

# TERMINAL, GIT, AND GITHUB

# Today's Learning Objectives

- Describe the importance of reproducibility in coding projects
- Enter basic commands in terminal to navigate a file system, manipulate files, and run scripts
- Maintain proper version control for a coding project using git
- Collaborate on a coding project with Github
- Share your research work and maintain bibliographies through Latex and Zotero

# | Agenda

- Reproducibility and Terminal (30 mins)

# **Reproducibility**

**Motivation for today's class**  
**Adapted from slides by Sam Gilmour and the Turing Way**

# What is reproducibility?

“The ability to obtain consistent results no matter the machine a project runs on.”

		Data	
		Same	Different
Analysis	Same	Reproducible	Replicable
	Different	Robust	Generalisable

# One possible reproducibility nightmare

- Imagine you created a new optimization model and algorithm for a problem and ran many computational experiments without keeping in mind reproducibility.
- Months later, you finish writing your paper on it and stumble upon a new model that was wayyyyy better and faster. Welp... **Time to re-run everything!**
  - Spend a day trying to get code set up to run again
  - But new results didn't match up with original results
  - Eventually, you track down a single parameter that had changed
  - Change the parameter and re-run everything again
  - Would take **weeks of work** and huge amounts of **cluster resources**

# What is reproducibility good for?



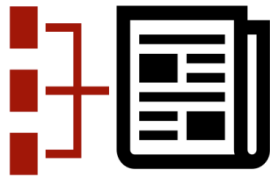
Track Project History



Collaborate & Review



Avoid Misinformation



Write Paper Efficiently



Get Credits Fairly



Ensure Continuity

# Tools for reproducibility

- **Version control** – maintain a history of your project to better track changes and decisions
- **Testing** – simple tests to detect errors and changes in functionality when new features are added
- **Maintaining a reproducible computational environment** – for example, software and package versions
- Making code and data **open and available** when possible



# Tools for Reproducible Research

**Git** - Version control system



- Command line tool
- Used to make and document local code changes (and communicate those changes with Github)

**Github** - Hosting platform for version control and collaboration



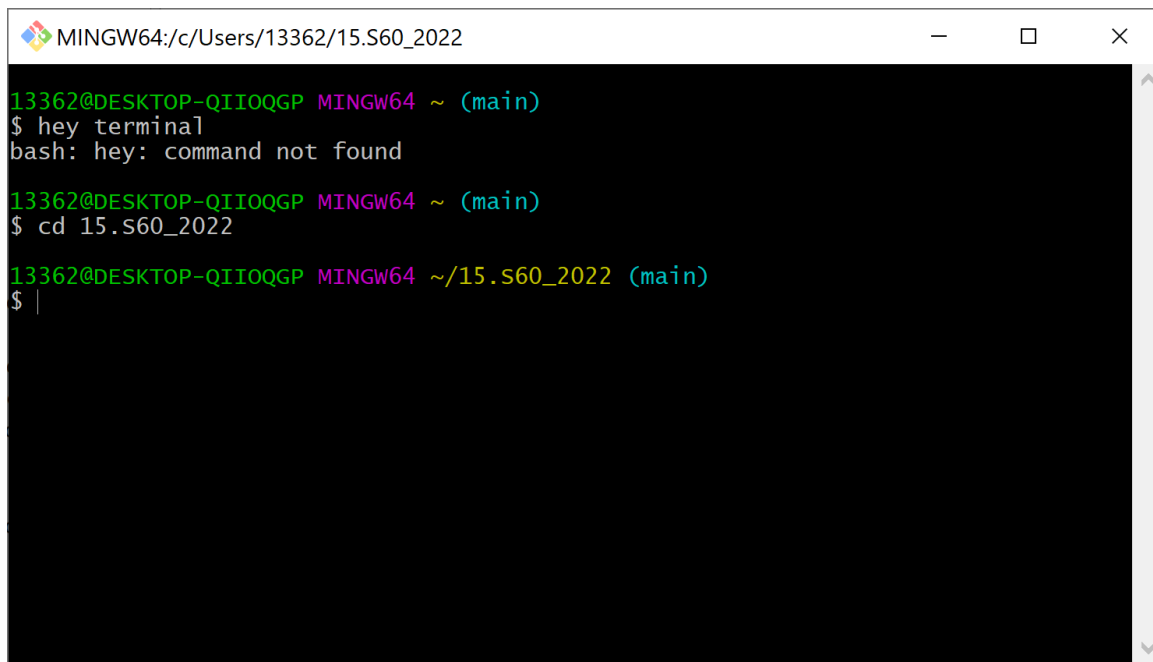
- Widely used by developers to store their projects
- Easily share code and data, privately and publicly

# Terminal

Adapted from slides by Galit Lukin, Jackie Baek

# What is the terminal?

The terminal/Unix shell is a text-based interface to interact with the computer.



```
MINGW64:/c/Users/13362/15.S60_2022
13362@DESKTOP-QIIQGP MINGW64 ~ (main)
$ hey terminal
bash: hey: command not found

13362@DESKTOP-QIIQGP MINGW64 ~ (main)
$ cd 15.S60_2022

13362@DESKTOP-QIIQGP MINGW64 ~/15.S60_2022 (main)
$ |
```

# Why use the terminal?



Repetitive tasks, like  
“delete all files in a  
directory ending in .csv”



Chain commands across  
programming languages  
sequentially



Athena



Engaging

Access to client servers  
and computing clusters  
with Secure Shell (SSH)

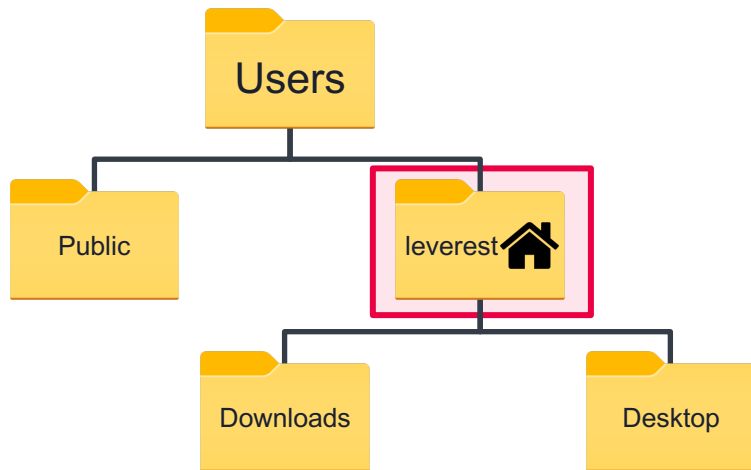
**We must learn a few basic commands to interact!**

# Learning Objectives - Terminal

At the end of the session, students will be able to...

- Navigate directories and manipulate files using terminal commands
- Recognize scenarios when it is necessary or more efficient use the terminal rather than a graphical interface

# Files and Directories



**Working  
directory**

## **Absolute path:**

`/Users/leverest/Desktop/15.S60_2024/data.csv`

## **Relative path:**

`Desktop/15.S60_2024/data.csv`

`data.csv`

`stuff.txt`

`script.py`

# Terminal Basics

- We are using a shell called **bash**. This program will interpret and process the commands you input into the terminal.
- A typical command looks like:

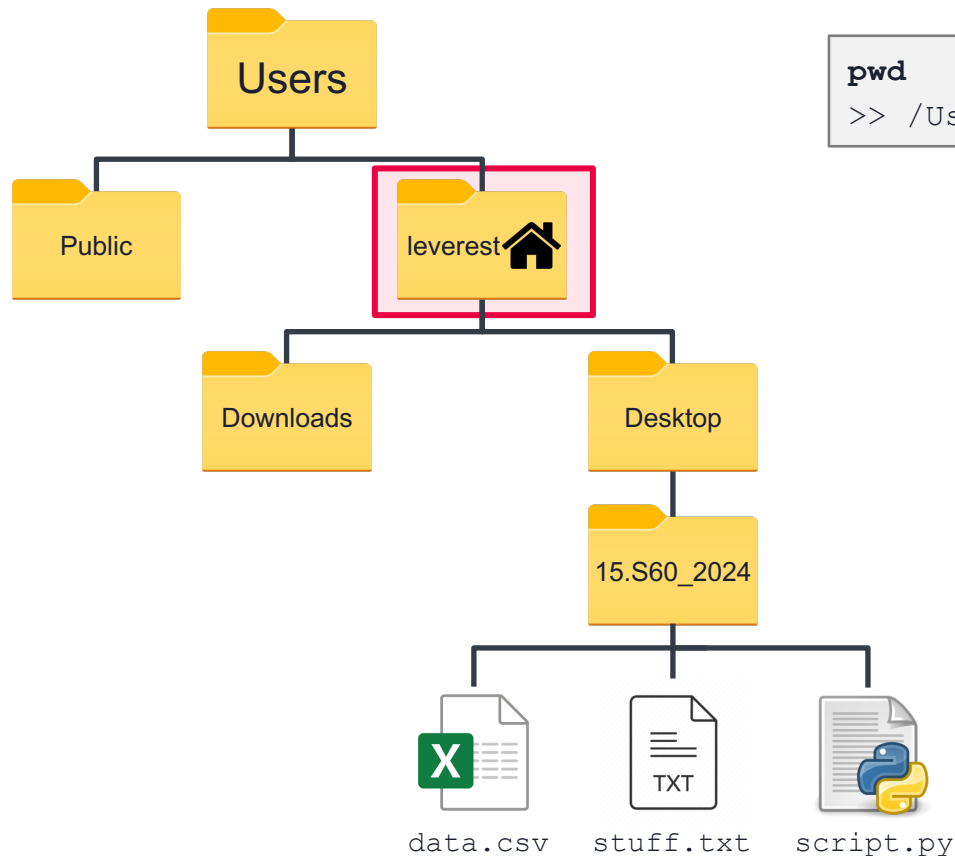
```
command <argument1> <argument2> ...
```

To open:

- Mac users open Terminal
- Windows users open Git Bash (installed in the pre-assignment)

# Navigating – Commands

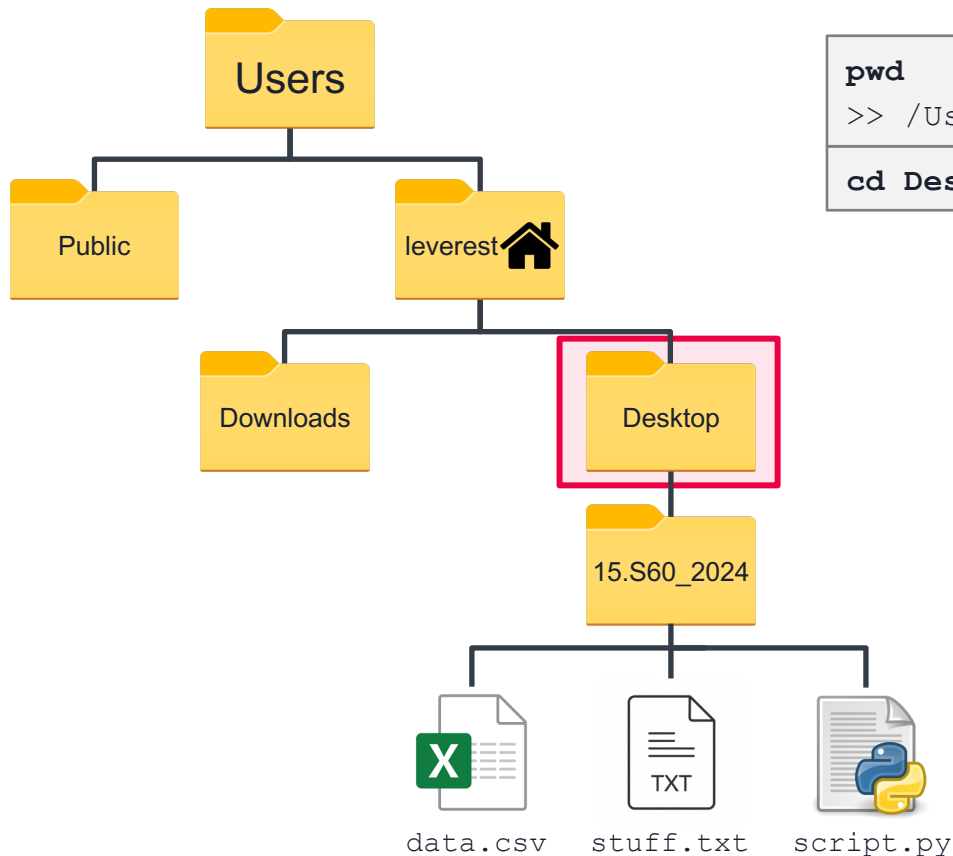
Working  
directory





# Navigating – Commands

Working  
directory



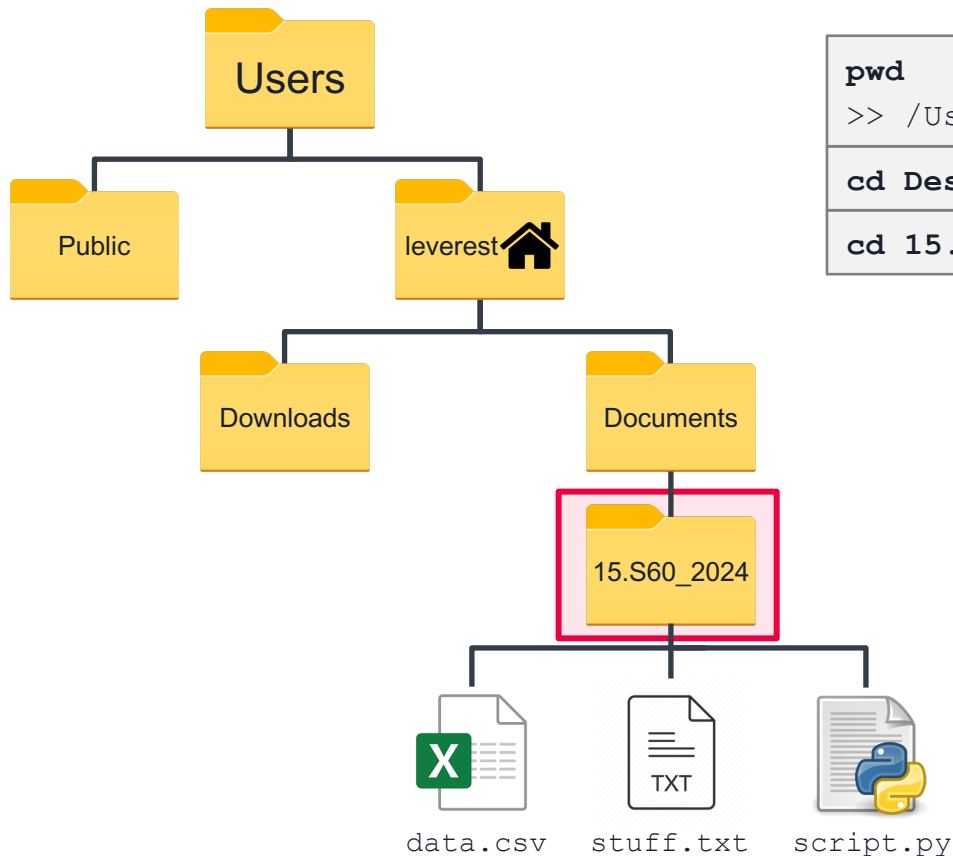
```
pwd
```

```
>> /Users/leverest
```

```
cd Desktop
```

