

# About this Site

## Contents

### Syllabus

- Computer Systems and Programming Tools
- Tools and Resources
- Grading
- Badge Deadlines and Procedures
- Detailed Grade Calculations
- Schedule
- Support Systems
- General Policies
- Office Hours & Communication

### Notes

- 1. Welcome, Introduction, and Setup
- 2. More orientation
- 3. Why Systems?
- 4. Git Offline
- 5. How do git branches work?
- 6. Terminal
- 7. What is a commit?
- 8. When do I get an advantage from git and bash?
- 9. Unix Philosophy
- 10. Unix Philosophy In Practice
- 11. How do programmers communicate about code?
- 12. What *is* git?
- 13. How does git make a commit?
- 14. How can I automate things with bash?
- 15. Why did we learn the plumbing commands?
- 16. How can I work on a remote server?
- 17. What happens when I build code in C?
- 18. How can I authenticate more securely from a terminal?
- 19. What happens when we run code?
- 20. What *is* an IDE?

### Activities

- KWL Chart

- Team Repo
- Review Badges
- Prepare for the next class
- Practice Badges
- Index
- Explore Badges
- Build Badges

## FAQ

- Syllabus and Grading FAQ
- Git and GitHub
- Other Course Software/tools

## Resources

- Glossary
- General Tips and Resources
- How to Study in this class
- GitHub Interface reference
- Language/Shell Specific References
- Getting Help with Programming
- Getting Organized for class
- More info on cpus
- Windows Help & Notes
- Advice from Spring 2022 Students
- Advice from Fall 2022 Students
- Advice from Spring 2023 Students
- Advice from Fall 2023 Students

Welcome to the course website for Computer Systems and Programming Tools in Spring 2025.

This class meets TuTh 12:30PM - 1:45PM in Ranger 302 and lab on Monday 3:00PM - 4:45PM in Ranger 202.

This website will contain the syllabus, class notes, and other reference material for the class.

## Navigating the Sections

The Syllabus section has logistical operations for the course broken down into sections. You can also read straight through by starting in the first one and navigating to the next section using the arrow navigation at the end of the page.

This site is a resource for the course. We do not follow a text book for this course, but all notes from class are posted in the notes section, accessible on the left hand side menu, visible on large screens and in the menu on mobile.

The resources section has links and short posts that provide more context and explanation. Content in this section is for the most part not strictly the material that you'll be graded on, but it is often material that will help you understand and grow as a programmer and data scientist.

# Reading each page

Some pages of the syllabus and resources are also notebooks, if you want to see behind the curtain of how I manage the course information.

```
# this is a comment in a code block  
command argument --option -a
```

command output  
important line, emphasized

## 🔔 Try it Yourself

Notes will have exercises marked like this

## 🔔 Question from Class

Questions that are asked in class, but unanswered at that time will be answered in the notes and marked with a box like this. Long answers will be in the main notes

## 🔔 Further reading

Notes that are mostly links to background and context will be highlighted like this. These are optional, but will mostly help you understand code excerpts they relate to.

## 💡 Hint

Both notes and assignment pages will have hints from time to time. Pay attention to these on the notes, they'll typically relate to things that will appear in the assignment.

## 🔔 Click here!

Special tips will be formatted like this

## 🔔 Check your Comprehension

Questions to use to check your comprehension will looklike this

## 🔔 Contribute

Chances to earn community badges will sometimes be marked like this

## 🔔 Ques

Question  
but una  
answer  
with a b  
will be i

# Computer Systems and Programming Tools

## About this course

In this course we will study the tools that we use as programmers and use them as a lens to study the computer system itself. We will begin with two fundamental tools: version control and the [shell](#). We will focus on [git](#) and [bash](#) as popular examples of each. Sometimes understanding the tools requires understanding an aspect of the system, for example [git](#) uses cryptographic [hashing](#) which requires understanding number systems. Other times the tools helps us see how parts work: the [shell](#) is our interface to the operating system.

## About this syllabus

This syllabus is a *living* document. You can get notification of changes from GitHub by “watching” the [repository](#). You can view the date of changes and exactly what changes were made on the Github [repository](#) page.

Creating an [issue](#) is also a good way to ask questions about anything in the course it will prompt additions and expand the FAQ section.

### Should you download the syllabus and rely on your offline copy?

No, because the syllabus changes

## About your instructor

Name: Dr. Sarah M Brown

Dr. Sarah M Brown is a third year Assistant Professor of Computer Science, who does research on how social context changes machine learning. Dr. Brown earned a PhD in Electrical Engineering from Northeastern University, completed a postdoctoral fellowship at University of California Berkeley, and worked as a postdoctoral research associate at Brown University before joining URI. At Brown University, Dr. Brown taught the Data and Society course for the Master's in Data Science Program. You can learn more about me at my [website](#) or my research on my [lab site](#).

Name: Ayman Sandouk Office hours: listed on communication page

Ayman is a Masters student at the University of Rhode Island with Bachelors in CS from URI. Ayman's research is currently focusing on benchmarking LLMs for fairness

The best way to contact me is e-mail or an [issue](#) on an assignment repo. For more details, see the [Communication Section](#)

# Land Acknowledgement

## Important

The University of Rhode Island land acknowledgment is a statement written by members of the University community in close partnership with members of the Narragansett Tribe. For more information see [the university land acknowledgement page](#)

The University of Rhode Island occupies the traditional stomping ground of the Narragansett Nation and the Niantic People. We honor and respect the enduring and continuing relationship between the Indigenous people and this land by teaching and learning more about their history and present-day communities, and by becoming stewards of the land we, too, inhabit.

## Tools and Resources

We will use a variety of tools to conduct class and to facilitate your programming. You will need a computer with Linux, MacOS, or Windows. It is unlikely that a tablet will be able to do all of the things required in this course. A Chromebook may work, especially with developer tools turned on. Ask Ayman if you need help getting access to an adequate computer.

All of the tools and resources below are either:

- paid for by URI **OR**
- freely available online.

## BrightSpace

On BrightSpace, you will find links to other resource, this site and others. Any links that are for private discussion among those enrolled in the course will be available only from Brightspace.

## Prismia chat

Our class link for [Prismia chat](#) is available on Brightspace. Once you've joined once, you can use the link above or type the url: prismia.chat. We will use this for chatting and in-class understanding checks.

On Prismia, all students see the instructor's messages, but only the Instructor and TA see student responses.

## Important

Prismia is **only** for use during class, we do not read messages there outside of class time

You can get a transcript from class from Prismia.chat using the menu in the top right.

## Course Website

The course website will have content including the class policies, scheduling, class notes, assignment information, and additional resources.

## Note

Seeing  
require  
SSO a  
course

Links to the course reference text and code documentation will also be included here in the assignments and class notes.

## GitHub

You will need a [GitHub](#) Account. If you do not already have one, please [create one](#) by the first day of class. If you have one, but have not used it recently, you may need to update your password and login credentials as the [Authentication rules](#) changed in Summer 2021.

