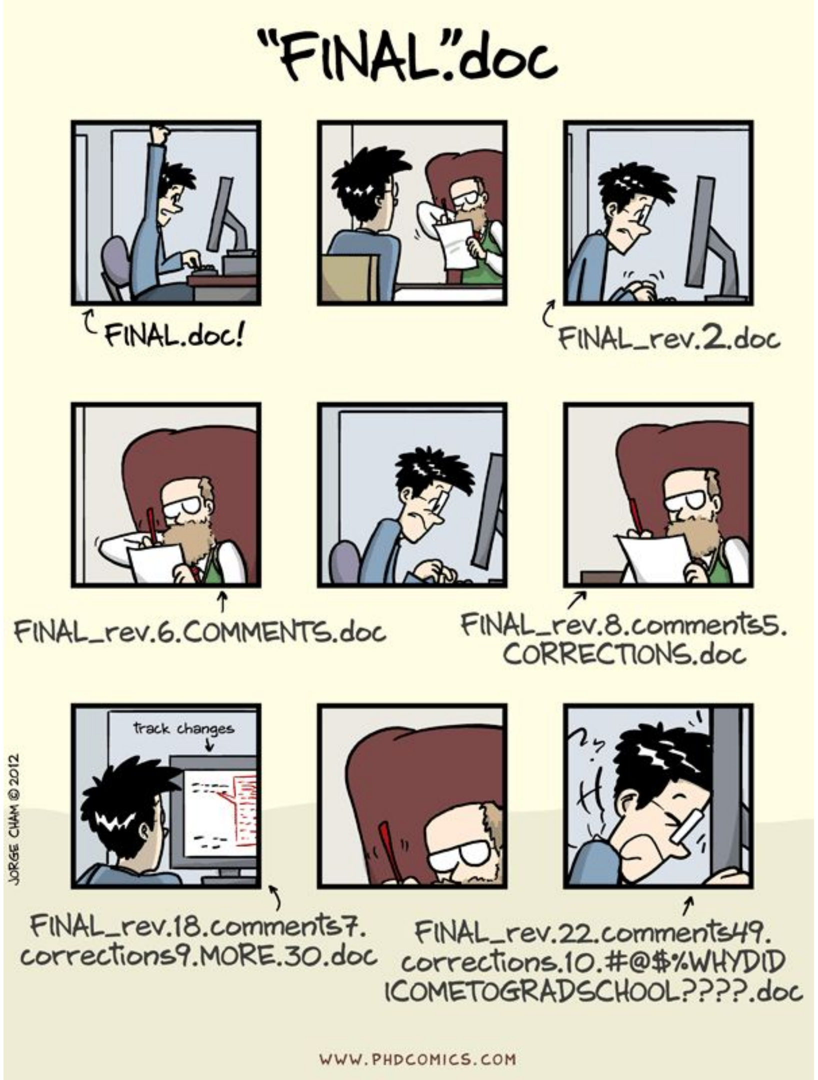
Version Control with Git

Ben Winjum
OARC
January 20, 2022

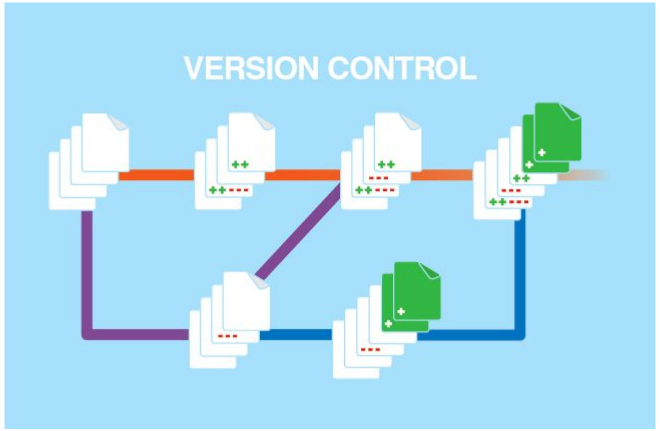
What is version control and why should I use it?