# Navigating – Commands

Working  
directory



```
pwd
```

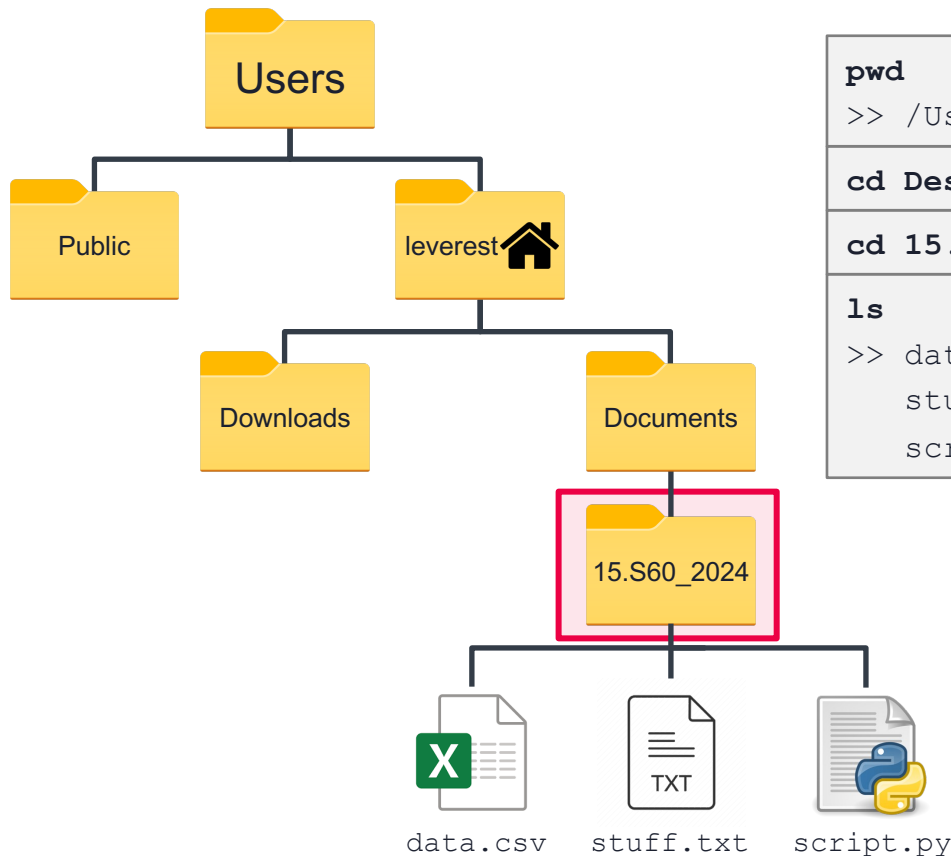
```
>> /Users/leverest
```

```
cd Desktop
```

```
cd 15.S60_2024
```

# Navigating – Commands

Working  
directory



**pwd**

```
>> /Users/leverest
```

**cd Desktop**

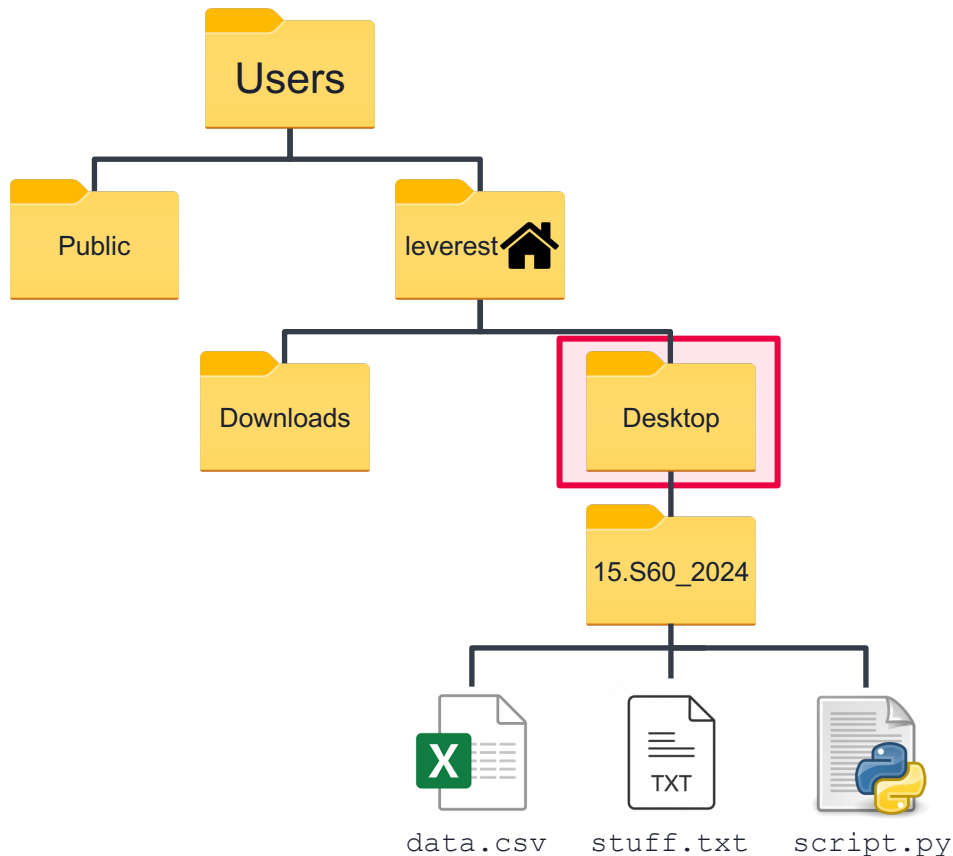
**cd 15.S60\_2024**

**ls**

```
>> data.csv  
stuff.txt  
script.py
```

# Navigating – Commands

Working  
directory

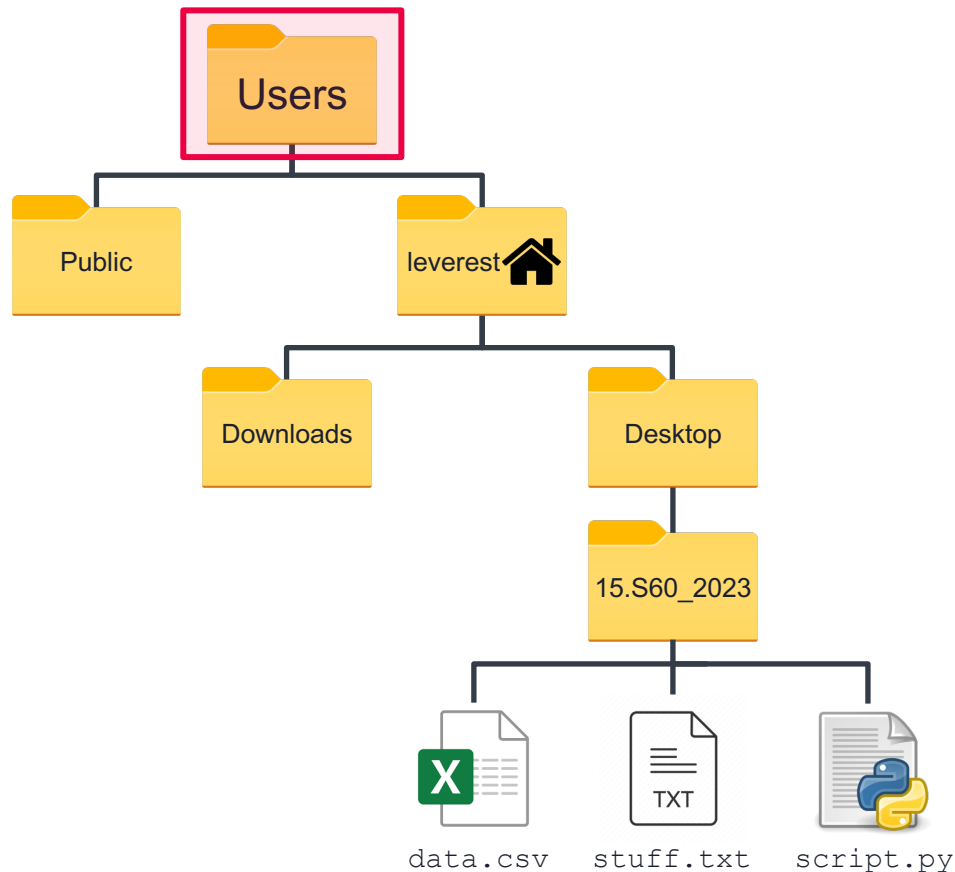


("Up" or "back" one level)

```
cd ..
```

# Navigating – Commands

Working  
directory



("Up" or "back" two levels)

```
cd ../../
```

("Up" or "back" one level)

```
cd ..
```

# Navigating – Commands

Working  
directory

(Back to home directory)

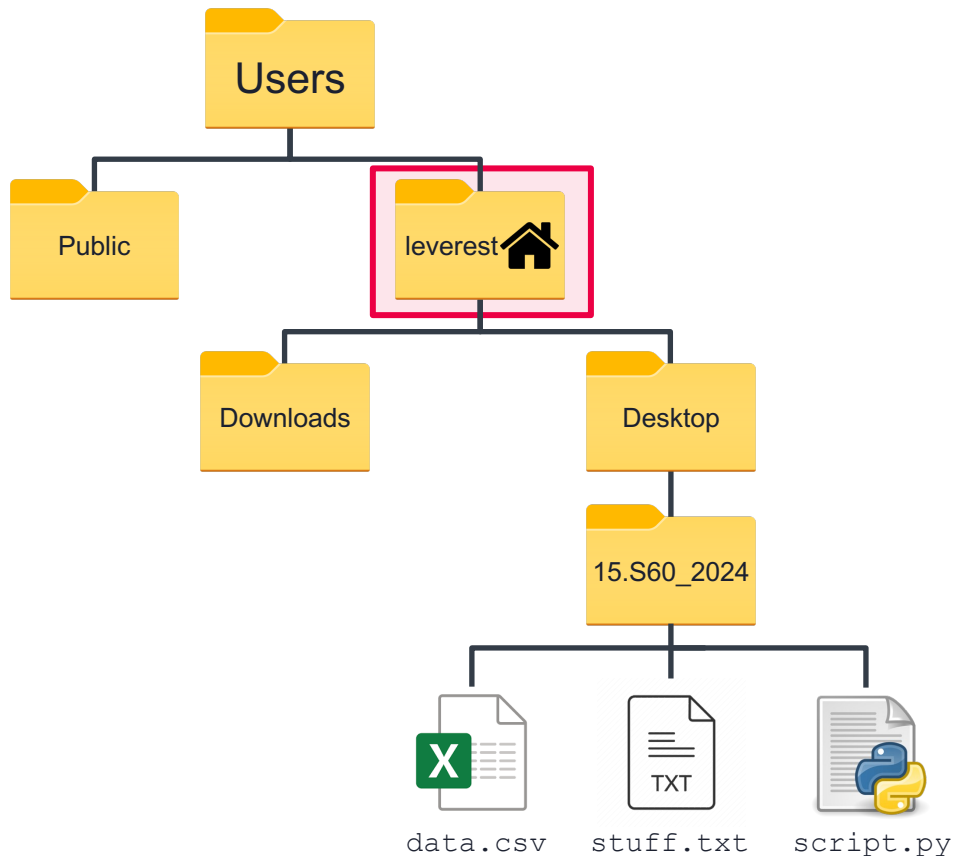
```
cd
```

(“Up” or “back” two levels)

```
cd ../../
```

(“Up” or “back” one level)

```
cd ..
```



# Navigating – Commands

Print working directory

```
pwd
```

---

List contents of working directory

```
ls
```

---

List contents of specified directory

```
ls <directory_name>
```

---

Change to a new directory

```
cd <directory_name>
```

---

Return to “home” directory

```
cd
```

---

Open a file (analogous to double clicking)

```
open <filename>
```

# Navigation – Commands 2

Reference current working directory

.

Reference parent of current working directory

..

Reference home directory

~

- Navigate history of previous commands using up and down arrow keys
- Use tab to autocomplete commands and paths



# File Manipulation – Commands

Create a new directory

```
mkdir <directory_name>
```

Create a new file

```
touch <filename>
```

Delete a file **(cannot be undone!)**

```
rm <filename>
```

Delete a directory **(cannot be undone!)**

```
rm -r <directory_name>
```

Edit file contents (Nano, vim are text editors)

```
nano <filename>  
vim <filename>
```

Print contents of a file

```
cat <filename>
```

Move or rename a file

```
mv <source_filename> <target_filename>
```

# Demo – Navigation

1. Open Terminal (MacOS) or Git Bash (Windows)

2. Navigate to the class directory from the pre-assignment

```
cd 15.S60_2024
```

3. Pull the class Github to get any updates

```
git pull
```

4. Navigate to the class directory from the pre-assignment

```
cd 7_terminal_and_git
```

5. Print off the list of files in that directory

```
ls
```

6. Navigate back to your home directory

```
cd      or      cd ~      or      cd ../..
```

# File Manipulation – Demo

We want to create a new directory called `mydirectory`, navigate to it, add a new file called `myfile.txt`, then add a line of text to the file.

1. Create and navigate to a new directory

```
mkdir mydirectory  
cd mydirectory
```

2. Add and open a new file for editing

```
touch myfile.txt  
nano myfile.txt
```

OR

```
vim myfile.txt
```

3. Add some text

Save (Ctrl+O), and  
exit Nano (Ctrl+X)

OR

Command interface (Esc), and  
save and exit (:wq)

# Redirecting Outputs



**Run a script**



**Send the output log to a file**  
(rather than the terminal)

---

```
python processStuff.py > outputfile.txt
```

# Redirecting Outputs



Run a script



Send the output log to a file  
(rather than the terminal)

Overwrite / create file

```
<command> > outputfile.txt
```

Append to file

```
<command> >> outputfile.txt
```

# Redirecting Outputs - Demo

I can create a Julia script in `mydirectory` called `myscript.jl`. I can then run it and direct the output to a file called `scriptoutput.txt`.

1. Create new text file

```
touch myscript.jl
```

2. Print the contents of the directory and redirect the output

```
julia myscript.jl > scriptoutput.txt
```

3. Check the output:

```
cat scriptoutput.txt
```

# Redirecting Outputs - Try it out

Create another text file in `mydirectory` called `newfile.txt`. Print the contents of `mydirectory` and direct the list to a file called `outputfile.txt`.

1. Create new text file

```
touch newfile.txt
```

2. Print the contents of the directory and redirect the output

```
ls > outputfile.txt
```

3. Check the output:

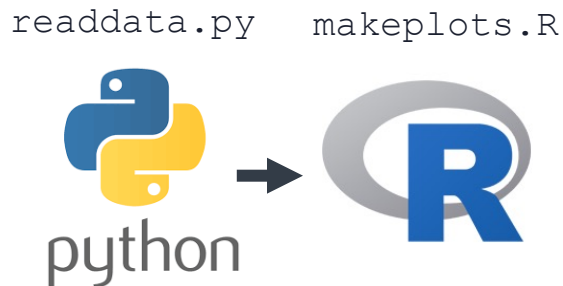
```
cat outputfile.txt
```

# More Terminal

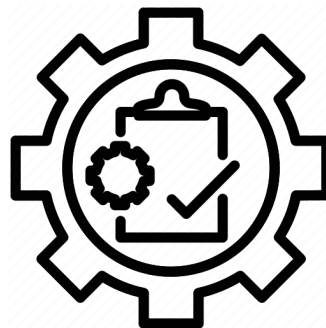
There are many more things you can do in Terminal.