You will also need the [gh CLI](#). It will help with authentication and allow you to work with other parts of [GitHub](#) besides the core [git](#) operations.

### Important

You need to install this on Mac

## Programming Environment

In this course, we will use several programming environments. In order to participate in class and complete assignments you need the items listed in the requirements list. The easiest way to meet these requirements is to follow the recommendations below. I will provide instruction assuming that you have followed the recommendations. We will add tools throughout the semester, but the following will be enough to get started.

### Warning

This is not technically a *programming* class, so you will not need to know how to write code from scratch in specific languages, but we will rely on programming environments to apply concepts.

## Requirements:

- Python with scientific computing packages (numpy, scipy, jupyter, pandas, seaborn, sklearn)
- a C compiler
- [Git](#)
- access to a bash [shell](#)
- A high compatibility web browser (Safari will sometimes fail; Google Chrome and Microsoft Edge will; Firefox probably will)
- [nano text editor](#) (comes with GitBash and default on MacOS)
- one IDE with [git](#) support (default or via extension)
- [the GitHub CLI](#) on all OSs

## Recommendation

[Windows- option A](#)   [Windows - option B](#)   [MacOS](#)   [Linux](#)   [Chrome OS](#)

- If you will not do any side projects, install python via [Anaconda video install](#)
- Otherwise, use the [base python installer](#) and then install libraries with pip

- Git and Bash with [GitBash](#) ([video instructions](#)).

## Zoom

(backup only & office hours only)

This is where we will meet if for any reason we cannot be in person. You will find the link to class zoom sessions on Brightspace.

URI provides all faculty, staff, and students with a paid Zoom account. It can run in your browser or on a mobile device, but you will be able to participate in office hours and any online class sessions if needed best if you download the [Zoom client](#) on your computer. Please [log in](#) and [configure your account](#). Please add a photo (can be yourself or something you like) to your account so that we can still see your likeness in some form when your camera is off. You may also wish to use a virtual background and you are welcome to do so.

For help, you can access the [instructions provided by IT](#).

## Grading

This section of the syllabus describes the principles and mechanics of the grading for the course. The course is designed around your learning so the grading is based on you demonstrating how much you have learned.

Additionally, since we will be studying programming tools, we will use them to administer the course. To give you a chance to get used to the tools there will be a grade free zone for the first few weeks.

Each section be viewed at two levels of detail. You can toggle the tabs and then the whole page will be at the level of your choice as you scroll.

**TL;DR**

**Full Detail**

this will be short explanations; key points you should **remember**

## Learning Outcomes

**TL;DR**

**Full Detail**

The goal is for you to learn and the grading is designed to as close as possible actually align to how much you have learned.

You should be a more independent and efficient developer and better collaborator on code projects by the end of the semester.

## Principles of Grading

**TL;DR**

**Full Detail**

- Learning happens with practice and feedback
- I value **learning** not perfect performance or productivity
- a C means you can follow a conversation about the material, but might need help to apply it
- a B means you can *also* apply it in basic scenarios or if the problem is broken down
- an A means you can *also* apply it in complex scenarios independently

*please do not make me give you less than a C, but a D means you showed up basically, but you may or may not have actually retained much*

The course is designed to focus on **success** and accumulating knowledge, not taking away points.

#### If you made an error in an assignment what do you need to do?

^

Read the suggestions and revise the work until it is correct.

## Penalty-free Zone

**TL;DR**

Full Detail

We will use developer tools to do everything in this class; in the long term this will benefit you, but it makes the first few weeks hard, so **mistakes in the first few weeks cannot hurt your grade** as long as you learn eventually.

Deadlines are *extra flexible* for 3 weeks while you figure things out.

#### What happens if you merged a PR without feedback?

^

During the Penalty-Free zone, we will help you figure that out and fix it so you get credit for it. After that, you have to fix it on your own (or in office hours) in order to get credit.

#### Important

If there are terms in the rest of this section that do not make sense while we are in the penalty-free zone, do not panic. This zone exists to help you get familiar with the terms needed.

#### What happens if you're confused by the grading scheme right now?

^

Nothing to worry about, we will review it again in week three after you get a chance to build the right habits and learn vocabulary. There will also be a lab activity that helps us to be sure that you understand it at that time.

## Learning Badges

**TL;DR**

Full Detail

Different badges are different levels of complexity and map into different grades.

- experience: like attendance
- lab: show up & try
- review: understand what was covered in class
- practice: apply what was covered in class
- explore: get a mid-level understanding of a topic of your choice
- build: get a deep understanding of a topic of your choice

To pass:

- 22 experience badges
- 12 lab checkouts

Add 18 review for a C or 18 practice for a B.

For an A you can choose:

- 18 review + 3 build
- 18 practice + 6 explore

**you can mix & match, but the above plans are the simplest way there**

### **Warning**

These counts assume that the semester goes as planned and that there are 26 available badges of each base type (experience, review, practice). If the number of available badges decreases by more than 2 for any reason (eg snowdays, instructor illness, etc) the threshold for experience badges will be decreased.

All of these badges will be tracked through PRs in your kwl repo. Each PR must have a title that includes the badge type and associated date. We will use scripts over these to track your progress.

### **Important**

There will be 20 review and practice badges available after the penalty free zone. This means that missing the review and practice badges in the penalty free zone cannot hurt you. However, it does not mean it is a good idea to not attempt them, not attempting them at all will make future badges harder, because reviewing early ideas are important for later ideas.

You cannot earn both practice and review badges for the same class session, but most practice badge requirements will include the review requirements plus some extra steps.

In the second half of the semester, there will be special *integrative* badge opportunities that have multipliers attached to them. These badges will count for more than one. For example an integrative 2x review badge counts as two review badges. These badges will be more complex than regular badges and therefore count more.

### **Can you do any combination of badges?**



No, you cannot earn practice and review for the same date.

# Experience Badges

## In class

You earn an experience badge in class by:

- preparing for class
- following along with the activity (creating files, using git, etc)
- responding to 80% of inclass questions (even incorrect, `\idk`, `\dgt`)
- reflecting on what you learned
- asking a question at the end of class

## Makeup

You can make up an experience badge by:

- preparing for class
- reading the posted notes
- completing the activity from the notes
- completeeing an “experience report”
- attaching evidence as indiated in notes OR attending office hours to show the evidence

### 💡 Tip

On prismia questions, I will generally give a “Last chance to get an answer in” warning before I resume instruction. If you do not respond at all too many times, we will ask you to follow the makeup procedure instead of the In Class proccedure for your experience badge.

To be sure that your response rate is good, if you are paying attention, but do not have an answer you can use one of the following special commands in prismia:

- `\idk`: “I am paying attention, but do not know how to answer this”
- `\dgt`: “I am paying attention, not really confused, but ran out of time trying to figure out the answer”

you can send these as plain text by pressing `enter` (not Mac) or `return` (on Mac) to send right away or have them render to emoji by pressing `tab`

An experience report is evidence you have completed the activity and reflection questions. The exact form will vary per class, if you are unsure, reach out ASAP to get instructions. These are evaluated only for completeness/ good faith effort. Revisions will generally not be required, but clarification and additional activity steps may be advised if your evidence suggests you may have missed a step.