"FINAL".doc
FINAL.doc!
FINAL_rev.2.doc
FINAL_rev.6.COMMENTS.doc
FINAL_rev.8.comments5.
CORRECTIONS.doc
track changes
FINAL_rev.18.comments7.
corrections9.MORE.30.doc
FINAL_rev.22.comments49.
corrections.10.#@$%WHYDID
ICOMETOGRADSCHOOL????.doc
JORGE CHAM © 2012
WWW.PHDCOMICS.COM

Git is an **Open Source Distributed Version Control System**

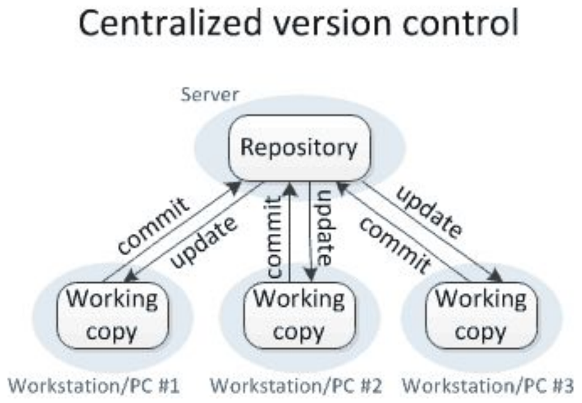
Git is an **Open Source Distributed <u>Version Control System</u>**

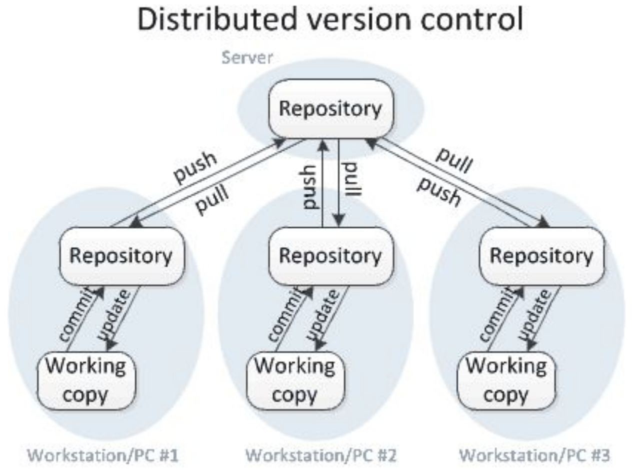
Git stores content and keeps track of the changes.



https://www.linode.com/docs/development/version-control/introduction-to-version-control/1204-image_versioning_intro_1.jpg

Git is an **Open Source** **Distributed Version Control System**





Git is an **<u>Open Source Distributed Version Control System</u>**

Git is released under the GNU General Public License version 2.0, which is an open source license.

The code is freely available at https://github.com/git/git

Initial Setup

First things first

- Git commands are written as `git verb options`
- Example: setting up our initial configuration

```
$ git config --global user.name ”first last"
$ git config --global user.email ”username@ucla.edu"


$ git config --list
```

A few technical details

- We will be interacting with GitHub, and the email address set with config should be the same email used for GitHub.
    - If you are concerned about email privacy, you can use a private email address with GitHub and set the same email address for `"user.email"` (e.g. `username@users.noreply.github.com`)
- Line endings – different operating systems use different characters to represent the end of a line. This may cause unexpected issues comparing files edited on different machines!
    - The following settings are recommended:
    - MacOS and Linux: `$ git config --global core.autocrlf input`
    - Windows: `$ git config --global core.autocrlf true`
    - More info: https://help.github.com/articles/dealing-with-line-endings/

Editor	Configuration command
Atom	`$ git config --global core.editor "atom --wait"`
nano	`$ git config --global core.editor "nano -w"`
BBEdit (Mac, with command line tools)	`$ git config --global core.editor "bbedit -w"`
Sublime Text (Mac)	`$ git config --global core.editor "/Applications/Sublime\ Text.app/Contents/SharedSupport/bin/subl -n -w"`
Sublime Text (Win, 32-bit install)	`$ git config --global core.editor "'c:/program files (x86)/sublime text 3/sublime_text.exe' -w"`
Sublime Text (Win, 64-bit install)	`$ git config --global core.editor "'c:/program files/sublime text 3/sublime_text.exe' -w"`
Notepad++ (Win, 32-bit install)	`$ git config --global core.editor "'c:/program files (x86)/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
Notepad++ (Win, 64-bit install)	`$ git config --global core.editor "'c:/program files/Notepad++/notepad++.exe' -multiInst -notabbar -nosession -noPlugin"`
Kate (Linux)	`$ git config --global core.editor "kate"`
Gedit (Linux)	`$ git config --global core.editor "gedit --wait --new-window"`
Scratch (Linux)	`$ git config --global core.editor "scratch-text-editor"`
Emacs	`$ git config --global core.editor "emacs"`
Vim	`$ git config --global core.editor "vim"`

HELP!!