Simple pattern matching  
to find and sort files



Use multiple programming  
languages to run  
sequential processes



Shell scripts to automate  
chains of commands  
you use often

If you're interested, check out the tutorial here: <https://swcarpentry.github.io/shell-novice/>



# Git

**Adapted from slides by Galit Lukin and Jackie Baek  
with activities from Turing Way and Software Carpentry**

# Learning Objectives – Git/Github

At the end of the session, students will be able to:

- Maintain a version control history with appropriate documentation
- Contribute to a project with shared code
- Share and contribute code to the broader community

# Git and Github

**Github** - Hosting platform for version control and collaboration



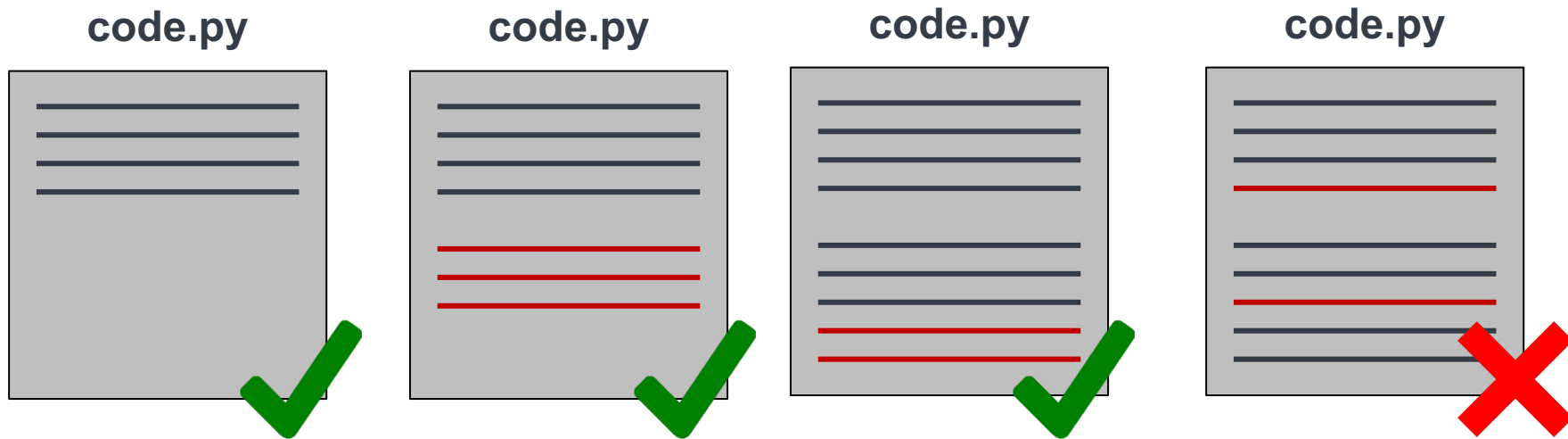
- Widely used by developers to store their projects
- Easily share code and data, privately and publicly

**Git** - The version control system itself



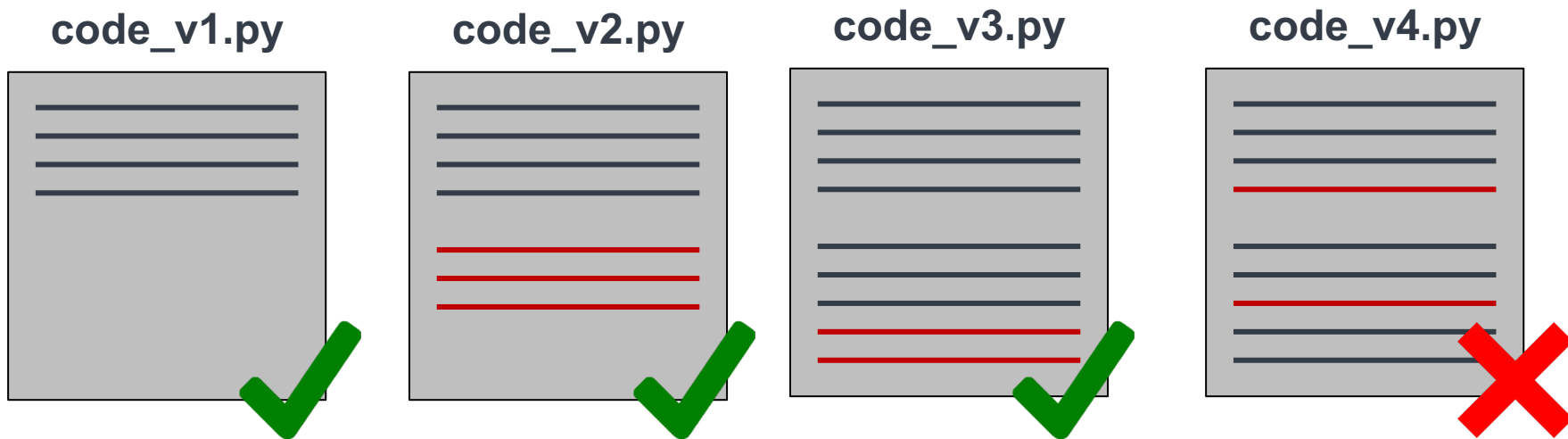
- Command line tool
- Used to make local code changes and communicate those changes with Github

# Why use version control (Git)?



If only I could go back to a version that was working...

# Why use version control (Git)?



What changed between versions? Which version to go back to? What if it's been months since I worked on this? What if my project has many files?

# High-Level Idea

## Benefits:

- Snapshot of entire repo, not one file
- Git stores commit messages
- Commits are lightweight
- Can go back to previous commits

**repository (aka repo)**



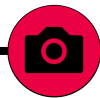
Initial  
commit



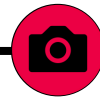
Add client  
data files



Fix initial  
solution for  
IP



Refactor data  
cleaning  
process



Change graph  
color scheme

# Create a repository - Commands

1. Create a new directory for your repo

```
mkdir <directory_name>
```

---

2. Navigate to the directory

```
cd <directory_name>
```

---

3. Initialize repository (you only need to do this once for each repo you create)

```
git init
```

# Initialize a Git Repo - Demo

Let's turn `mydirectory` into a git repo.

1. Navigate to a `mydirectory` if you aren't already there

```
cd mydirectory
```

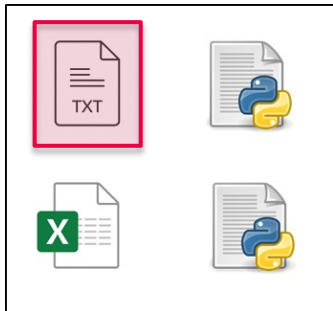
2. Initialize the git repo

```
git init
```



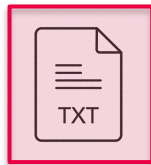
# Add and commit

**working directory**



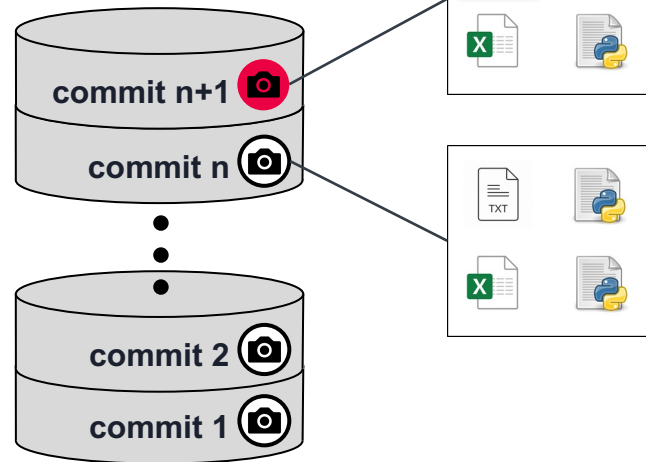
**Local repository, plus  
your local changes**

**staging area**



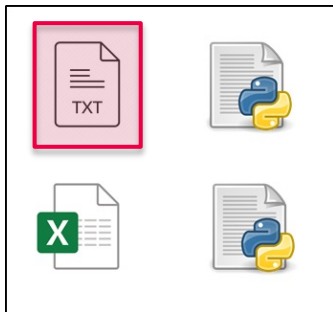
**Your local changes that  
you want to get  
“permanently” added to  
local repository**

**local repository**



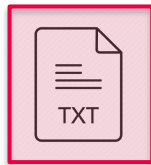
# Add and commit

working directory



Local repository, plus  
your local changes

staging area

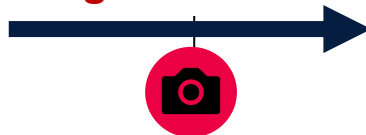


Your local changes that  
you want to get  
“permanently” added to  
local repository

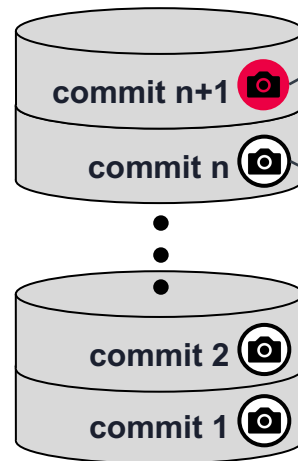
git add



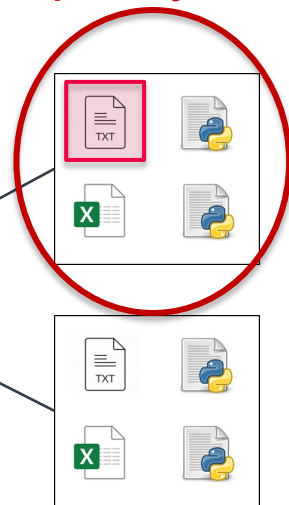
git commit



local repository



newly-updated local  
repository



# Add and commit - Commands

1. Make changes in working directory

---

2. Check the files in the staging area and any “untracked” changes (not yet added)

```
git status
```

---

3. As you change files:

**working directory** → **staging area**

```
git add <filename>
```

```
git add . (adds all files)
```

---

4. Once you have added a new functionality/completed an update:

**staging area** → **local repo**

```
git commit -m "comment"
```

Add a short, intuitive comment  
describing the change in functionality  
→ For reference later!

# Question

In your opinion, which would be the most helpful commit message three months from now? (**somewhat subjective!**)

A.) `"Fix bug"`

B.) `"Add functions: cleanData, processData, runPredictions"`

C.) `"Add function to plot model results"`

D.) `"Write latest optimization model"`

# Add and commit - Demo

Let's commit something to `mydirectory`! Run `git status` between each step to see the staging area.

1. Check the current status

```
git status
```

2. Add the file to the staging area

```
git add newfile.txt  
git status
```

3. Commit the changes to the repository

```
git commit -m "Add empty file"  
git status
```

# Add and commit - Try it out

We want to create a new repository called `myrepo` (outside of `mydirectory`), create a text file in `myrepo` called `myfile.txt`, and add and commit it to the repository.

1. Leave `mydirectory`. Create and navigate to a new directory and initialize Git.

```
cd ..  
mkdir myrepo  
cd myrepo  
git init
```

2. Create an empty text file in `myrepo`

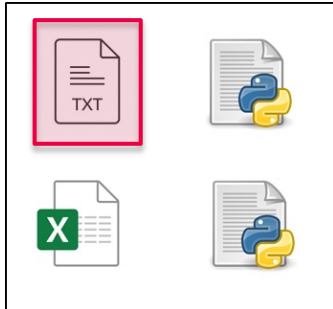
```
touch myfile.txt
```

3. Add and commit the text file to your new repository

```
git add myfile.txt  
git commit -m "Add first text file"
```

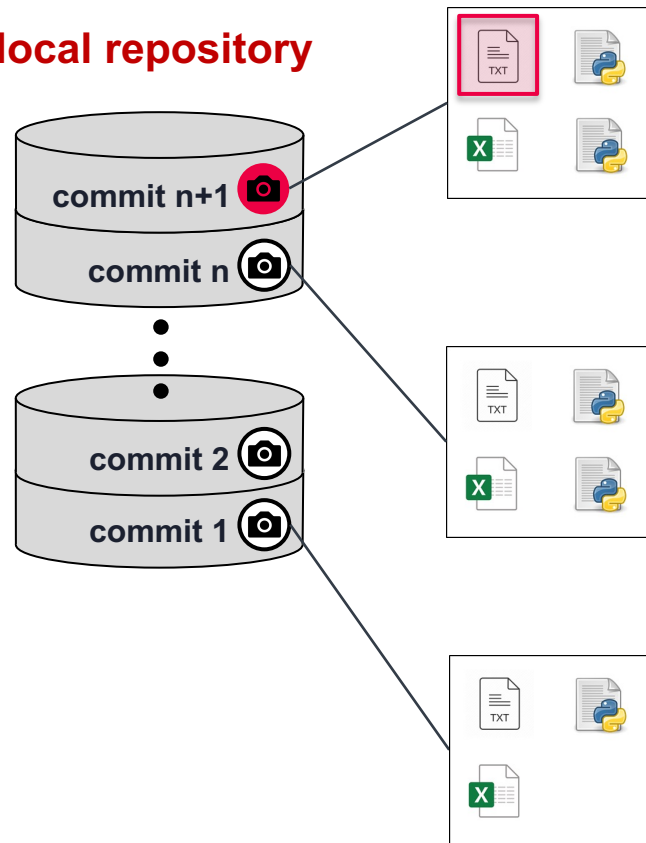
# Reverting Changes

working directory



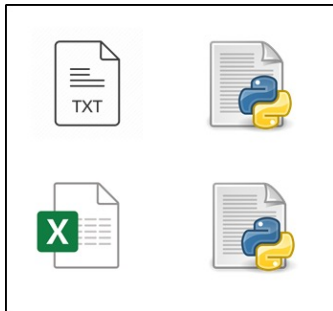
staging area

local repository



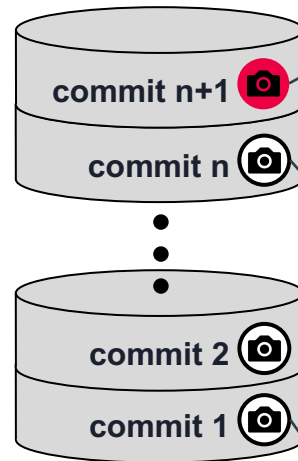
# Reverting Changes

working directory

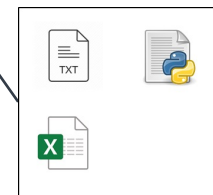
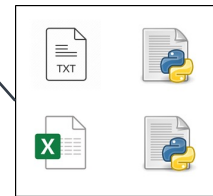
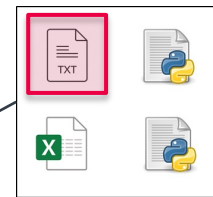


staging area

local repository



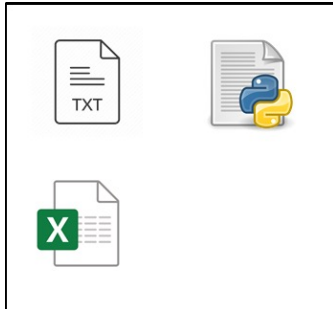
git checkout





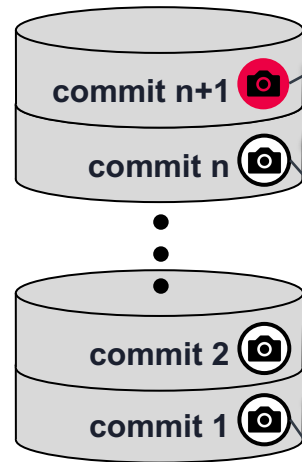
# Reverting Changes

working directory

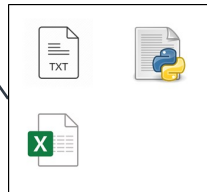
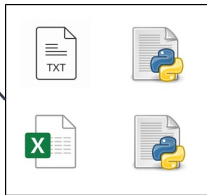
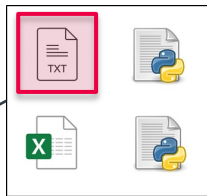


staging area

local repository



git checkout



# Reverting Changes - Commands

See log of commits, including  
SHA ID for each version

```
git log
```

---

Return to a previous version of  
a file from the commit history

```
git checkout <version_SHA>
```

```
git checkout <version_SHA> <filename>
```

---

See differences between two  
commits or differences in a  
specific file from two commits

```
git diff <version1_SHA> <version2_SHA>
```

```
git diff <version1_SHA:filename>  
        <version2_SHA:filename>
```

# Reverting Changes - Try it out

Add some text to `myfile.txt` and commit. Check the log of commits, look at the differences between our two commits so far, and then check out the first commit.