### 🔔 Do you earn badges for prepare for class?

^

No, prepare for class tasks are folded into your experience badges.

### What do you do when you miss class?

^

Read the notes, follow along, and produce and experience report or attend office hours.

### What if I have no questions?

^

Learning to ask questions is important. Your questions can be clarifying (eg because you misunderstood something) or show that you understand what we covered well enough to think of hypothetical scenarios or options or what might come next. Basically, focused curiosity.

## Lab Checkouts

You earn credit for lab by attending and completing core tasks as defined in a lab issue posted to your repo each week. Work that needs to be correct through revisions will be left to a review or practice badge.

You will have to have a short meeting with a TA or instructor to get credit for each lab. In the lab instructions there will be a checklist that the TA or instructor will use to confirm you are on track. In these conversations, we will make sure that you know how to do key procedural tasks so that you are set up to continue working independently.

To make up a lab, complete the tasks from the lab issue on your own and attend office hours to complete the checkout.

## Review and Practice Badges

The tasks for these badges will be defined at the bottom of the notes for each class session *and* aggregated to badge-type specific pages on the left hand side fo the course website.

You can earn review and practice badges by:

- creating an [issue](#) for the badge you plan to work on
- completing the tasks
- submitting files to your KWL on a new [branch](#)
- creating a PR, linking the [issue](#), and requesting a review
- revising the PR until it is approved
- merging the PR after it is approved

### Where do you find assignments?

^

At the end of notes and on the separate pages in the activities section on the left hand side

## You should create one PR per badge

The key difference between review and practice is the depth of the activity. Work submitted for review and practice badges will be assessed for correctness and completeness. Revisions will be common for these activities, because understanding correctly, without misconceptions, is important.

### Important

Revisions are to help you improve your work **and** to get used to the process of making revisions. Even excellent work can be improved. The **process** of making revisions and taking good work to excellent or excellent to exceptional is a useful learning outcome. It will help you later to be really good at working through PR revisions; we will use the same process as code reviews in industry, even though most of it will not be code alone.

## Explore Badges

Explore badges require you to pose a question of your own that extends the topic. For inspiration, see the practice tasks and the questions after class.

Details and more ideas are on the [explore](#) page.

You can earn an explore badge by:

- creating an [issue](#) proposing your idea (consider this ~15 min of work or less)
- adjusting your idea until given the proceed label
- completing your exploration
- submitting it as a PR
- making any requested changes
- merging the PR after approval

For these, ideas will almost always be approved, the proposal is to make sure you have the right scope (not too big or too small). Work submitted for explore badges will be assessed for depth beyond practice badges and correctness. Revisions will be more common on the first few as you get used to them, but typically decrease as you learn what to expect.

### Important

Revisions are to help you improve your work **and** to get used to the process of making revisions. Even excellent work can be improved. The **process** of making revisions and taking good work to excellent or excellent to exceptional is a useful learning outcome. It will help you later to be really good at working through PR revisions; we will use the same process as code reviews in industry, even though most of it will not be code alone.

## You should create one PR per badge

## Build Badges

Build badges are for when you have an idea of something you want to do. There are also some ideas on the [build](#) page.

You can earn a build badge by:

- creating an [issue](#) proposing your idea and iterating until it is given the “proceed” label
- providing updates on your progress
- completing the build
- submitting a summary report as a PR linked to your proposal [issue](#)
- making any requested changes

- merging the PR after approval

## You should create one PR per badge

For builds, since they're bigger, you will propose intermediate milestones. Advice for improving your work will be provided at the milestones and revisions of the complete build are uncommon. If you do not submit work for intermediate review, you may need to revise the complete build. The build proposal will be assessed for relevance to the course and depth. The work will be assessed for completeness in comparison to the proposal and correctness. The summary report will be assessed only for completeness, revisions will only be requested for skipped or incomplete sections.

## Community Badges

**TL;DR** [Full Detail](#)

These are like extra credit, they have very limited ability to make up for missed work, but can boost your grade if you are on track for a C or B.

## 🎁 Free corrections

**TL;DR** [Full Detail](#)

If you get a 🎁 apply the changes to get credit.

### ❗ Important

These free corrections are used at the instructional team's discretion and are not guaranteed.

This means that, for example, the first time you make a particular mistake, might get a 🎁, but the second time you will probably get a hint, and a third or fourth time might be a regular revision with a comment like [see #XX and fix accordingly](#) where XX is a link to a previous badge.

### ℹ Note

We do URI as class. T exam w all work

### 🔔 IDEA



If the course response rate on the IDEA survey is about 75%, 🎁 will be applicable to final grading. **this includes the requirement of the student to reply**

## Ungrading Option

**TL;DR** [Full Detail](#)

You should try to follow the grading above; but sometimes weird things happen. I care that you learn.

If you can show you learned in some other way besides earning the badges above you may be able to get a higher grade than your badges otherwise indicate.

## 💡 What do you think?

^

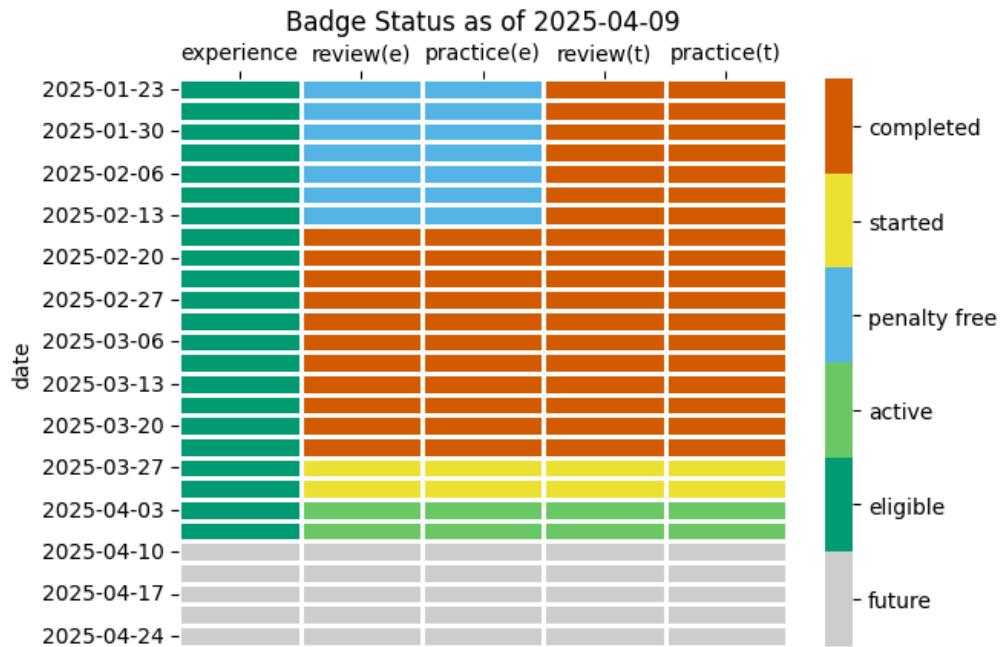
share your thoughts on this option [in the discussions for the class](#) and then log it for a community badge!