Always remember that if you forget a Git command, you can access the list of common commands with --help:

```
$ git --help
```

For Git commands, you can see command options by using -h and access the Git manual by using --help :

```
$ git commandname -h
$ git commandname --help
 - or -
$ git help commandname
```

The repository

- A Git repository is a virtual storage of your project.
- It allows you to save versions of your code (or projects), which you can access when needed.

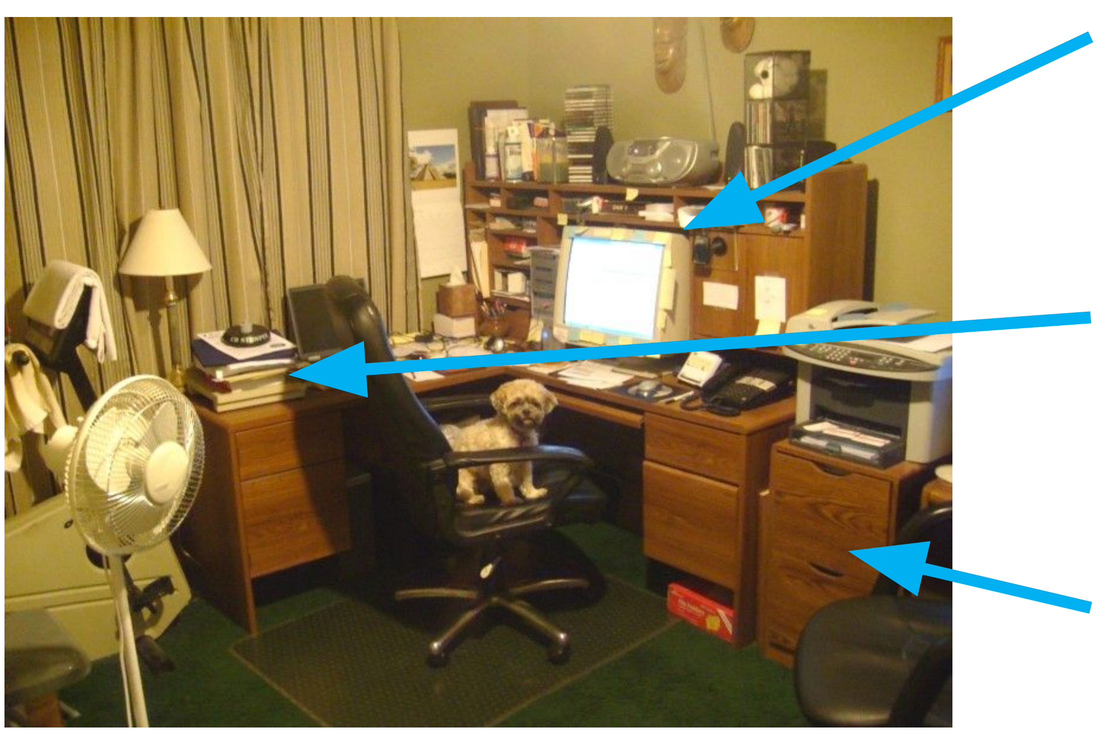
Making a repository

- `git init` is used to initialize a Git repository in the directory in which it is executed
    - `git init dirname` will make a directory called dirname and initialize the repo inside
    - `git init` can be run in subdirectories, but it is redundant and can cause trouble down the road
- .git -- Git will store information about the repository in a hidden directory

Adding files and keeping track of versions

- Recording changes
- Checking status
- Making notes of changes
- Ignoring certain changes

Git is like a desk



https://successfulsoftware.files.wordpress.com/2009/07/officedog2.jpg

git add and git commit

- Adding files to the staging area:
  - `git add file1 file2`
- Moving changes from the staging area into the repository:
  - `git commit –m ‘a short message describing the changes’`
- You can add everything at once with: `git add .`
  - But this is not advised in practice
- You can also commit all changes without adding them: `git commit -a`
  - This is also not advised in practice

git commit for directories

Git tracks files within directories, but not directories

```
$ mkdir newdir
$ git status
$ git add newdir
$ git status
```

If you have files in a directory, you can add them all at once with:

```
$ git add dirname
```

Committing Changes with Git

Suppose you use Git to manage a file named `myfile.txt`. After editing the file, which command(s) below would save the changes to the repository?