1. Edit and add some text to `myfile.txt` and commit changes.

```
nano myfile.txt  
git add myfile.txt
```

```
git commit -m "Add some text"
```

2. View the log to see our commit history

```
git log
```

3. Find the differences between the two commits (The first 7 digits of the SHA id should be sufficient)

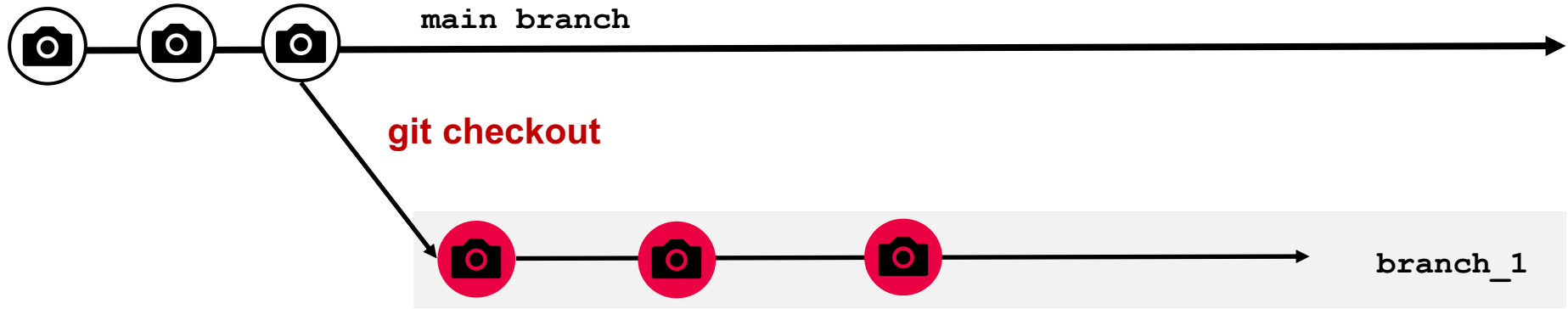
```
git diff 1093106 976686d
```

4. Check out the previous commit

```
git checkout 976686d
```

(Note this will put you in “detached HEAD” state, which you can get out of with: `git checkout main`)

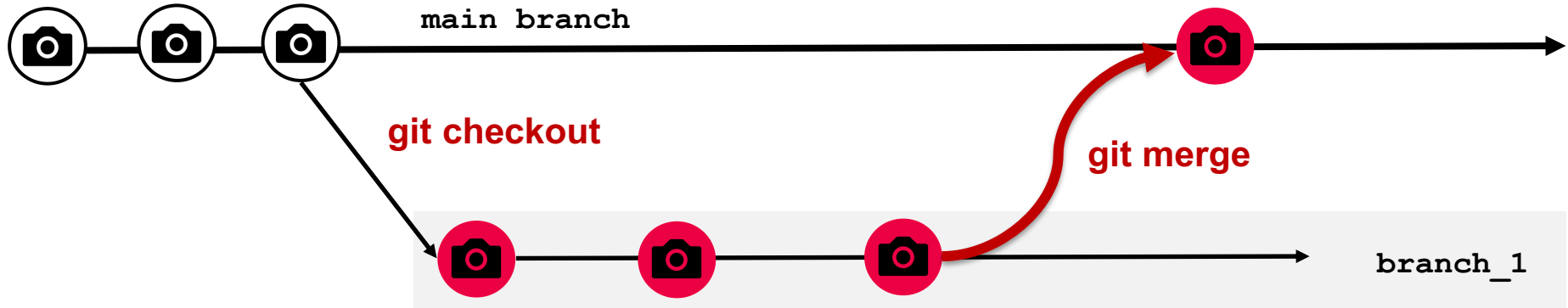
# Branching



Suppose you run a process for a client every morning.  
You want to make a change, but it will take a few days.  
How can you work on the change AND run the stable version?

# Branching

Branch when the intention is to eventually “merge” the branch back in with main



Suppose you run a process for a client every morning. You want to make a change, but it will take a few days. How can you work on the change AND run the stable version?

# Branching and Merging - Commands

See all branches

```
git branch
```

---

Create and navigate to a new branch

```
git checkout -b new_branch_name
```

---

Switch between branches

```
git checkout branch_name
```

---

Merge two branches

```
git checkout branch_receiving_changes  
git merge branch_giving_changes
```

---

Delete a branch

```
git branch -d branch_name
```

# Branching - Demo

Create a new branch called `my_branch`, add a file called `mynewfile.txt` to `my_branch` and commit, and look at the list of branches and log of commits.

1. Create and check out a new branch (one step)

```
git checkout -b my_branch
```

2. Add a new file to the new branch

```
touch mynewfile.txt
```

3. Add the new file to staging area and commit

```
git add mynewfile.txt
```

```
git commit -m "Add a second text file"
```

4. Display the branches, then look at the log of commits

```
git branch
```

```
git log
```

**Best practice: check the status  
between adding and committing**

```
git status
```

# Merging - Demo

Navigate to the `main` branch, merge `my_branch` into `main`, and delete `my_branch`.

1. Navigate to the main branch

```
git checkout main
```

2. Merge `my_branch` into the main branch

```
git merge my_branch
```

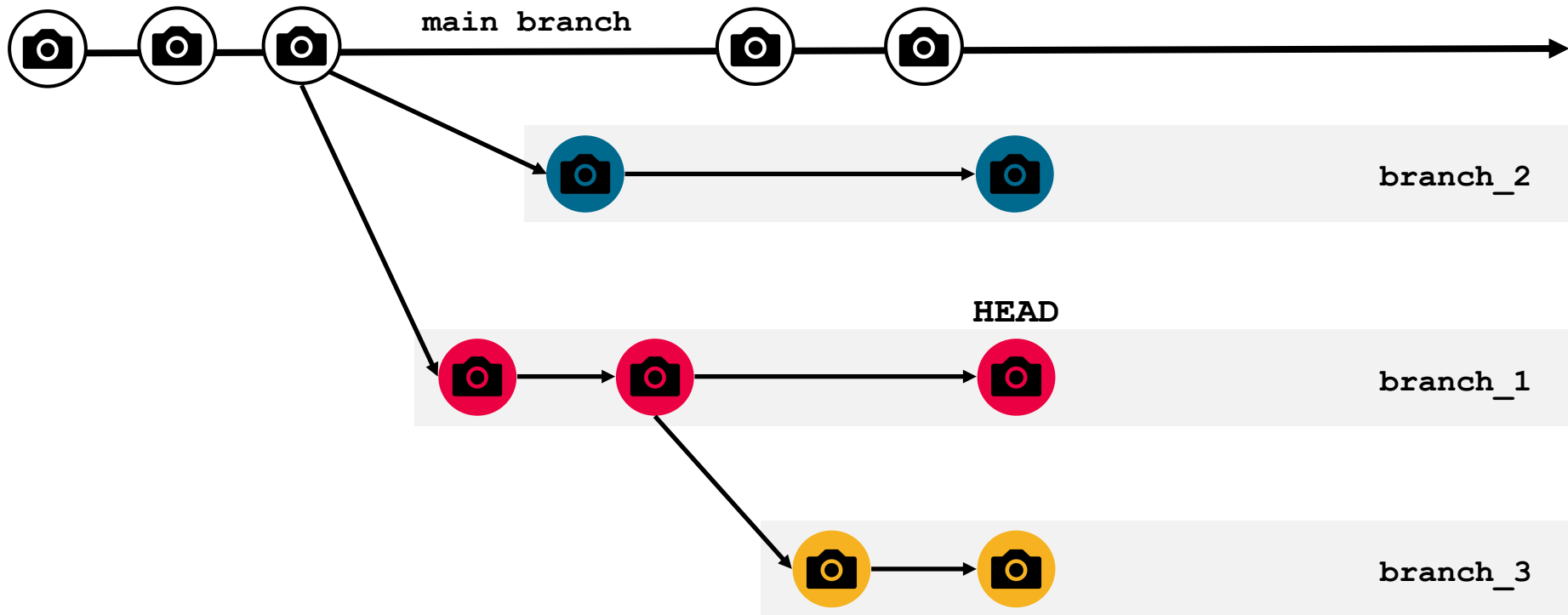
3. Delete `my_branch`

```
git branch -D my_branch
```