# Badge Deadlines and Procedures

This page includes more visual versions of the information on the [grading](#) page. You should read both, but this one is often more helpful, because some of the processes take a lot of words to explain and make more sense with a diagram for a lot of people.

▶ Show code cell source

```
Text(0.5, 1.0, 'Badge Status as of 2025-04-09')
```



## Getting Feedback

Who should you request/assign?

course item type	issue assignee	PR reviewer
prepare work	not required; can be student	none requierd; merge to experience branch
experience badge	N/A	TA assigned to your group (will be automatic after a few classes)
practice badge	not required; can be student	@instructors (will then convert to 1/3 people)
review badge	not required; can be student	@instructors (will then convert to 1/3 people)
explore badge	proposal, assigned to @AymanBx (also student optionally)	@AymanBx
build badge	proposal, assigned to @AymanBx (also student optionally)	@AymanBx
anything merged pre-emptively in penalty free	@AymanBx	clear others

## Deadlines

We do not have a final exam, but URI assigns an exam time for every class. The date of that assigned exam will be the final due date for all work including all revisions.

## Experience badges

Prepare for class tasks must be done before class so that you are prepared. Missing a prepare task could require you to do an experience report to make up what you were not able to do in class.

If you miss class, the experience report should be at least attempted/drafted (though you may not get feedback/confirmation) before the next class that you attend. This is strict, not as punishment, but to ensure that you are able to participate in the next class that you attend. Skipping the experience report for a missed class, may result in needing to do an experience report for the next class you attend to make up what you were not able to complete due to the missing class activities.

If you miss multiple classes, create a catch-up plan to get back on track by contacting instructor.

## Review and Practice Badges

These badges have 5 stages:

- posted: tasks are on the course website and an [issue](#) is created
- started: one task is attempted and a draft PR is open
- completed: all tasks are attempted PR is ready for review, and a review is requested
- earned: PR is approved (by instructor or a TA) and work is merged

### 💡 Tip

these badges *should* be started before the next class. This will set you up to make the most out of each class session. However, only prepare for class tasks have to be done immediately.

These badges must be *started* within one week of when they are posted (2pm) and *completed* within two weeks. A task is attempted when you have answered the questions or submitted evidence of doing an activity or asked a sincere clarifying question.

If a badge is planned, but not started within one week it will become expired and ineligible to be earned. You may request extensions to complete a badge by updating the PR message, these will typically be granted. Extensions for starting badges will only be granted in exceptional circumstances.

Expired badges will receive a comment and be closed

Once you have a good-faith attempt at a complete badge, you have until the end of the semester to finish the revisions in order to *earn* the badge.

### 💡 Tip

Try to complete revisions quickly, it will be easier for you

## Explore Badges

Explore badges have 5 stages:

- proposed: issue created
- in progress: issue is labeled “proceed” by the instructor
- complete: work is complete, PR created, review requested
- revision: “request changes” review was given
- earned: PR approved

Explore badges are feedback-limited. You will not get feedback on subsequent explore badge proposals until you earn the first one. Once you have one earned, then you can have up to two in progress and two in revision at any given time. At most, you will receive feedback for one explore badge per week, so in order to earn six, your first one must be complete by March 18.

## Build Badges

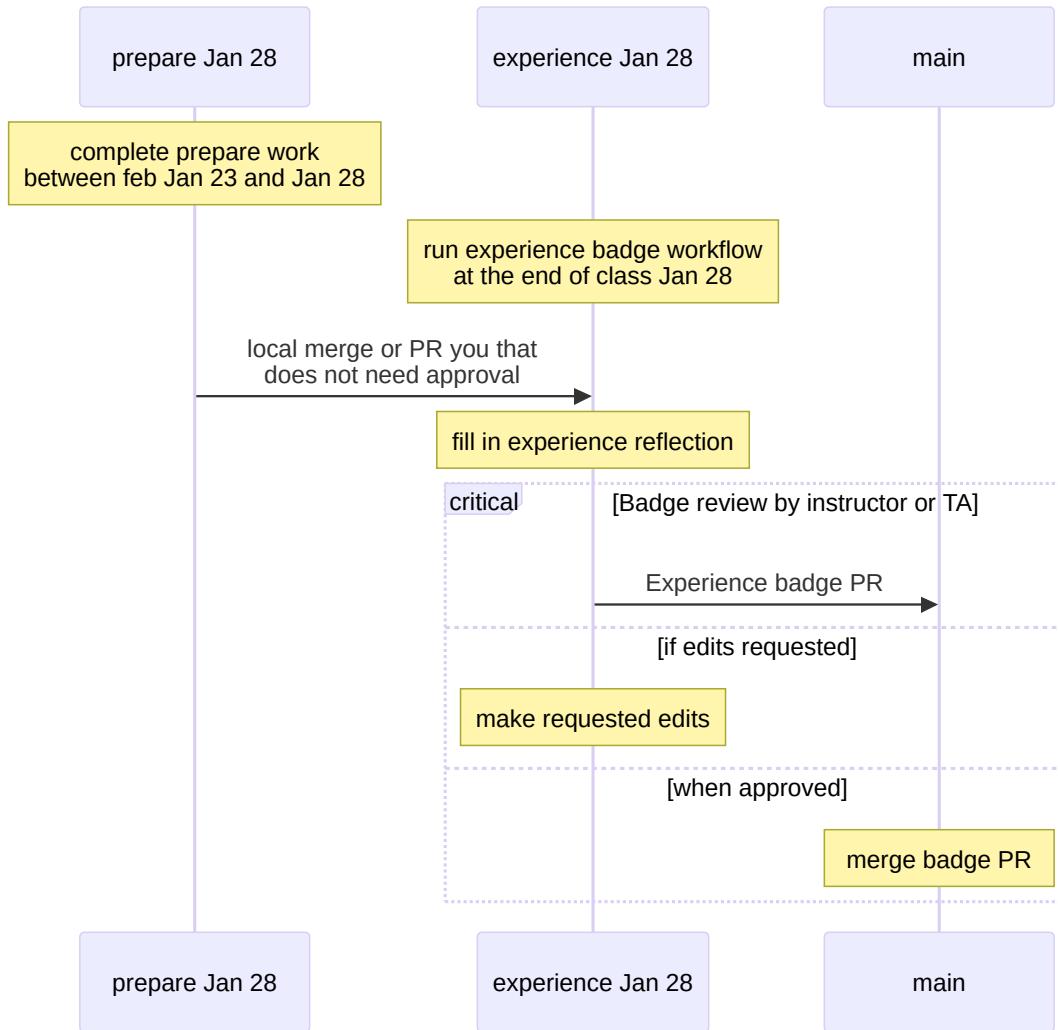
At most one build badge will be evaluated every 4 weeks. This means that if you want to earn 3 build badges, the first one must be in 8 weeks before the end of the semester, March 4. The second would be due April 1st, and the third submitted by the end of classes, April 29th.