1) `git commit -m “my recent changes”`
2) `git init myfile.txt; git commit -m “my recent changes”`
3) `git add myfile.txt; git commit -m “my recent changes”`
4) `git commit -m myfile.txt “my recent changes”`

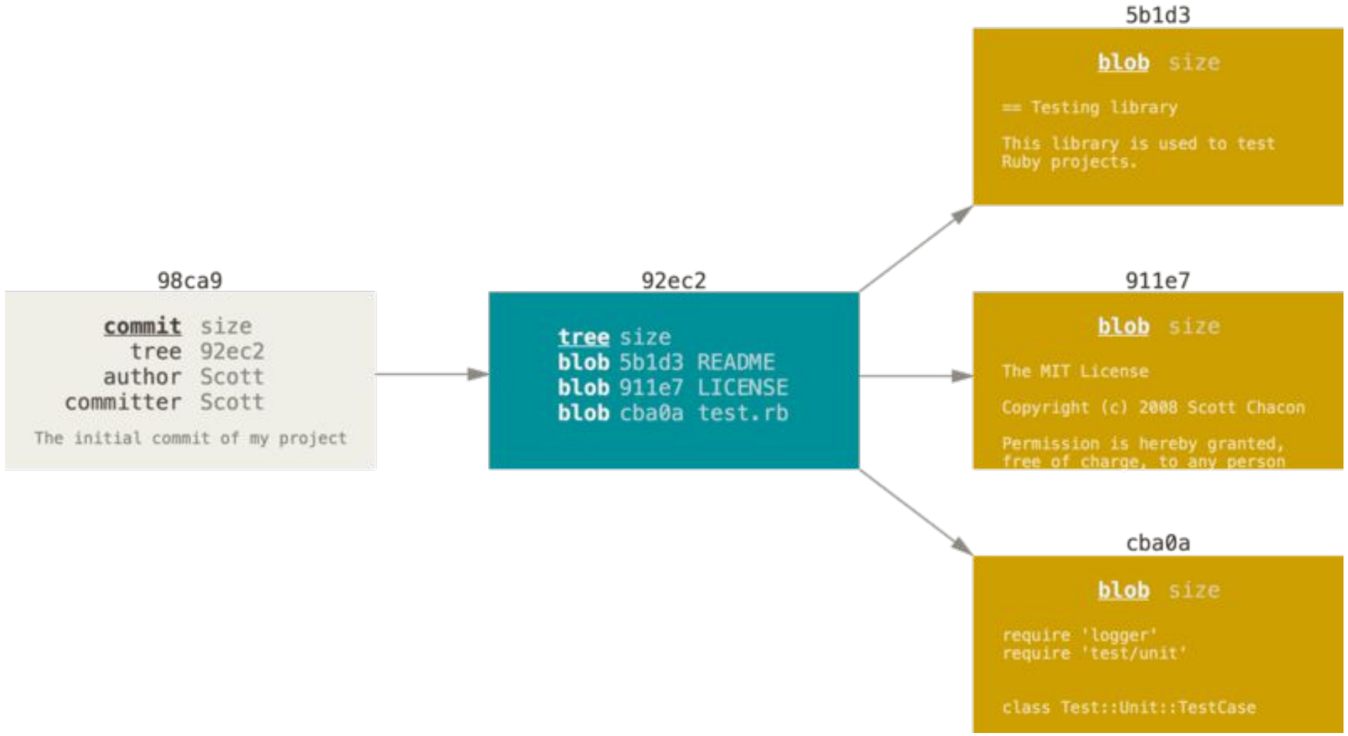
Exercises with `init`, `add`, `commit`

1. Initialize a Git repository
2. Make a new file