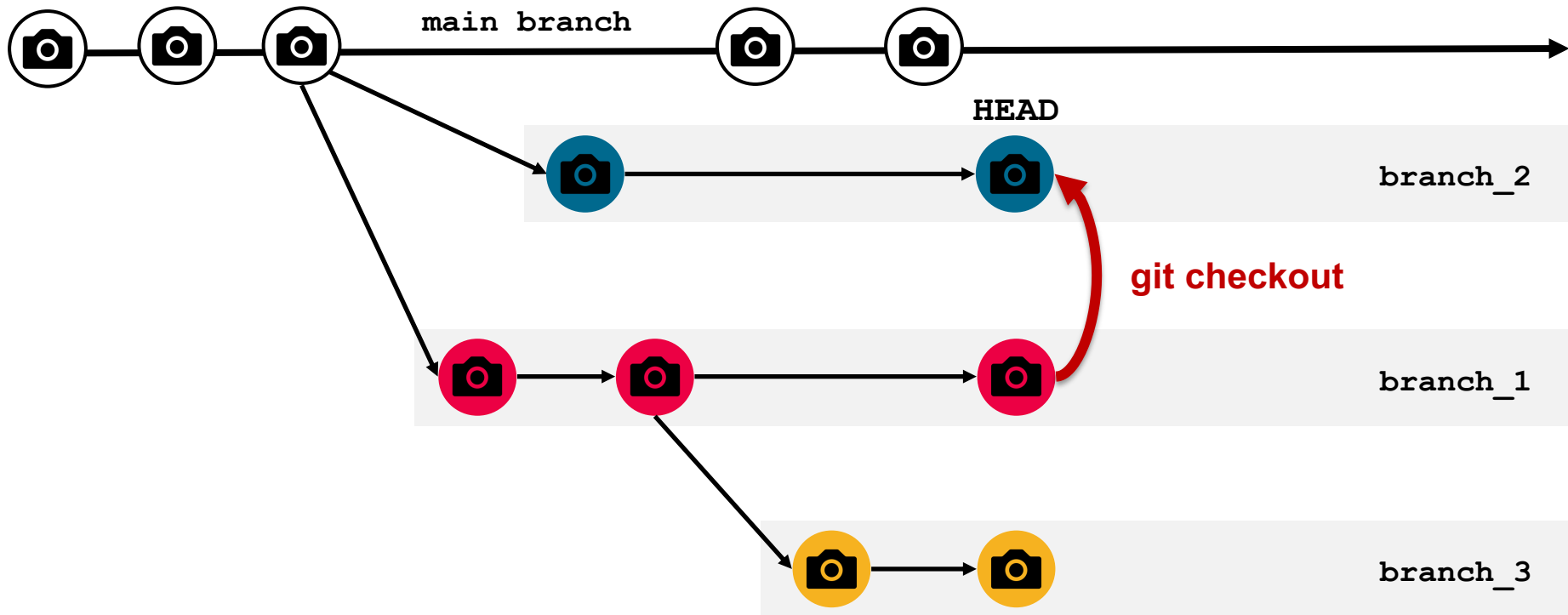
# Multiple Branches

HEAD – the most recent commit on the current branch



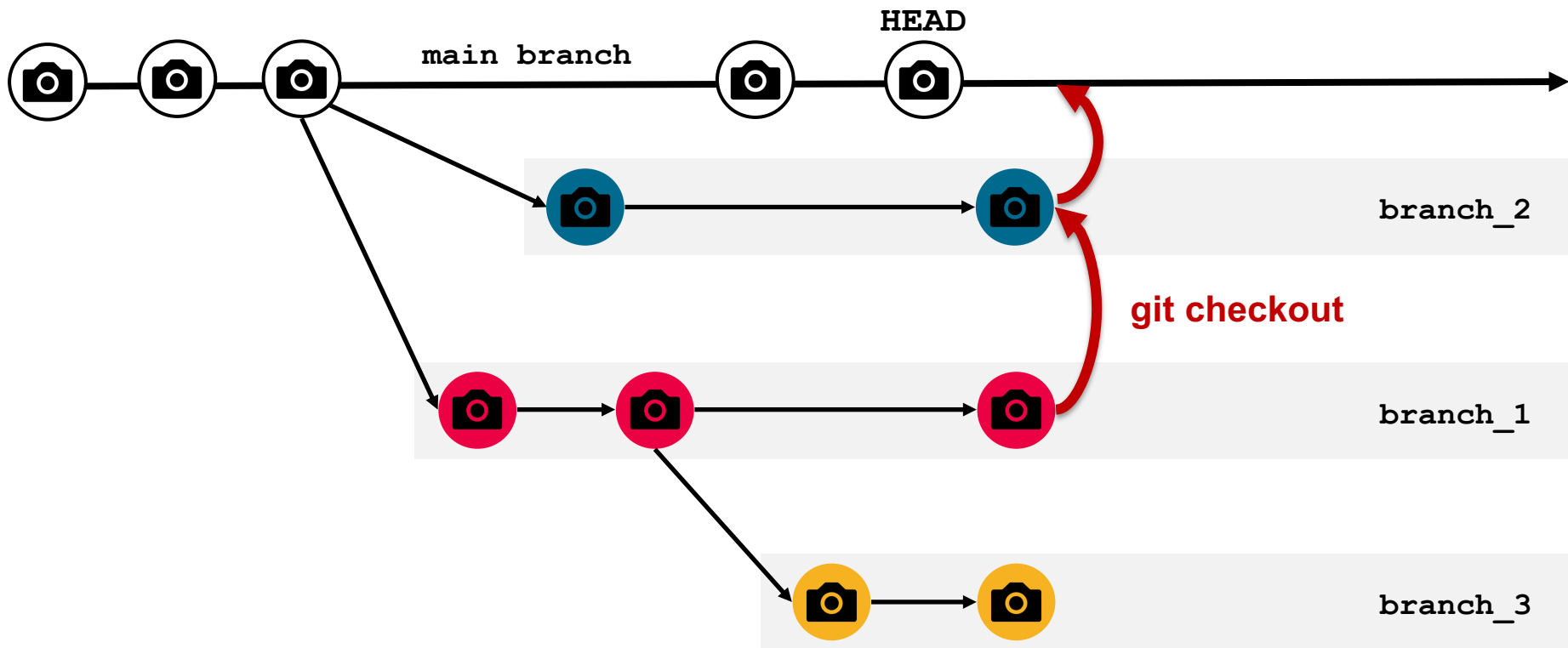
# Multiple Branches

HEAD – the most recent commit on the current branch



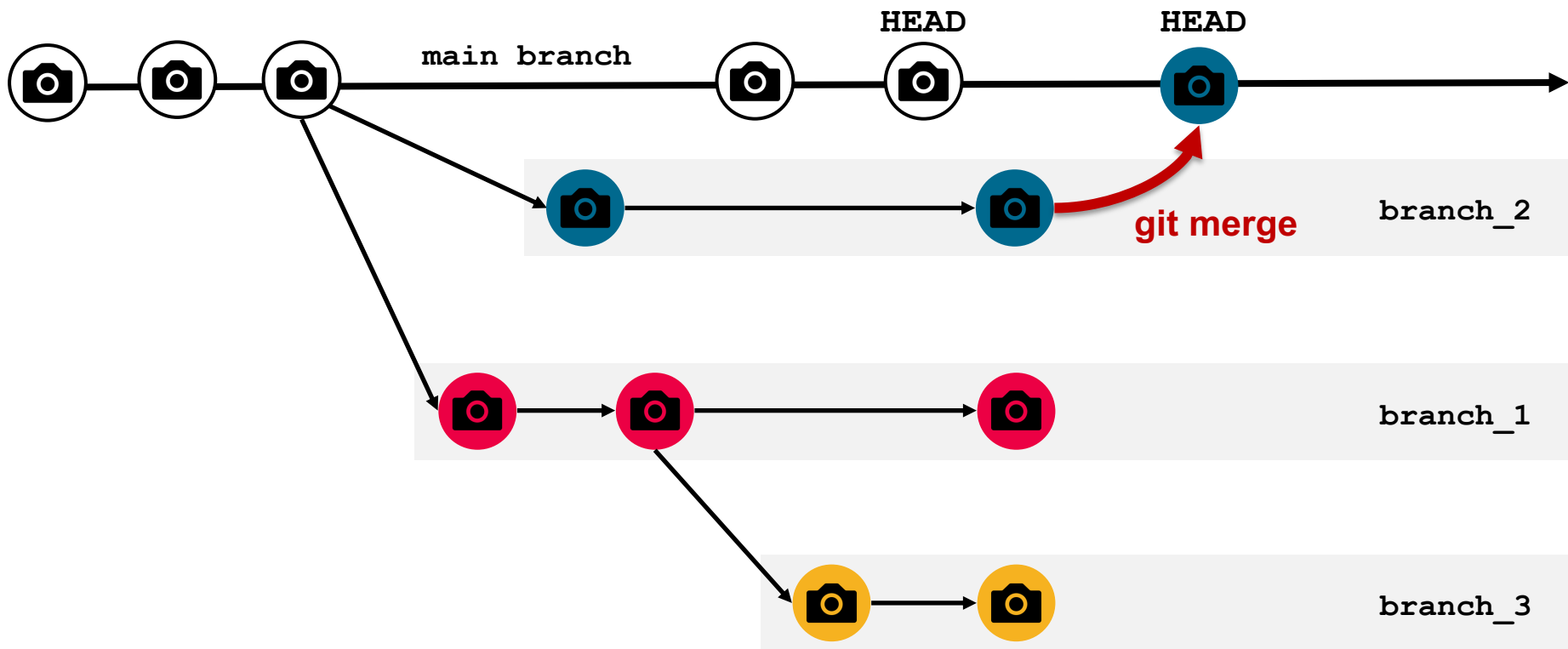
# Multiple Branches

HEAD – the most recent commit on the current branch



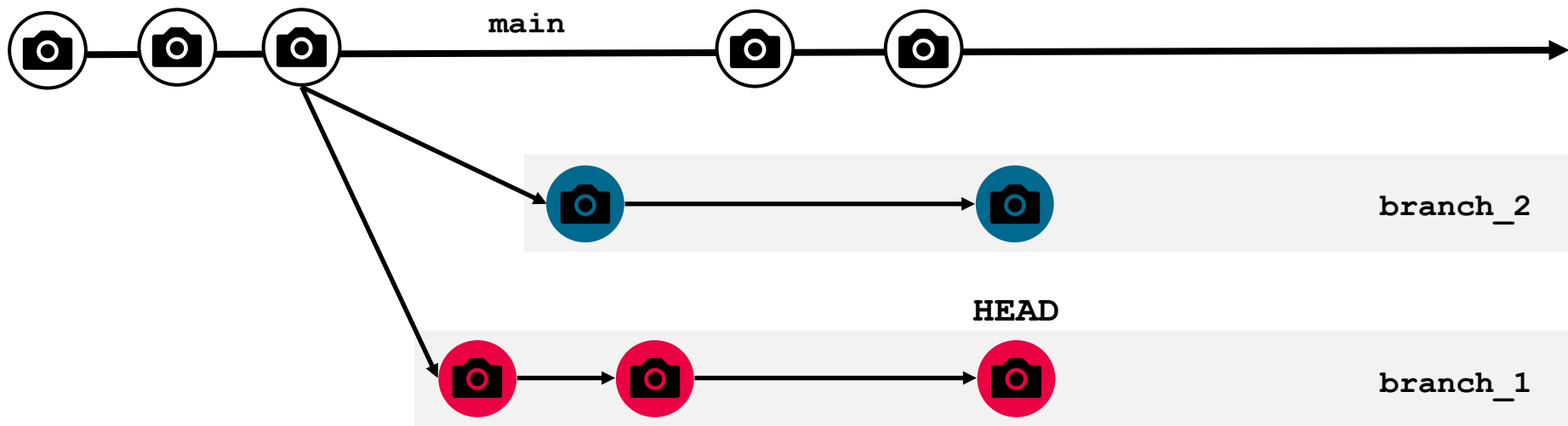
# Multiple Branches

HEAD – the most recent commit on the current branch



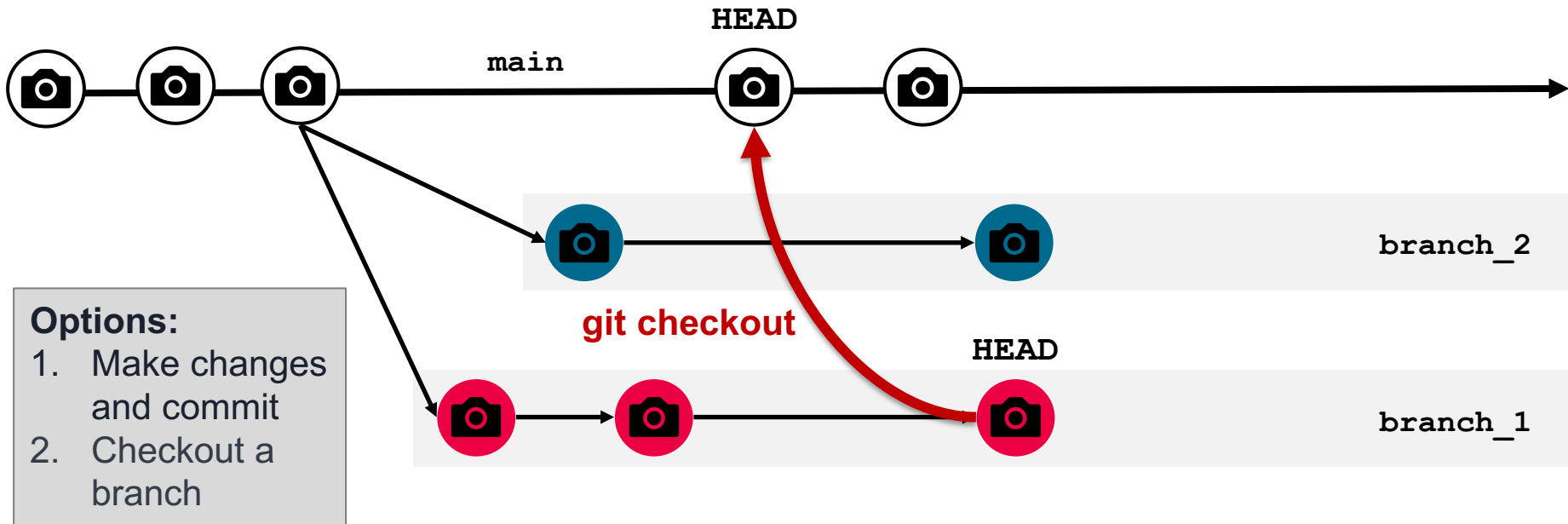
## Aside: 'detached HEAD' state

HEAD usually points at a branch directly. When you checkout a commit, HEAD points instead points directly at that commit. This is a 'detached HEAD' state. **That's okay!**



# Aside: 'detached HEAD' state

HEAD usually points at a branch directly. When you checkout a commit, HEAD points instead points directly at that commit. This is a 'detached HEAD' state. **That's okay!**



# Branching – Try it out

Checkout the `main` branch. Then checkout an earlier commit to enter a detached HEAD state. Create a new branch called `new_feature`. Check out the `main` branch.

1. Checkout main, view log, and checkout an earlier commit

```
git checkout main
git log
git checkout 976686d
```

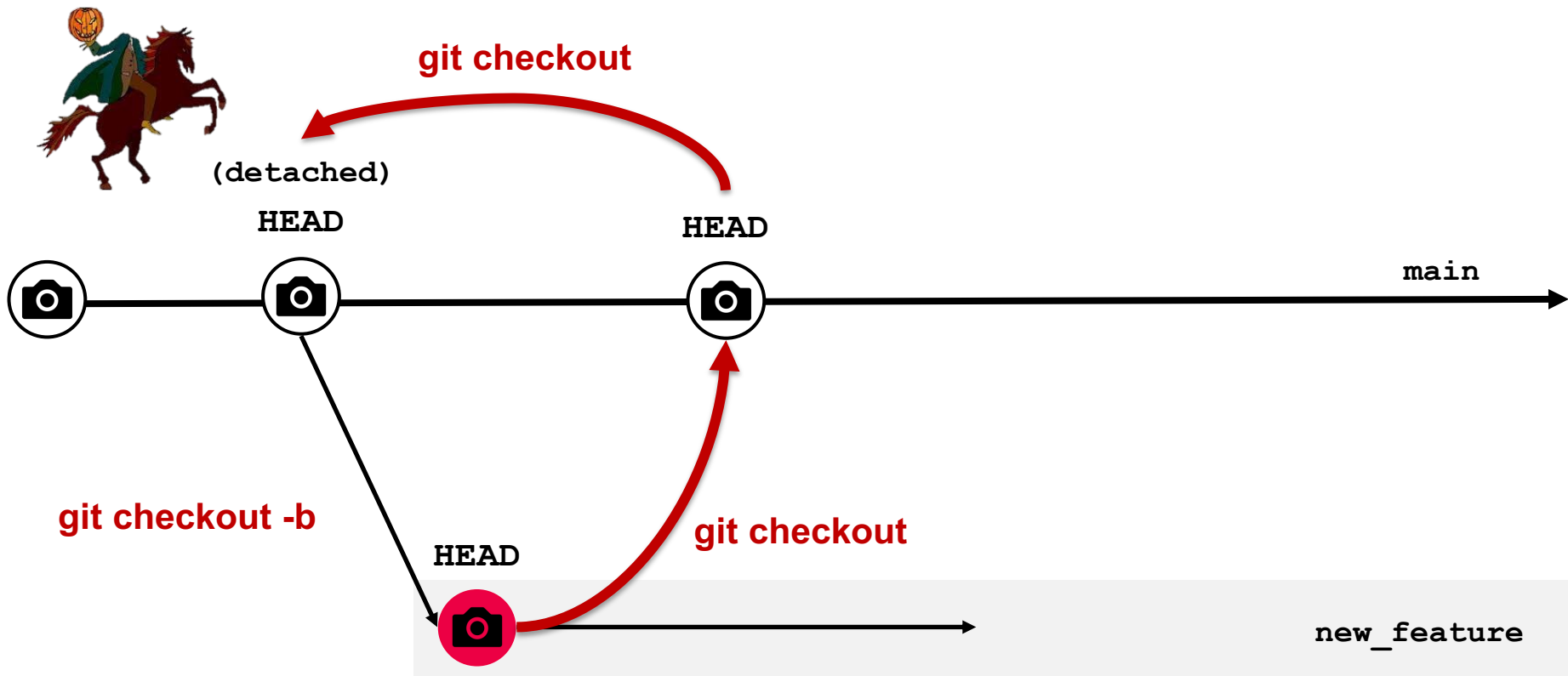
2. Create a new branch from this commit

```
git checkout -b new_feature
```

3. Display the branches, checkout main, then look at the log of commits

```
git branch
git checkout main
git log
```

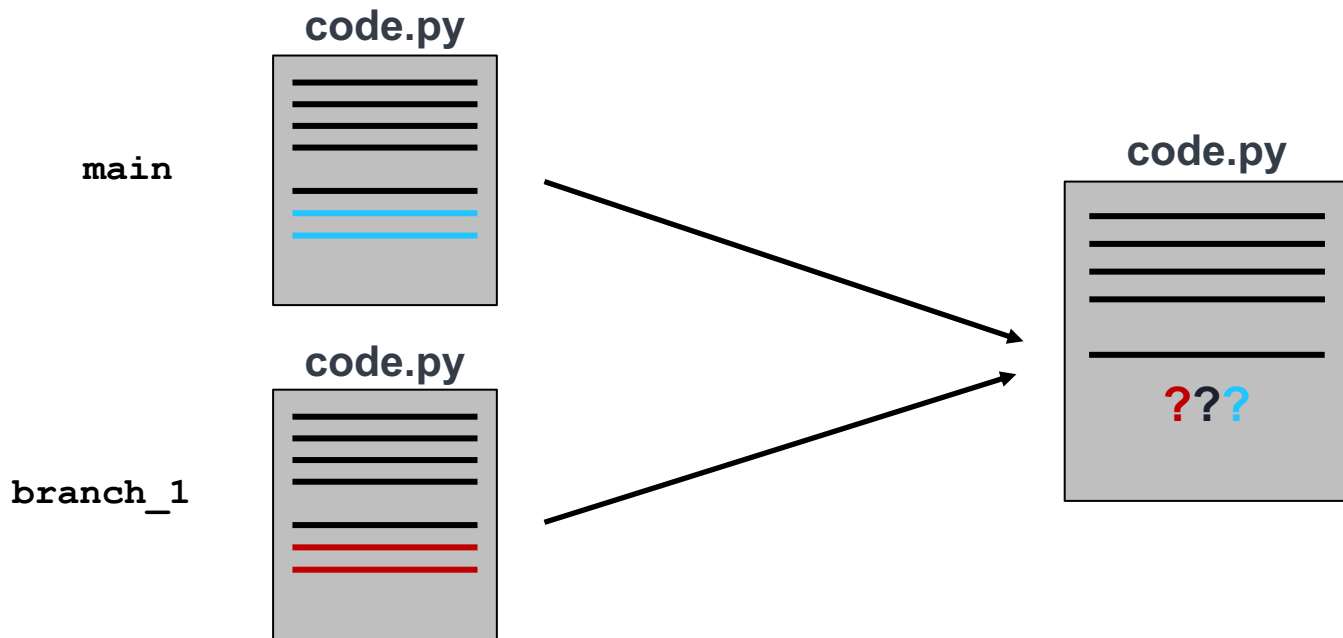
# 'detached HEAD' state





# Merge Conflicts

When merging changes, we may sometimes find that our branches conflict.



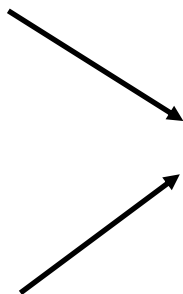
# Merge Conflicts

HEAD

The first line of the file is the same  
The second line is different

Commit dabb4c8c

The first line of the file is the same  
See? It's different



The first line of the file is the same  
<<<<<<< HEAD  
The second line is different  
=====  
See? It's different  
>>>>>> dabb4c8c450e8475aee9b14b4383acc99f42af1d

When this happens, Git shows you exactly what the conflicts are and where they are located. It is up to you to reconcile the conflicts and commit them.

# Merge Conflict – Demo (code along!)

Add text to `myfile.txt` in `main` and commit. Navigate to the `new_feature` branch, add different text to `myfile.txt`, and attempt to merge `new_feature` into `main`.

1. Add text to `myfile.txt` and commit to `main`

```
git checkout main
nano myfile.txt
git add myfile.txt
git commit -m "Add some text"
```

2. Add different text to `myfile.txt` in `new_feature` and commit

```
git checkout new_feature
nano myfile.txt
git add myfile.txt
git commit -m "Add different text"
```

# Merge Conflict – Demo (code along!)

Add text to `myfile.txt` in `main` and commit. Navigate to the `new_feature` branch, add different text to `myfile.txt`, and attempt to merge `new_feature` into `main`.