## Procedures

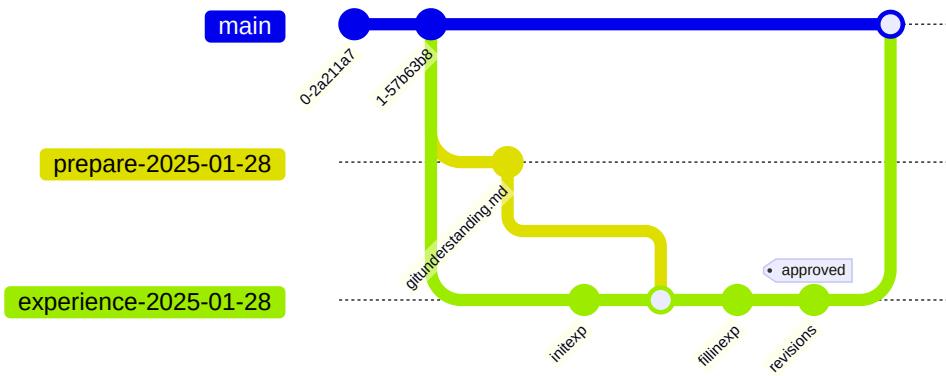
### Prepare work and Experience Badges Process

This is for a single example with specific dates, but it is similar for all future dates

The columns (and purple boxes) correspond to branches in your KWL repo and the yellow boxes are the things you have to do. The “critical” box is what you have to wait for us on. The arrows represent PRs (or a local merge for the first one)



In the end the commit sequence for this will look like the following:



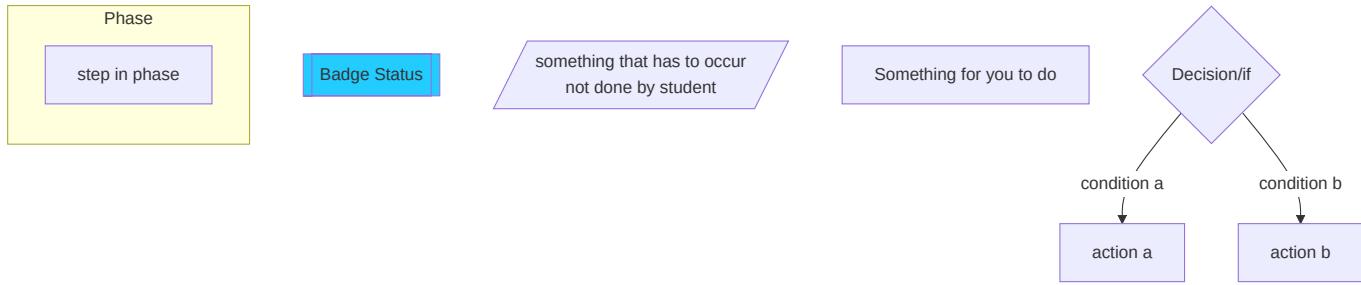
You can merge the prepare into the experience with a PR or on the command line, your choice.

Where the “approved” tag represents and approving review on the PR.

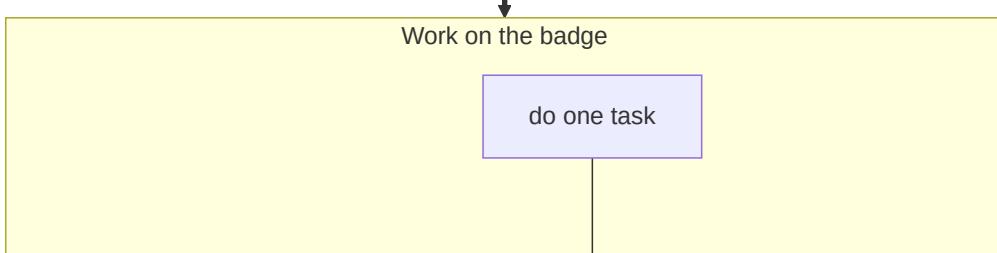
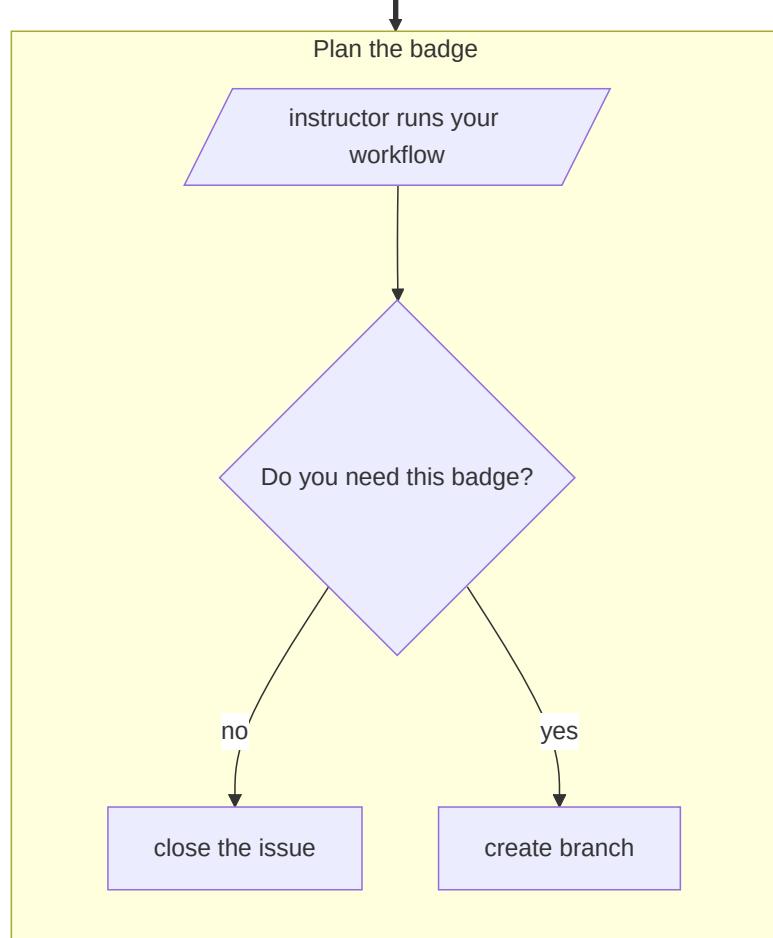
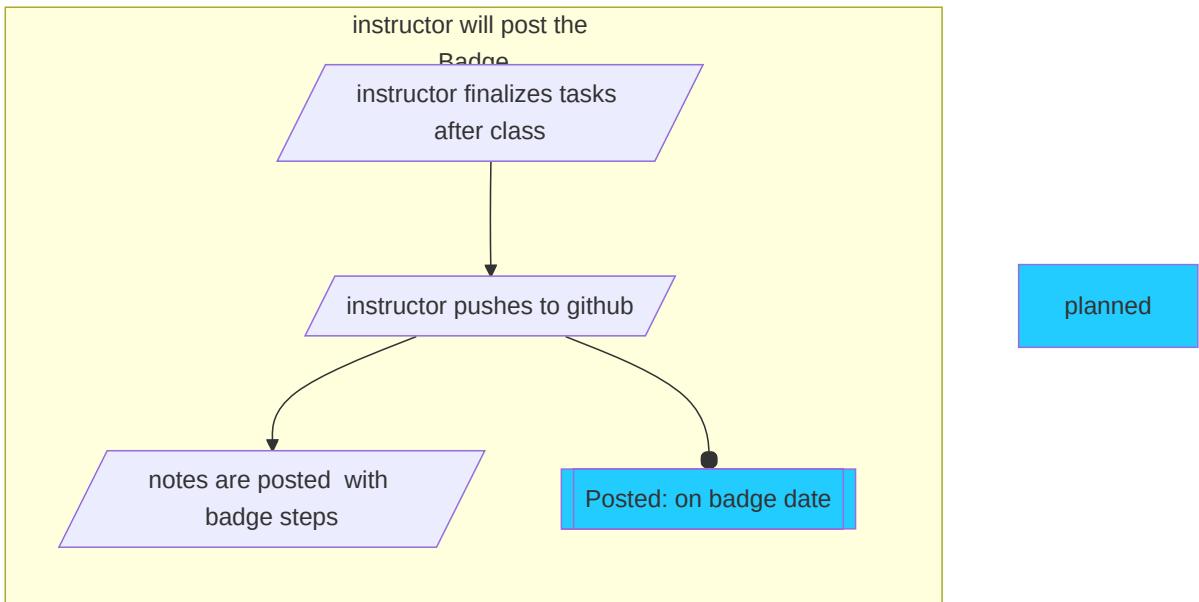
You can, once you know how, do this offline and do the merge with in the CLI instead of with a PR.