3. Commit this file to the repository

Looking at history

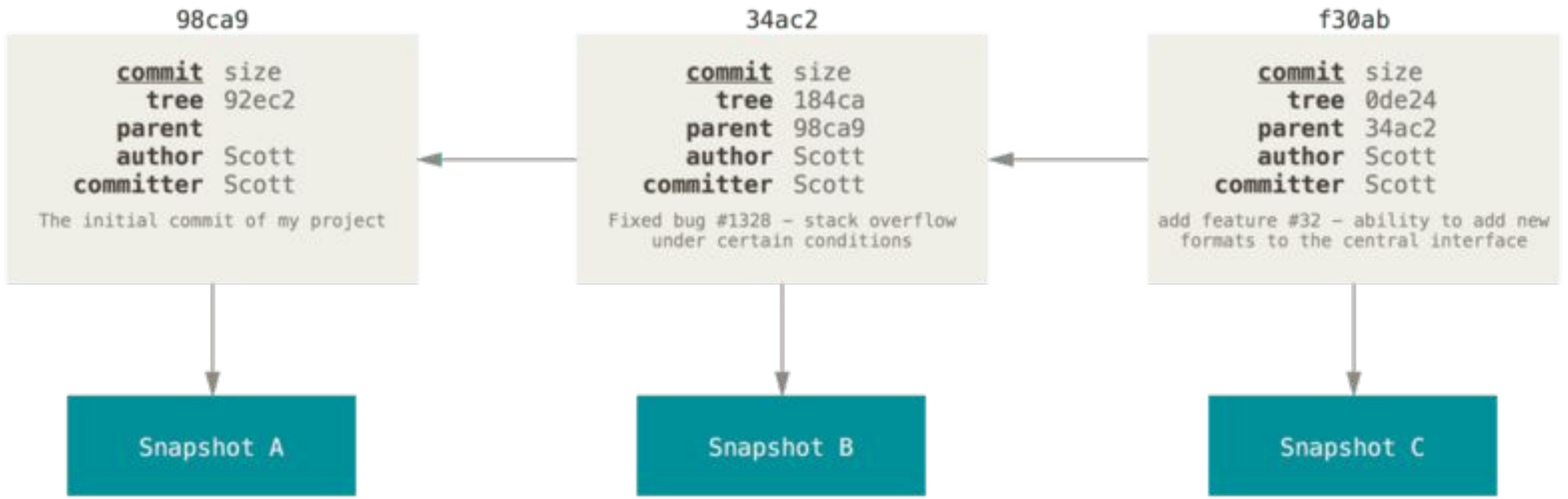
```
$ git log -[N]
$ git log --oneline
$ git log --oneline --graph
```

Files and the commit history



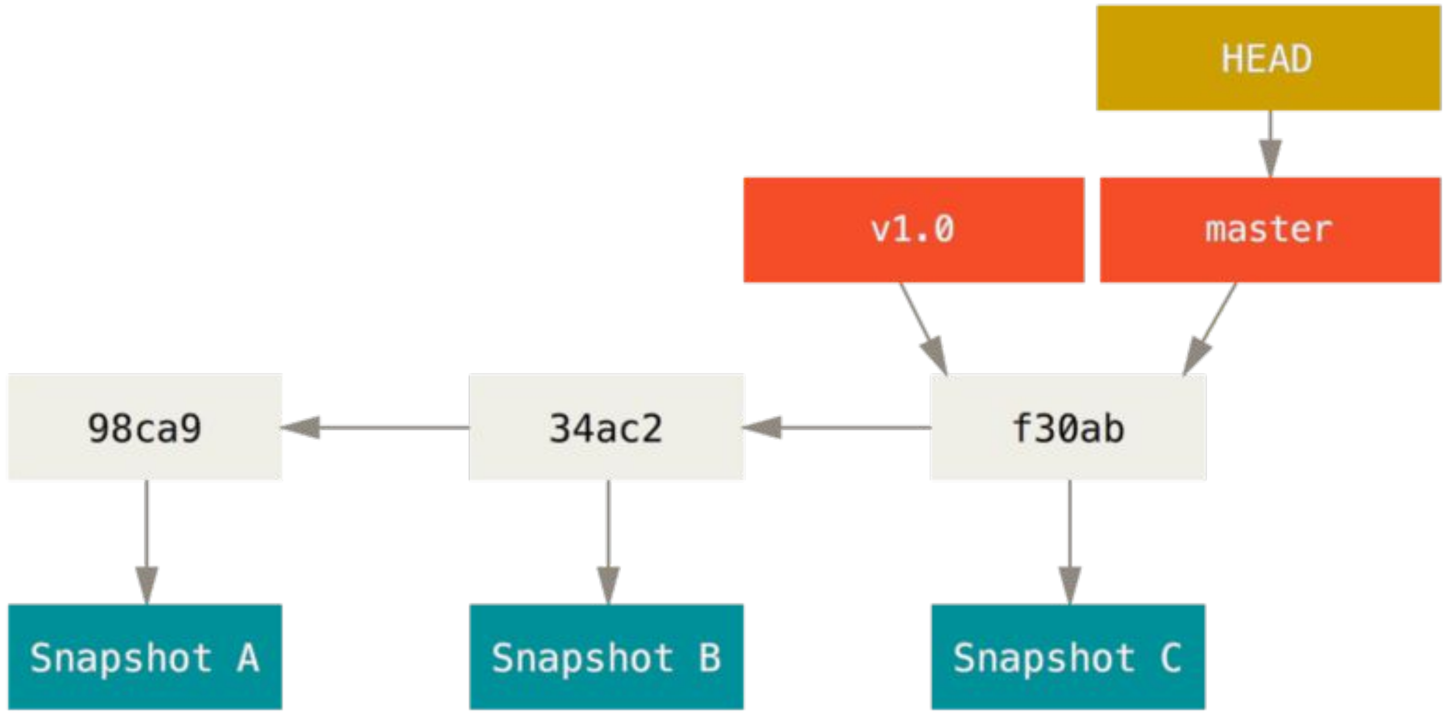
Images taken from the Pro Git book -- freely available online and recommended for further reading
https://git-scm.com/book/en/v2