3. Navigate to the `main` branch and merge `new_feature` branch

```
git checkout main  
git merge new_feature
```

4. Resolve merge conflict

```
nano myfile.txt
```

5. Commit changes to `main`

```
git add myfile.txt  
git commit -m "Resolve myfile conflict"
```

# Question

Suppose you create a new branch and add a file, `branchfile.txt`, and commit. You then add a file, `mainfile.txt`, to the `main` branch and commit. When you try to merge, your branch to `main`, what do you expect to happen?

- A.) Merge successful! Both files exist in `main`
- B.) Merge successful! But `main` now holds only `branchfile.txt` but not `mainfile.txt`
- C.) Merge conflict, you must resolve

# Question

You add a file, `community.txt`, to the `main` branch and commit. You then create a new branch and add a file, `branchfile.txt`, and commit. You then delete `community.txt` from `main` and commit. When you try to merge your branch to `main`, what do you expect to happen?

- A.) Merge successful! And `branchfile.txt` exists in `main`
- B.) Merge successful! But `branchfile.txt` does not exist in `main`
- C.) Merge conflict, you must resolve

# Git Wrap-Up

- Commit often, write descriptive messages, and keep each commit to one “atomic” task
  - Almost anything can be undone, as long as it’s committed
- Take advantage of branching when you have new lines of development and don’t want to disrupt the functionality of the `main` branch
- Google is your friend (e.g. “How to undo commit in Git”).)

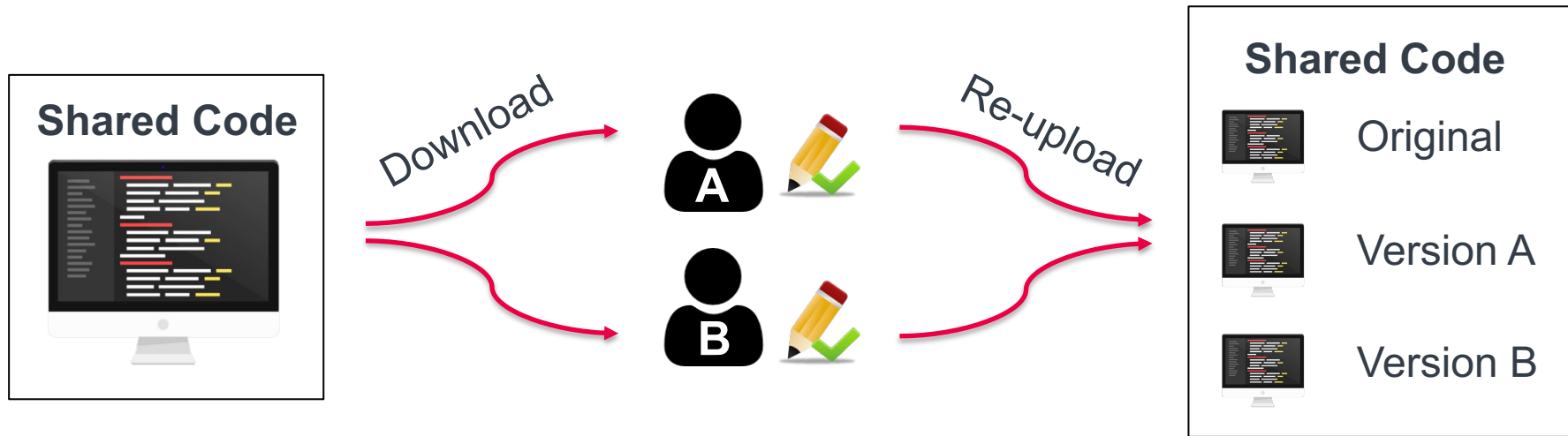
# **GitHub**

**Adapted from slides by Galit Lukin and Jackie Baek  
with activities from Turing Way and Software Carpentry**



# Why use Github?

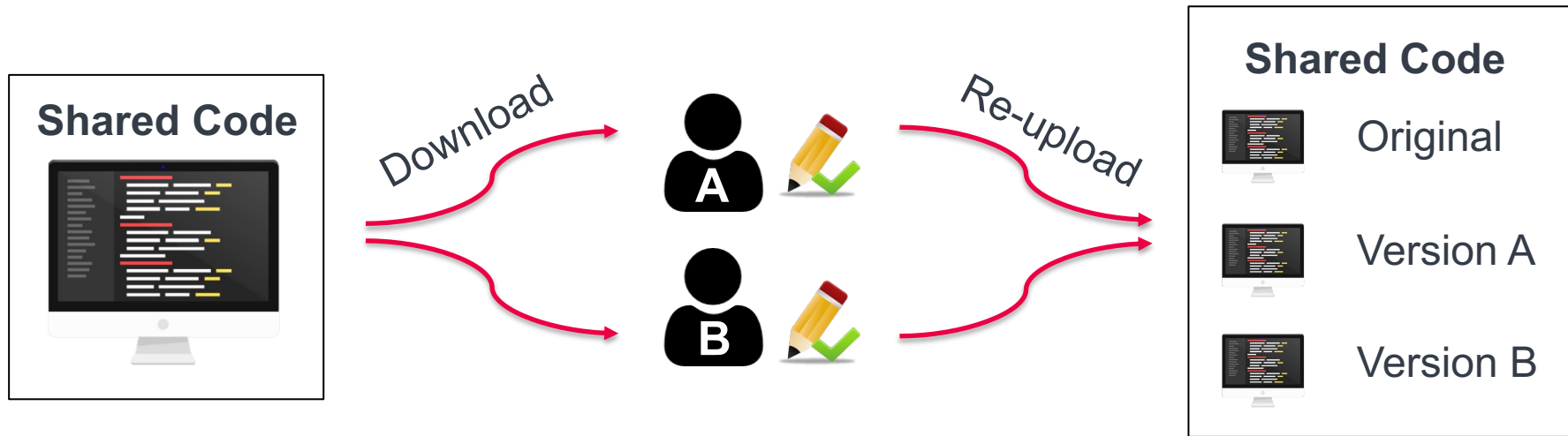
You're working on a research project with another graduate student and both have access to the code in a shared space, e.g. Dropbox.



**What are some potential problems that could arise using this process?**

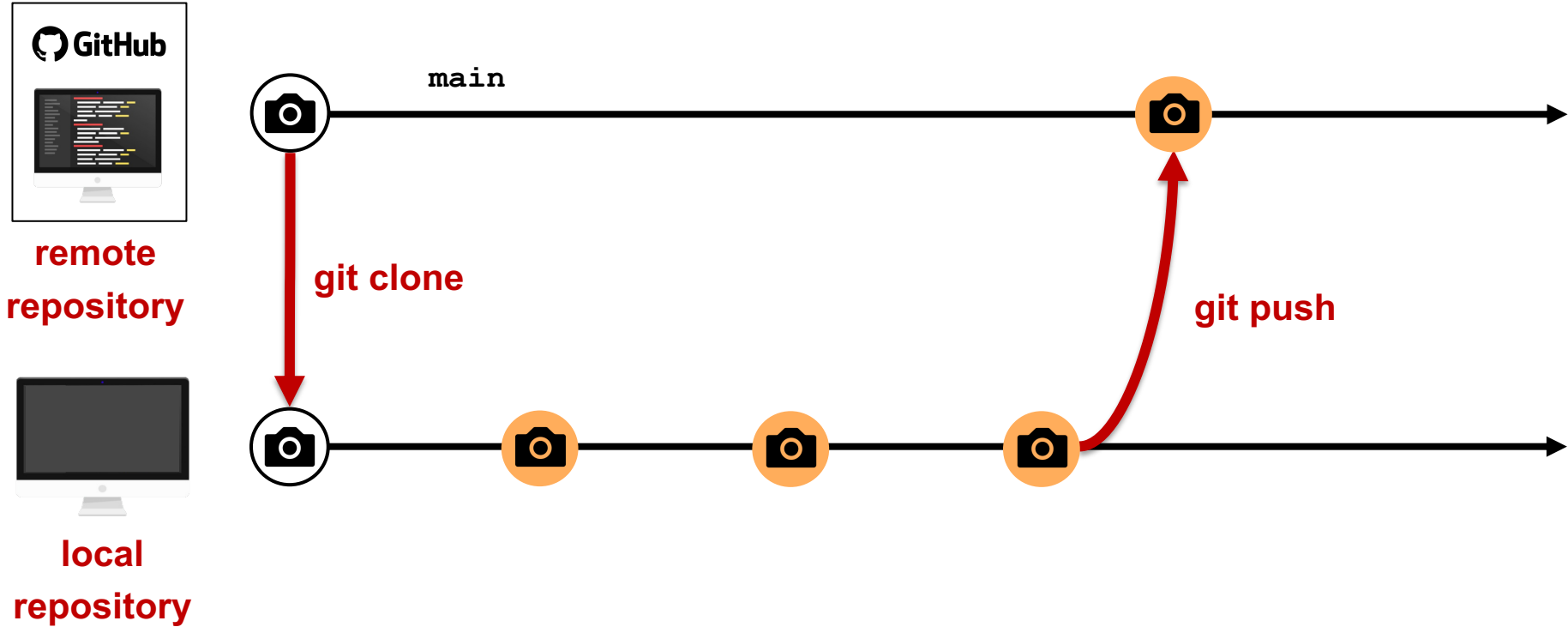
# Why use Github?

You're working on a research project with another graduate student and both have access to the code in a shared space, e.g. Dropbox.



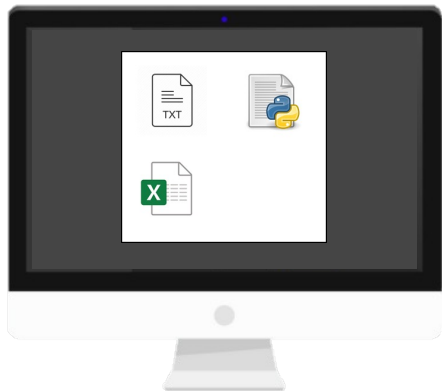
**Potential issues:** Documenting all changes, staying up to date with changes, overwriting others' work, conflicts between versions, etc.

# High-Level Idea



# Local and Remote Repositories

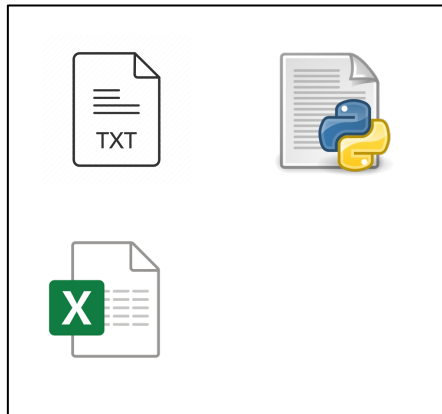
local repository



`git remote add`

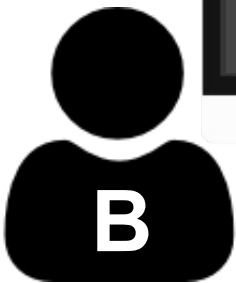
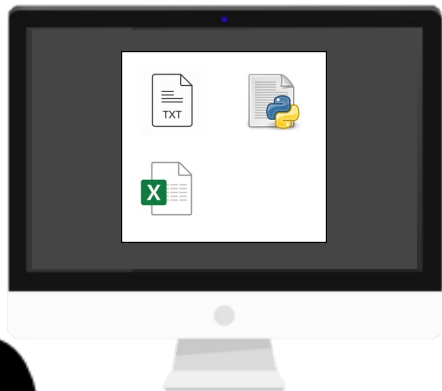


remote repository



# Local and Remote Repositories

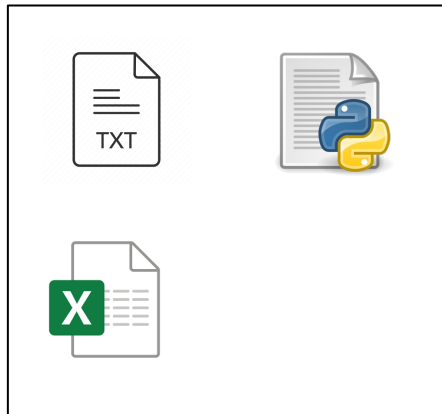
**Your collaborator's  
local repository**



**git clone**

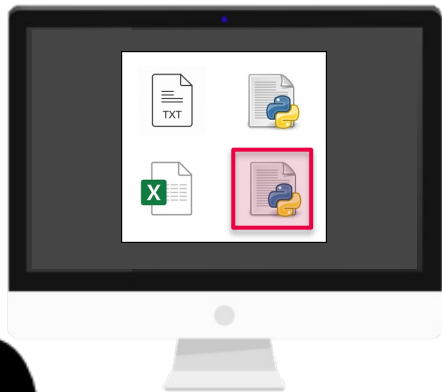


**remote repository**



# Local and Remote Repositories

**Your collaborator's  
local repository**

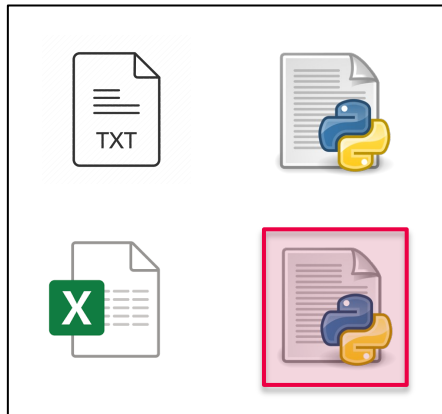


**git clone**

**git push**

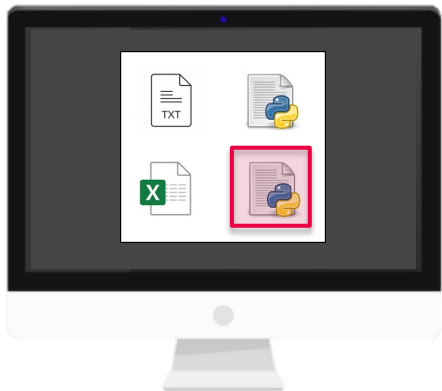


**remote repository**



# Local and Remote Repositories

**YOUR**  
local repository

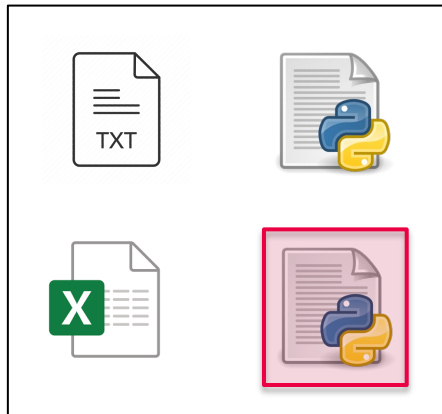


**git pull**

**git push**



**remote repository**



# Local and Remote - Commands

Add a remote connection to a Github repo for an existing local repo

```
git remote add origin <Github SSH>
```

Clone existing Github repo


```
git clone <Github URL>
```

Pull from Github repo to local repo

```
git pull
```

Push from local repo to Github repo

```
git push origin <branch_name>
```



Local machine's name for the remote repo  
(origin is usual convention)



# Cloning a Github repo - Try it out

Clone the class repo into the location of your choice.

1. Navigate to wherever you cloned the repo (or already cloned the repo)

```
cd ~/courses/
```

2. Clone the Github repository to your machine or pull changes if you already cloned it

```
git clone  
https://github.com/angkoulouras/15.S60_2024.git
```

- or -

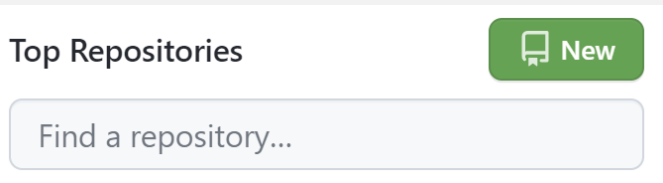
```
cd 15.S50_2024  
git pull
```

3. Find it on your computer and check it out! You should see the module for this session (or change your directory and type `ls` )

# Creating a Github repo through Github

Create your own repository in Github called `myrepo` (we'll walk through this together).

