHERRY-PICKING copies a *single commit* from one branch to another branch.

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

Branching Workflow

Cherry-Picking

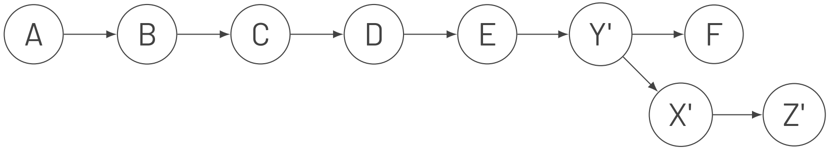
CHERRY-PICKING copies a *single commit* from one branch to another branch.



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.



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`)[1].

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

Branching Workflow

Branching Demo

Let's practice how to:

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

To Be Continued...

We'll pick back up with merge conflict resolution and collaboration in Lecture 10.

Some commands which (probably) 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.