Files and the commit history



Images taken from the Pro Git book -- freely available online and recommended for further reading
https://git-scm.com/book/en/v2

Pointers to commits

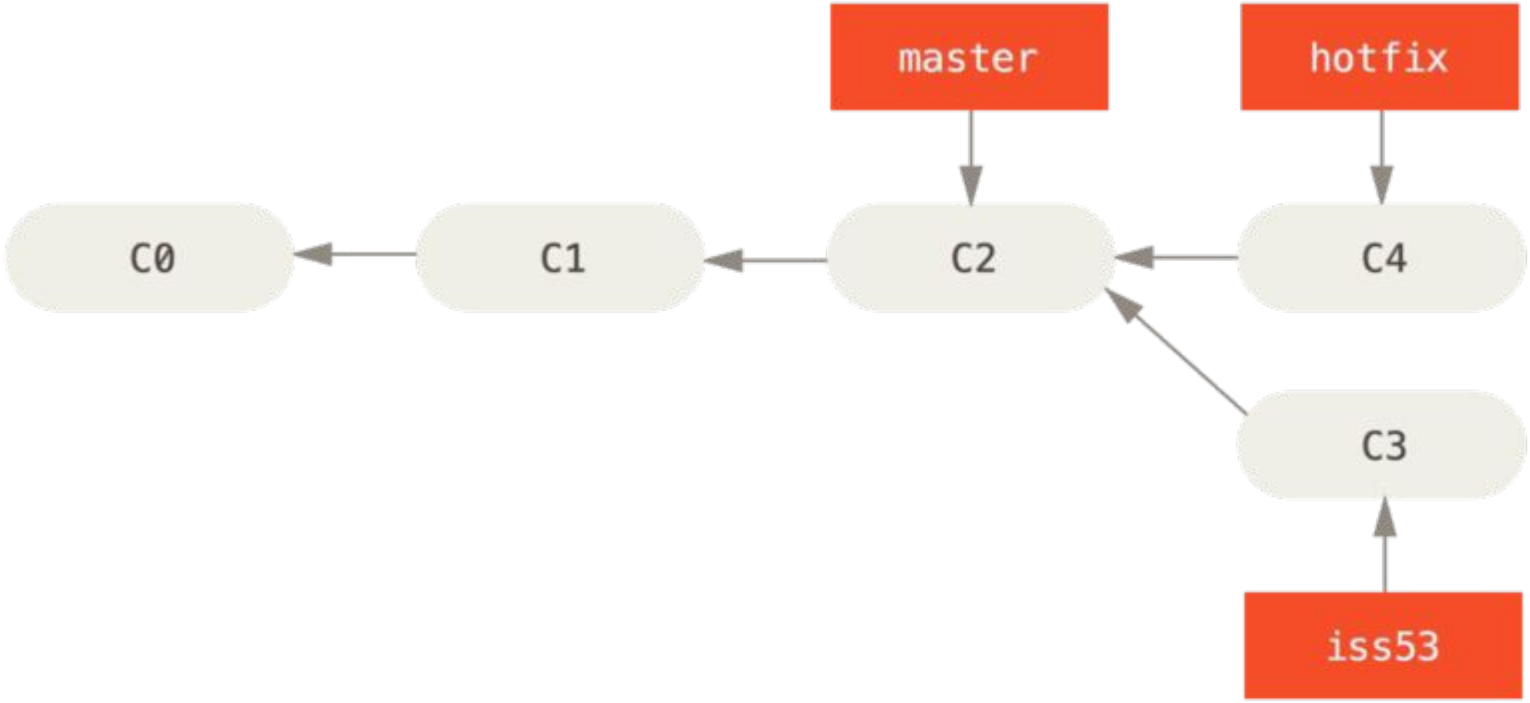


Images taken from the Pro Git book -- freely available online and recommended for further reading
https://git-scm.com/book/en/v2