1. Go to [github.com](https://github.com) and login
2. Click “New” in the top left by your list of repositories and fill in details



3. Clone the Github repository to your machine **[skip this step for now]**

```
git clone https://github.com/<username>/<reponame>.git
```

# Creating a Github repo for existing local repo

Create a remote connection for `myrepo`

1. Navigate to the existing repo (from the Git part of class)

```
cd mydirectory
```

2. Add a remote connection

```
git remote add origin <Github HTTPS>
```

3. Push to upstream to Github

```
git push -u origin main
```

# Pushing to Github - Demo

Create a new local branch called `new_branch`, add some text to `mynewfile.txt` on `new_branch` and commit, merge the changes into `main`, and push to Github.

1. Create a new file and add a line of text

```
touch localfile.txt  
nano localfile.txt
```

2. Commit the changes locally

```
git add localfile.txt  
git commit -m "Add file to local repo"
```

3. Push the changes to Github

```
git push origin main
```

# Merge conflicts with Github

Your collaborators may have pushed new changes to Github while you've been working on your own changes locally.

## Best practices:

- Pull from Github often, even when you aren't making changes to the remote repo
- Pull right before you push your changes
- Resolve merge conflicts as they arise when you pull

For example,

```
git commit -m "Fix bug"  
git pull  
git push origin main
```

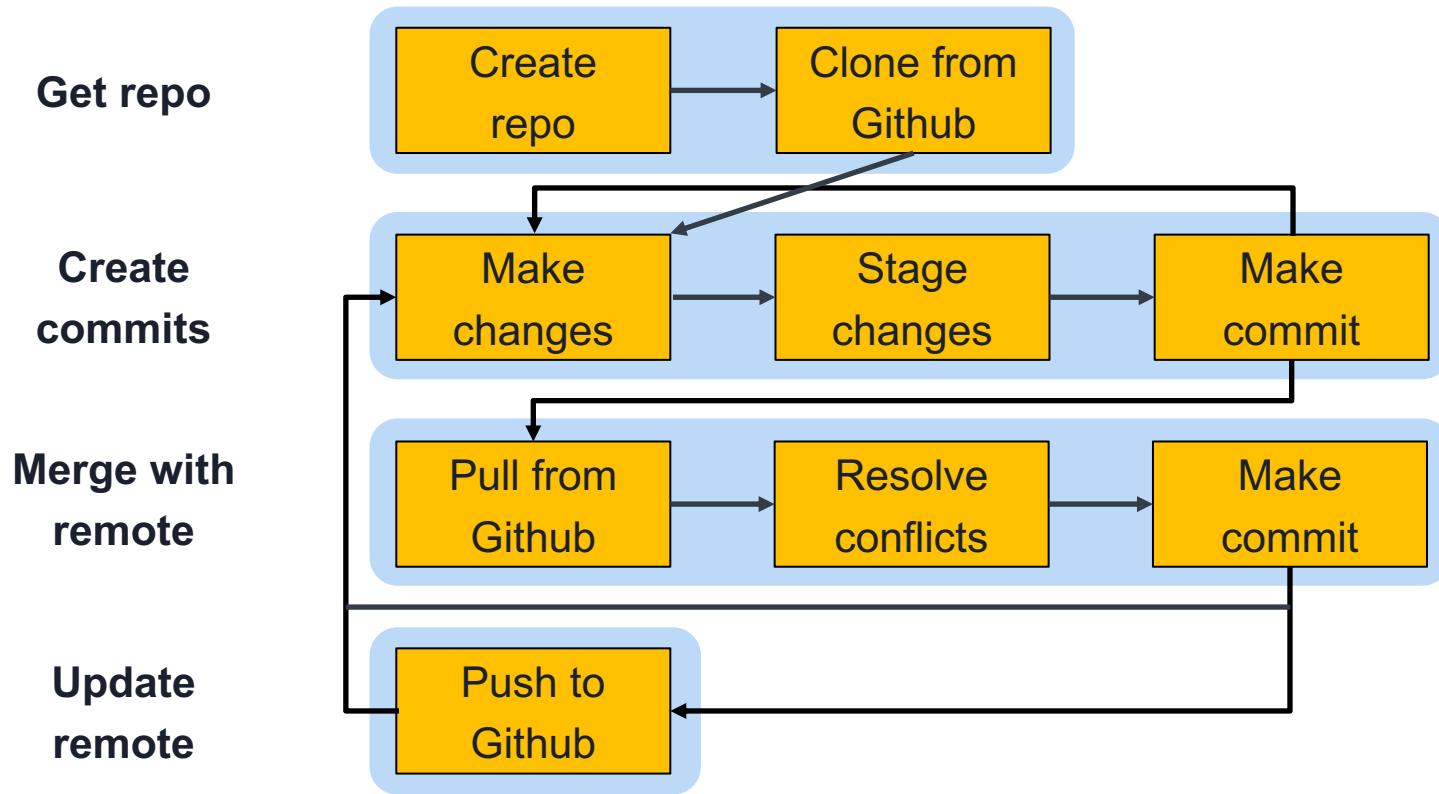
# Basic Workflow (Option 1)

Create a new local branch called `new_branch`, add some text to `newfile.txt` on `new_branch` and commit, merge the changes into `main`, pull any new changes from Github, and push to Github.

Step-by-step:

```
git checkout -b new_branch
nano mynewfile.txt
git add mynewfile.txt
git commit -m "Adding a text file"
git checkout main
git merge new_branch
git pull
git push origin main
```

# Basic Workflow (Option 1)



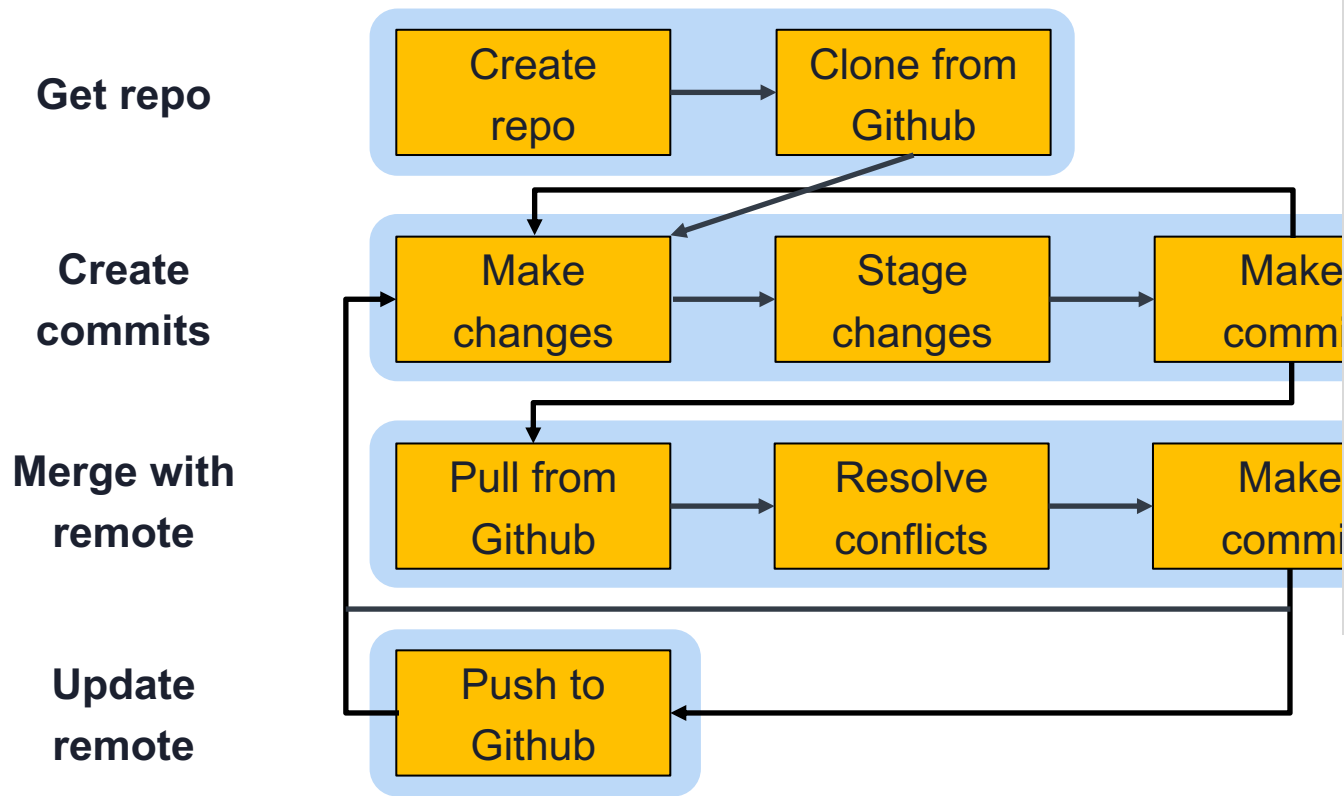
**When do you make a commit?**

- Think “atomic”
- Single bug fix, feature, refactor

**When do you push to Github?**

- Completed and tested feature, ready for use or review

# Basic Workflow (Option 1)



## When should I branch?

- May need to run `main`
- When you and a collaborator are working on different things

## Why might I push my branch to Github?

- If you want to collaborate on the branch
- As a code back-up!



# Social Coding Workflow (Option 2)

Thus far, we've focused on using Github with a small group of people who have shared access to a repository.

## **How can you contribute to projects in the broader Github community?**

- Not everyone has repo access, so we can't just push our changes
- Repository owner has a reviewing mechanism

# Social Coding Workflow (Option 2)

Create a new local branch called `new_branch`, add some text to `newfile.txt` on `new_branch` and commit, merge the changes into `main`, pull any new changes from Github, and push to Github.

Step-by-step:

```
git checkout -b new_branch
nano mynewfile.txt
git add mynewfile.txt
git commit -m "Adding a text file"
git push origin new_branch
<make "pull request" on Github>
```

# How to submit a pull request on Github

1. Navigate to your repository on Github and switch to this tab:

<> Code



Pull requests



Actions



Projects



Security



Insights

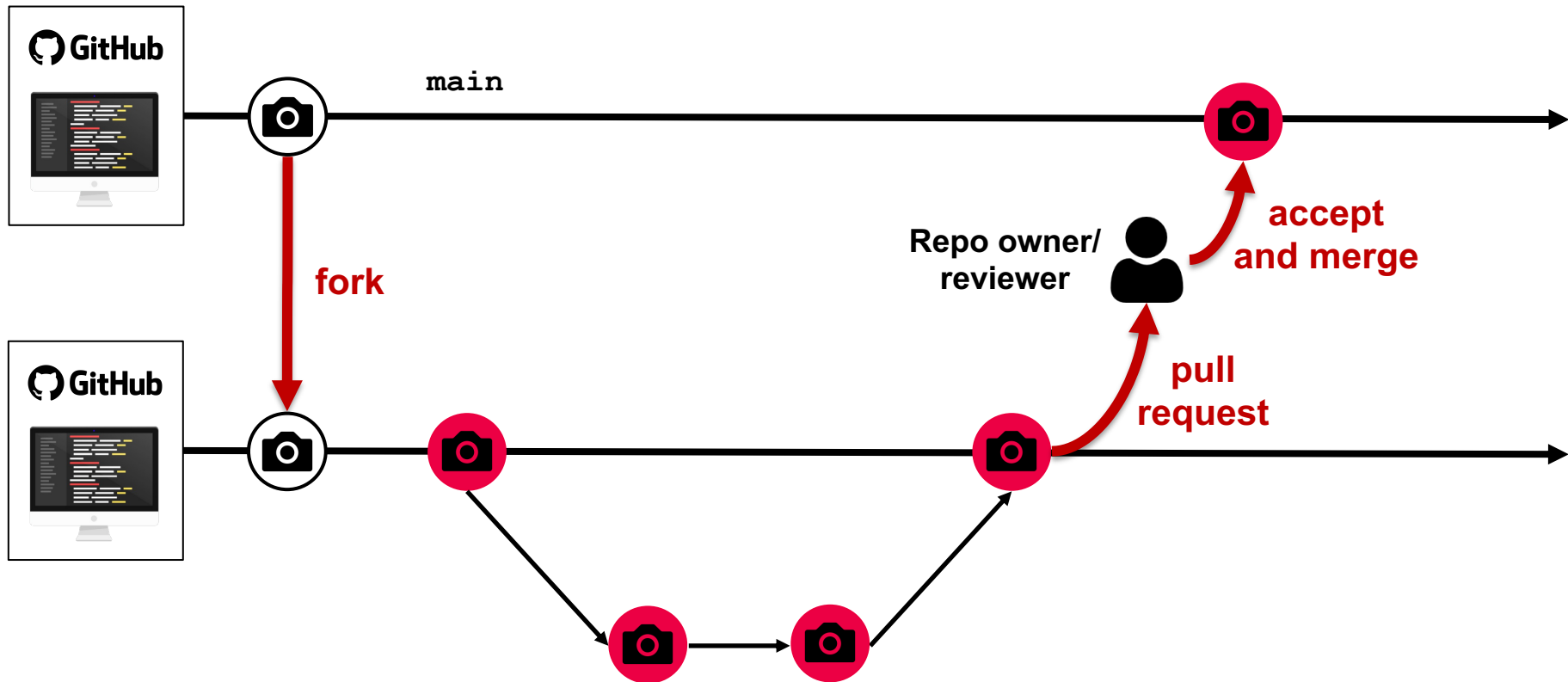


Settings

2. Click “New pull request” and follow prompts

New pull request

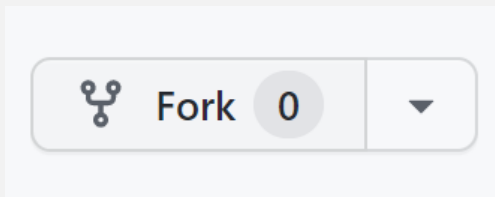
# Another Workflow – Forking / Pull Requests



# How to fork a repo on Github

Fork a repository from Github and clone your forked copy onto your local machine.