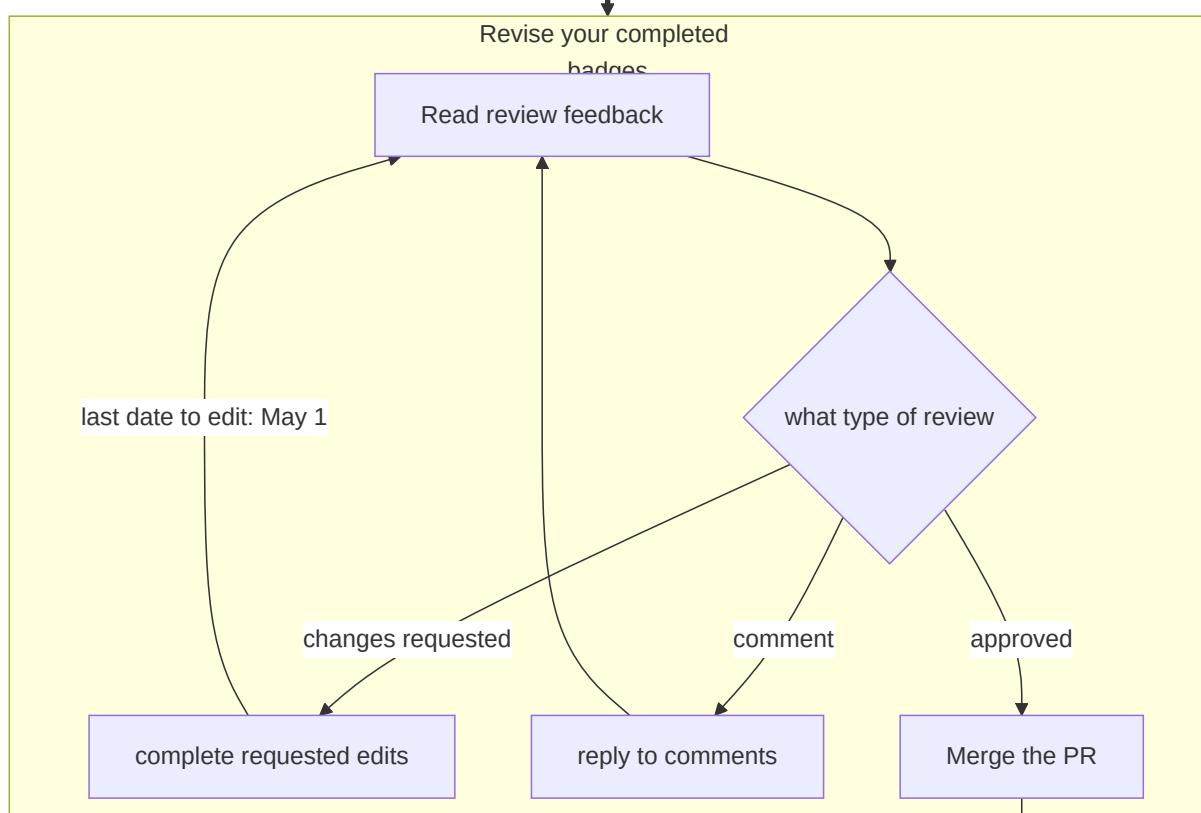
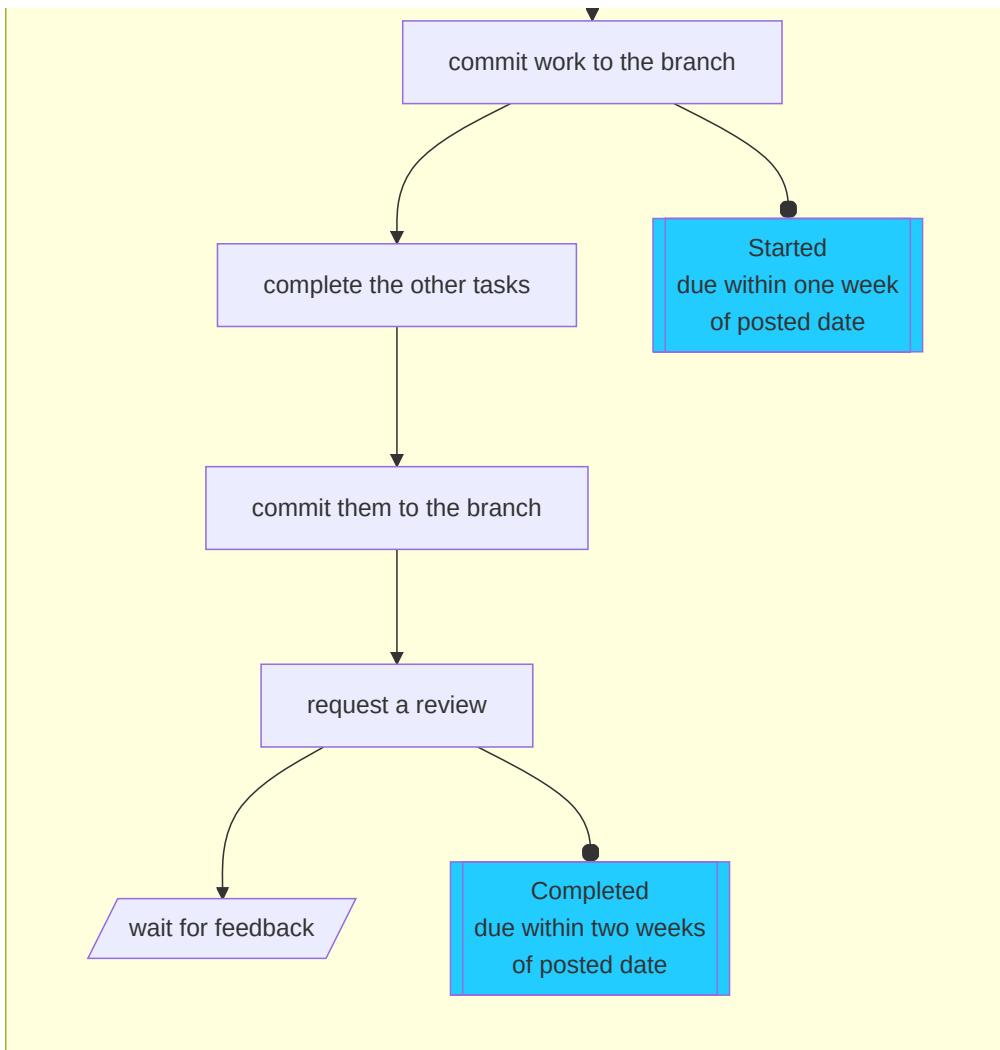
## Review and Practice Badge