The tool for visualizing git actions can be found at:

http://git-school.github.io/visualizing-git/

Tracking different development paths via different pointers



Images taken from the Pro Git book -- freely available online and recommended for further reading
https://git-scm.com/book/en/v2

Looking at differences

```
$ git diff
$ git diff --staged
$ git diff filename
```

Exercises with `add`, `commit`, `diff`, and `log`

1. Make a new file (or files) and/or change files in your working area
2. Display the differences between the files’ updated states and the repository’s previously committed state
3. Commit your changes
4. View your version history to confirm

Dealing with past history

- Identifying old versions
- Reviewing past changes
- Recovering old versions

Discerning what changed

You can refer to the most recent commit of the working directory by using the identifier HEAD

HEAD~N refers to the Nth parent commit relative to HEAD

```
$ git diff HEAD
$ git diff HEAD~3 HEAD~1
```

To look at changes that were made in a commit rather than differences between commits, you can use:

```
$ git show
```

git checkout

- Reverting files to a previous state
  - Can be done relative to HEAD
  - Can be specified with 40-digit identifier
  - Can be specified with smaller amount of initial digits
- Beware that the following are different commands!

```
$ git checkout HEAD~1 fname.txt
$ git checkout HEAD~1
```

One reverts fname.txt to a previous state, the other detaches HEAD!

Say that a user has made errors when editing their current version of analysis.py and wants to recover the last committed version of the file. Which command should she use?

1. $ git checkout HEAD
2. $ git checkout HEAD analysis.py
3. $ git checkout HEAD~1 analysis.py
4. $ git checkout <unique ID of last commit> analysis.py
5. Both 2 and 4

Pulling some ideas together – what gets printed to the screen?

```
$ echo "Venus is beautiful" > venus.txt

$ git add venus.txt

$ echo "Venus is too hot to be suitable as a planetary base" >> venus.txt

$ git commit -m "Comment on Venus as an unsuitable base"

$ git checkout HEAD venus.txt

$ cat venus.txt #this will print the contents of venus.txt to the screen
```

Ignoring changes

- Sometimes there may be files you don’t want to track
  - Log files, temporary files, cached data analysis files….
- You can use the ‘.gitignore’ file to tell Git not to track files

Git + collaboration (using GitHub as example)

- Git already includes the machinery to move work between two repositories
- In practice, a central repository is usually used as a master copy
    - GitHub, BitBucket, GitLab…
- These hosting services also offer tools to facilitate collaboration
    - Web-based – anyone with an internet connection can view the repo
    - Wikis, task management, bug tracking, feature requests

Git + collaboration (using GitHub as example)

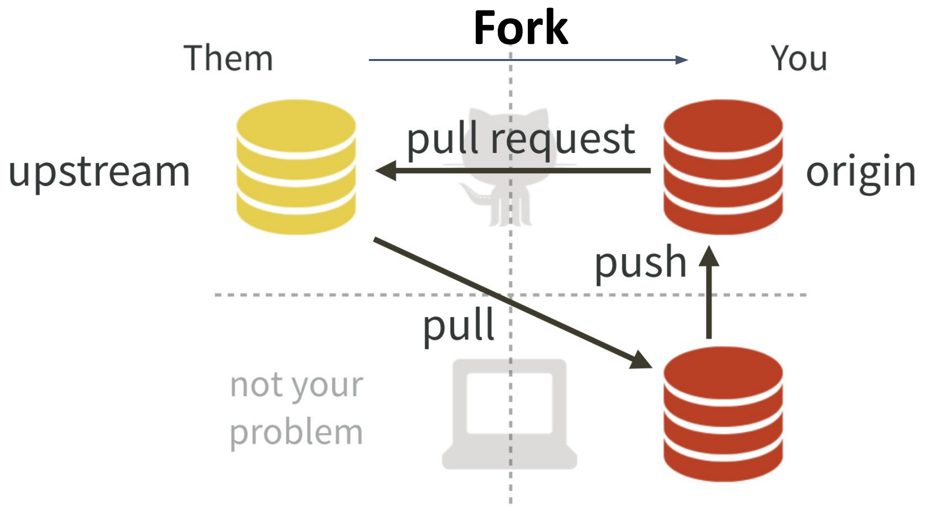
- Let’s make a new repository on GitHub

Git + collaboration (using GitHub as example)