1. Go to github.com, login, and find the repo you want to clone
2. Click “Fork” in the right of the repo page



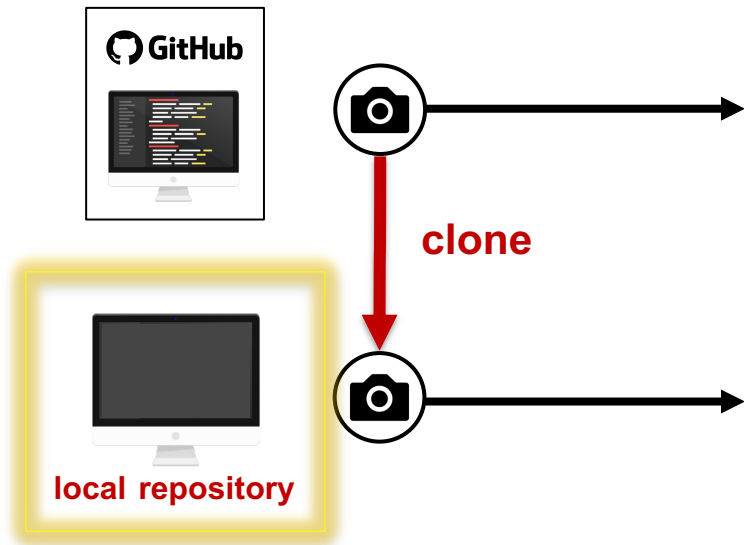
3. Clone the forked Github repository to your machine

```
git clone https://github.com/<yourusername>/<reponame>.git
```

# Cloning vs. Forking

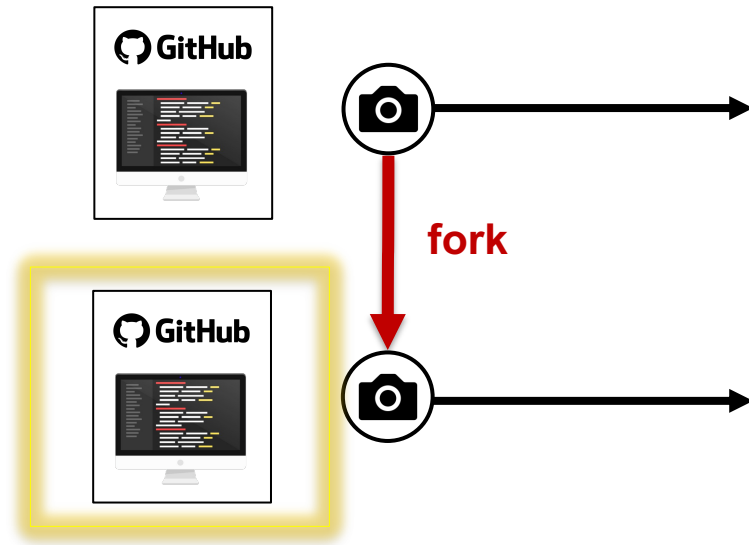
## Clone

- Creates a local copy of the repo
- Execute in terminal with git



## Fork

- Creates a copy of the repo on Github
- Execute in Github



# Question

Your collaborator gave you access to a repository that contains shared code. You want to be able to make your own changes and push them to the Github repo.

Should you clone or fork the repository?

A.) Clone

B.) Fork

# Question

You found a Github repo and think it would be a great starting point for a project you're working on.

Should you clone or fork the repository?

A.) Clone

B.) Fork (then clone)



# Question

You find a repo on Github that has some room for improvement, but to which you don't have access.

**What would happen if you tried to clone the repository?**

- A.) An error. You can't clone without access.
- B.) Git would create a local copy of the repo, but you couldn't push changes.
- C.) Git would create a local copy of the repo and you could push changes to the Github repo.

# LATEX

Adapted from slides and documents by Arthur Delarue

# Learning Objectives

At the end of the session, students will be able to...

- Organize and format a simple document in LaTeX, including tables, figures, and bibliography
- Add papers to their Zotero library, organize and categorize their papers, link their account to Overleaf, and directly cite their papers in LaTeX

# Motivation

## TeX

Donald Knuth created TeX to specify all kinds of formatting for documents.

Includes algorithms to make documents “beautiful” (kerning, spacing, word breaks).

## LaTeX

Leslie Lamport released LaTeX (Lamport’s TeX) which includes macros he built on top of TeX. This makes the writing process more content-focused instead of formatting-focused.

**To create a new “Intro” section:**

**TeX:** Specify larger font, bold, space before and after, section numbering convention, etc.

**LaTeX:** `\section{Introduction}`

# Benefits of LaTeX

- Separate content from formatting: type everything in plain text, then easily adjust formatting afterwards
- Easily format a wide range of documents with style templates
- Prettiest and easiest way to type up mathematical notation and formulas
- Helps keep citations and references in a consistent format

# Using LaTeX

## TeX distributions

- Installed locally on your machine
- Greater control of customization and formatting
- Fewer privacy concerns for sensitive documents



- Available online with no install needed
- Optimized for ease-of-use and a good place to learn
- Allows for easy collaboration, kind of like “Google Docs” for LaTeX

# Overleaf Demo

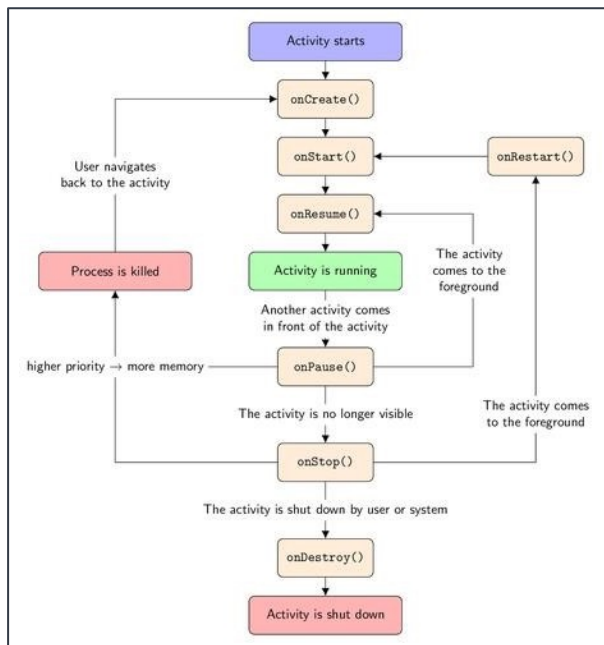
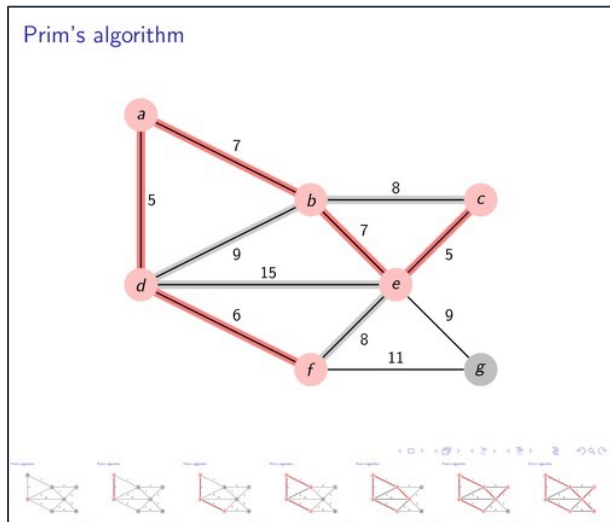
Let's go!

Code along by cloning the class Github repo and navigating to 1\_comp\_literacy/latexstuff. We will upload all of these files to Overleaf.

View project here: <https://www.overleaf.com/read/hhdngkdckcsm>

# Figures in TikZ

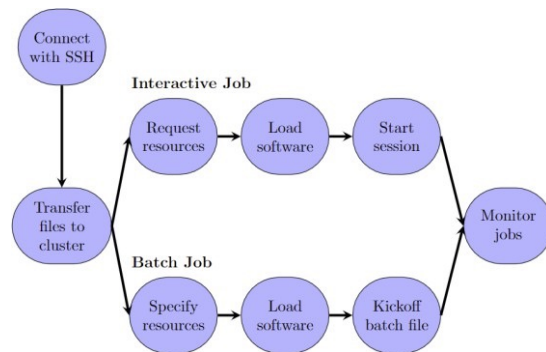
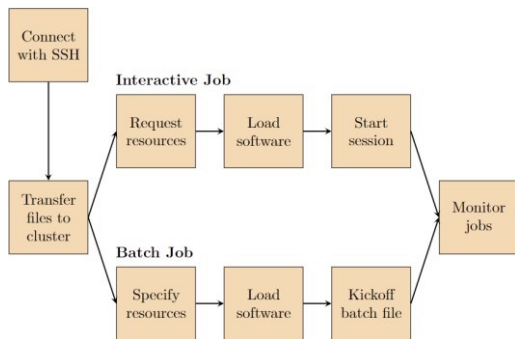
Make paper-ready figures using the TikZ package. Same “modular” formatting principle the rest of LaTeX.





# Figures in TikZ

- Specify style elements (red rounded boxes, thick black arrows, font size 12, etc.)
- Build your figure using the alias of each element (box, arrow, etc.)
- Easily change the formatting for all elements with one command, and TikZ will apply the new style to everything



# Key Takeaways

- LaTeX allows you to focus on content first, then easily adjust formatting
- Use Google to find commands, packages, and templates
- LaTeX is widely used: if you're doing a tedious task, there may be a simple tool to help if you search! (e.g. generate a large LaTeX table from an Excel table, translate markdown to LaTeX)

# **ZOTERO**

**Adapted from slides and documents by Alexandria Schmid**

# Motivation

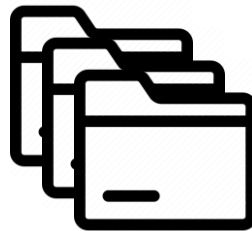
- It can be difficult to keep papers organized, both for specific literature reviews and for your general library
- Tools like Zotero, Mendeley, EndNote, and PaperPile are easy to use and can save lots of time and frustration



Add papers to library



Automatically capture  
citation information



Organize by project,  
topic, and tags

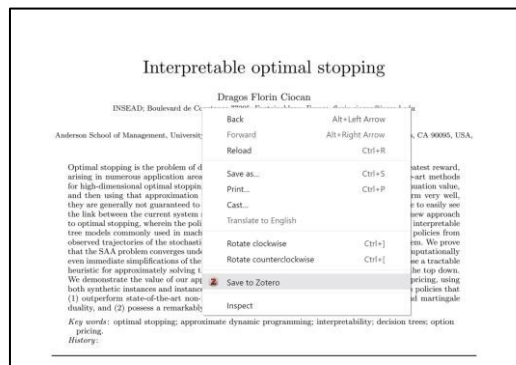
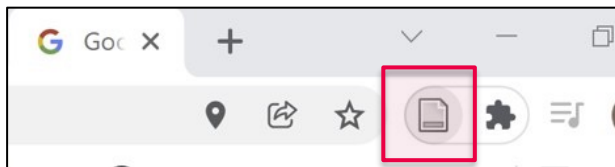


Connect to Overleaf to  
generate bibliographies

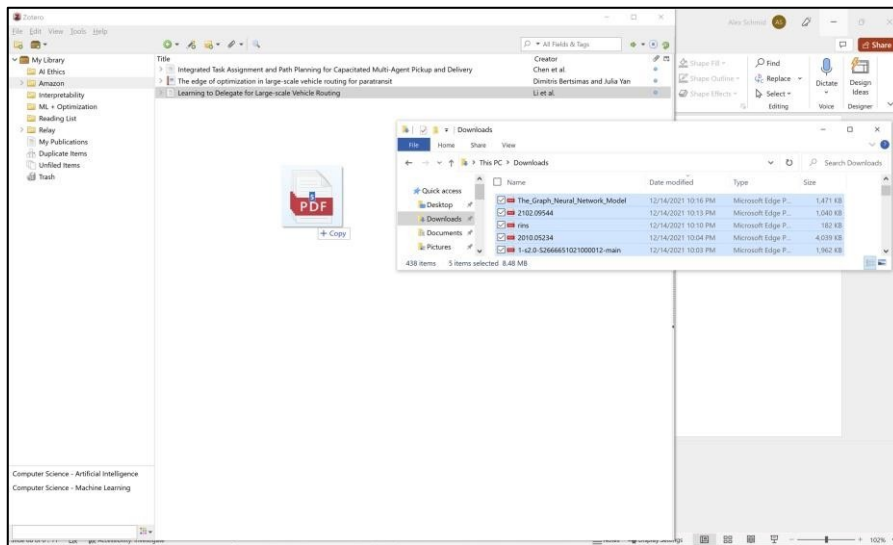
# Adding Papers

Download: <https://www.zotero.org/download/>

Add browser connector to add papers  
with button in search bar or with right click

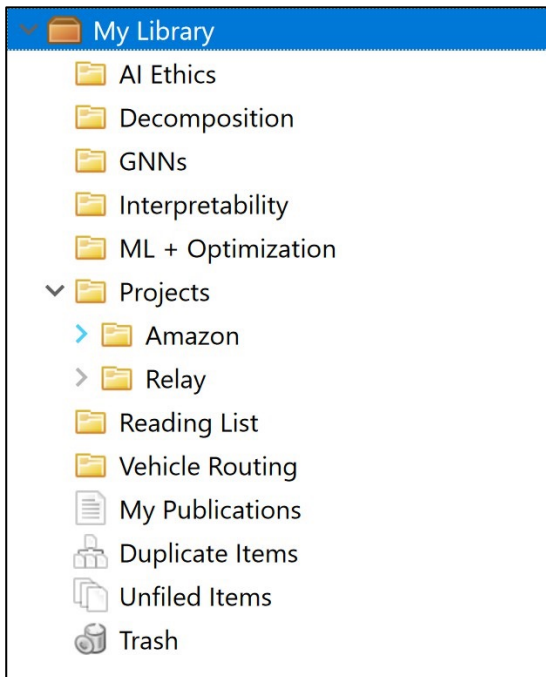


Drag downloaded papers into your  
Zotero library



# Organizing Your Library

Organize papers into folders  
by topic, project, etc.



## Organization Tips

- Keep a “reading list” to separate papers that you have not read yet and prune the list often. If you add lots of random, unread papers, you may end up with a cluttered and difficult to use library.
- Add tags and ensure citations are correct when you first add a paper, otherwise risk doing lots of organizational work later
- Link related papers to help jump start future literature reviews

# Connecting to Overleaf

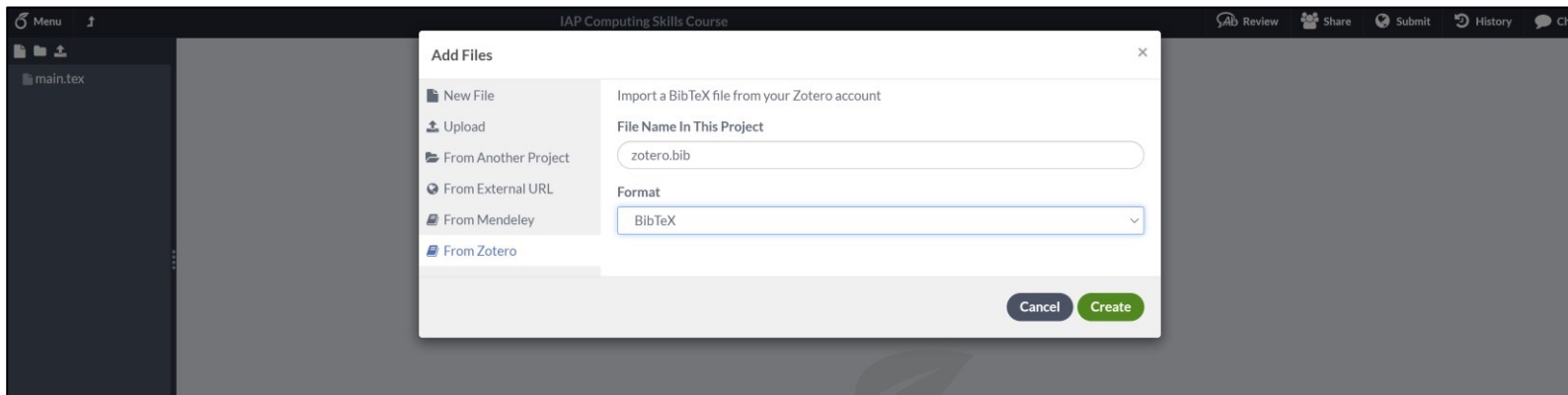
Overleaf > Account > Account Settings > Zotero Integration (or Mendeley)

## Zotero Integration

With Zotero integration you can import your references from Zotero into your Overleaf projects.

[Link to Zotero](#)

From your project, Upload > Import a .bib file containing your Zotero library



## More tips

- To change the reference label used in the Zotero BibTeX file for a paper, add “Citation key: <desired label>” to the “Extra” field of the paper’s Zotero info section.
- Mendeley also offers Overleaf integration, with the additional feature of allowing you to import BibTeX files for specific folders, rather than your entire library.



# Reference, reference, reference!

Now that you're **TeX-savvy**<sup>™</sup>, use your new skills to reference your Zotero papers in LaTeX documents with ease!