Legend:



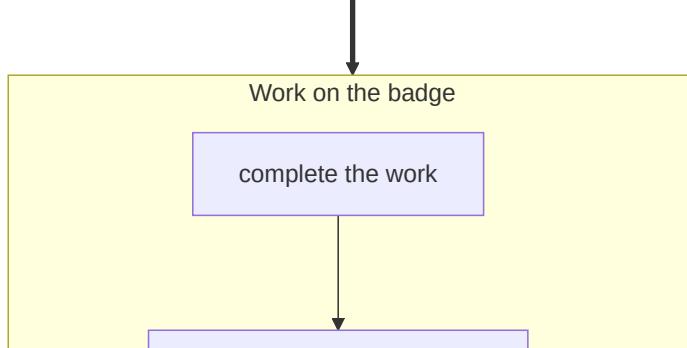
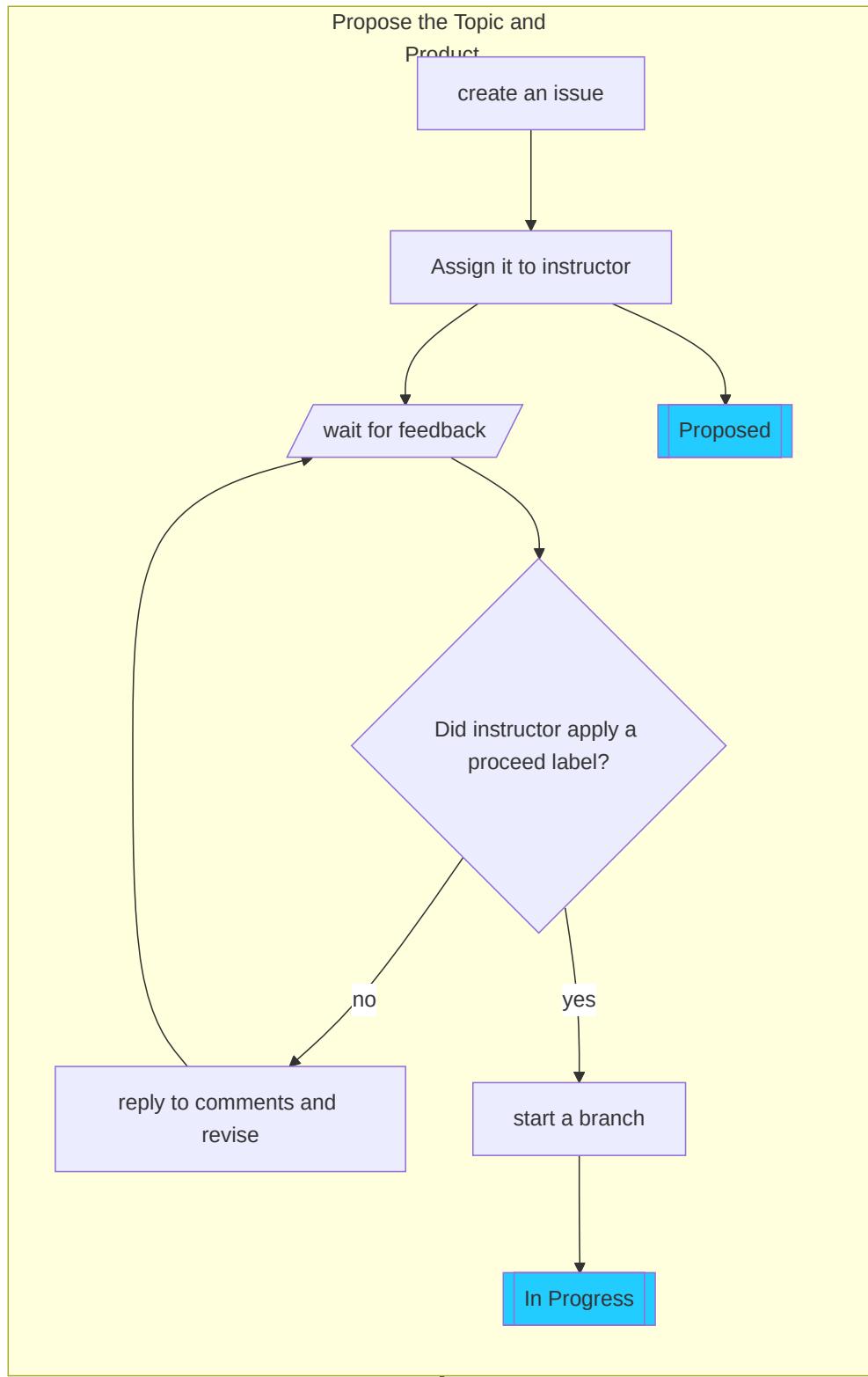
This is the general process for review and practice badges