- Let’s make a new repository on GitHub
- After making it, we need to link our local repository with the remote repository
    - We’ll use HTTPS – you are encouraged to investigate configuring the SSH option later

    -
      ```
      git remote add origin
      https://github.com/username/reponame.git
      ```

Final bit of common git activity: Forks and Pull Requests



get changes from the main repo

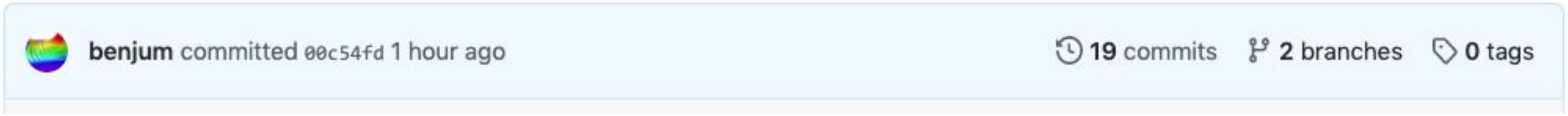
https://happygitwithr.com/upstream-changes.html

Git + collaboration (using GitHub as example)

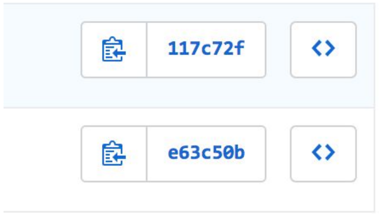
- Exchanging changes between repositories
  - `git push`
  - `git push origin main`
  - `git push -u origin main`
  - `git pull`
  - `git pull origin main`

GitHub exercise

- Look at your new repository on GitHub and find the bar that looks similar to this:



- Click on commits, and then find three buttons per line that look like:



- What do these buttons do when you click them? And how can you do something similar in the terminal?

Collaborative workflow

- The basic collaborative workflow:
  - update your local repo with `git pull origin main`
  - make your changes and stage them with `git add`
  - commit your changes with `git commit -m`
  - upload the changes to GitHub with `git push origin main`
- It is better to make many commits with smaller changes rather than one massive commit with lots of changes
  - Small commits are easier to read and review

GitHub exercise

- Since we’re all remote and it’s tricky to coordinate efforts…. Let’s pretend we are each two people

- Clone your own GitHub repository into a new directory

- Part1-of-you: Make a couple changes to the local repo, and push changes back to GitHub
- Part2-of-you: Pull the changes that Part1-of-you made into your local copy of the repository

- Switch roles and repeat

Dealing with conflicts

- You will inevitably encounter scenarios in which you make local commits and try to push them to a remote repository, only to discover that a collaborator has updated the repository too

- Git will recognize potential conflicts and refuse to accept a push request

    - Solution: pull changes from the remote repo, merge them locally, and push again

GitHub exercise in conflict

- Part1-of-you: Commit changes to the files from your GitHub repository and push the changes to GitHub
- Part2-of-you:
  - Commit *different* changes to the *same* files in your local repository and (<u>after Part1-of-you has pushed their changes)</u> try to push your changes to GitHub too
  - Resolve the conflict by pulling, merging, and pushing

- Switch roles and repeat

Some final thoughts: If you run into lots of conflicts

- Conflict resolution costs time and effort and can introduce errors if conflicts are not resolved correctly. If you find yourself resolving a lot of conflicts, consider these technical approaches:
    - Pull from upstream more frequently, especially before starting new work
    - Use topic branches to segregate work, merging to main when complete
    - Make smaller more atomic commits
    - Where logically appropriate, break large files into smaller ones so that it is less likely that two authors will alter the same file simultaneously
- Conflicts can also be minimized with project management strategies:
    - Clarify who is responsible for what areas with your collaborators
    - Discuss what order tasks should be carried out in with your collaborators so that tasks expected to change the same lines won’t be worked on simultaneously
    - If the conflicts are stylistic churn (e.g. tabs vs. spaces), establish a project convention that is governing and use code style tools (e.g. htmltidy, perltidy, rubocop, etc.) to enforce, if necessary

Final item:
please complete the brief
post-course survey

Any Future Questions:
Email – bwinjum@oarc.ucla.edu