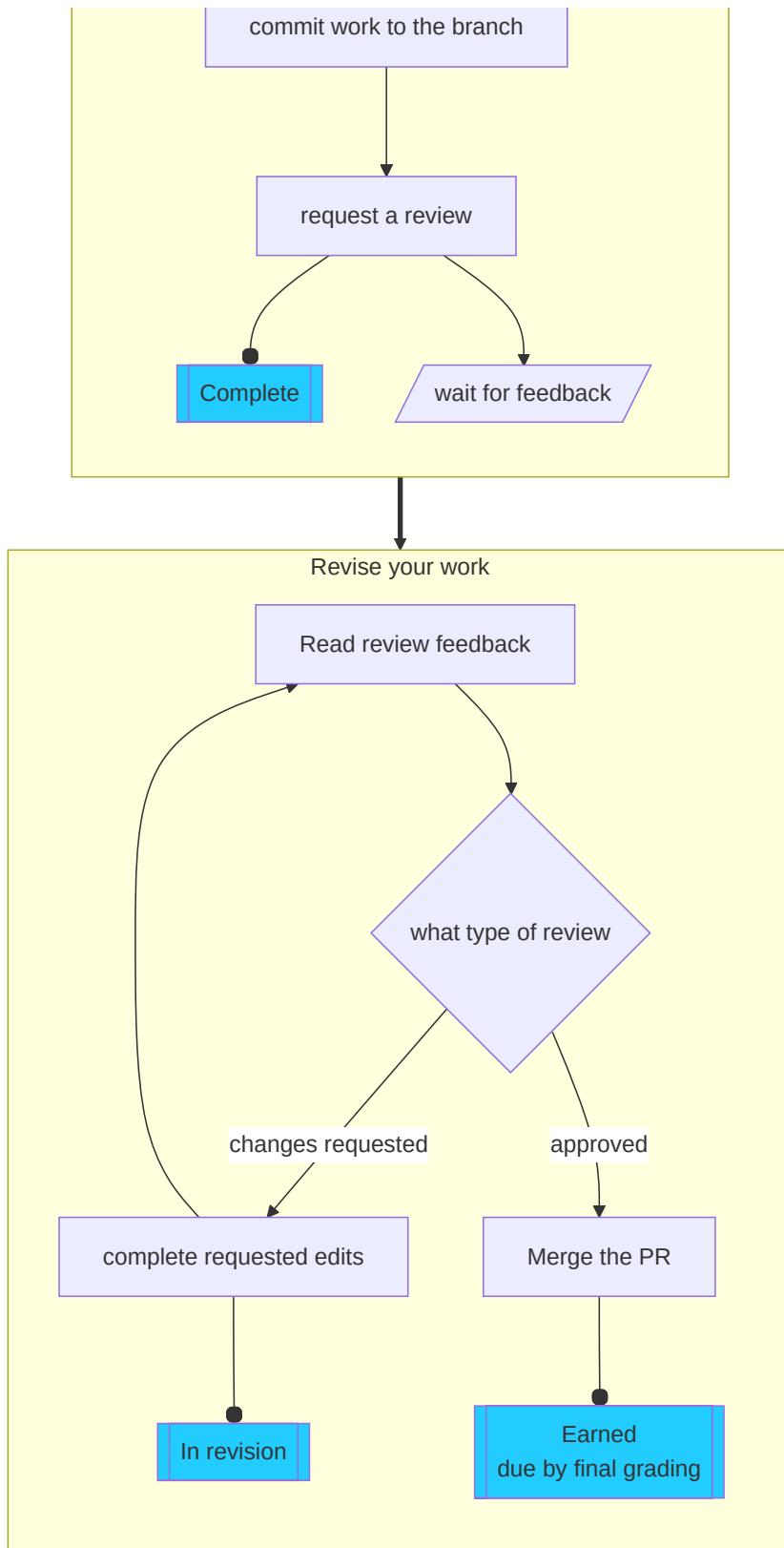




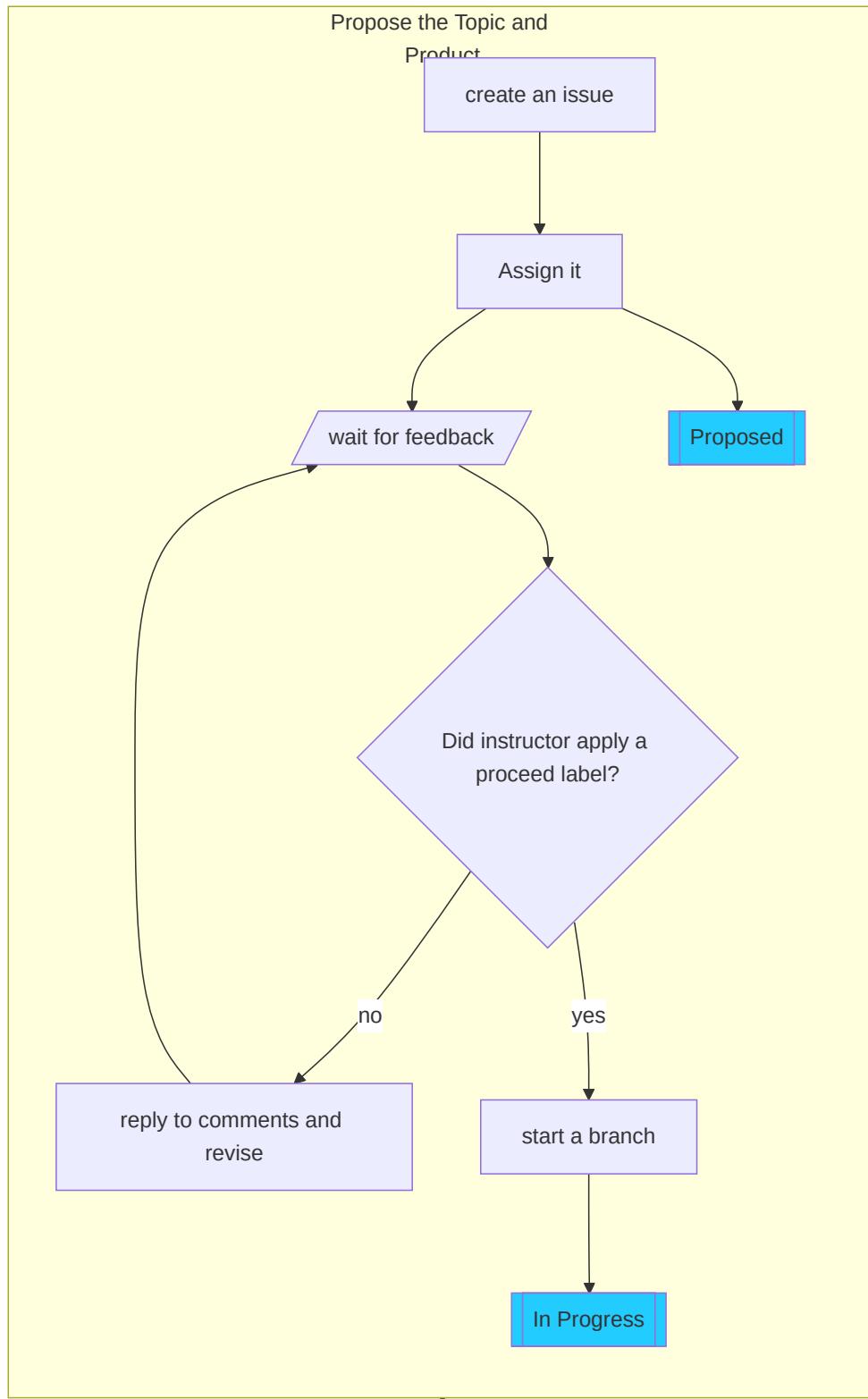
Earned  
due by final grading

## Explore Badges



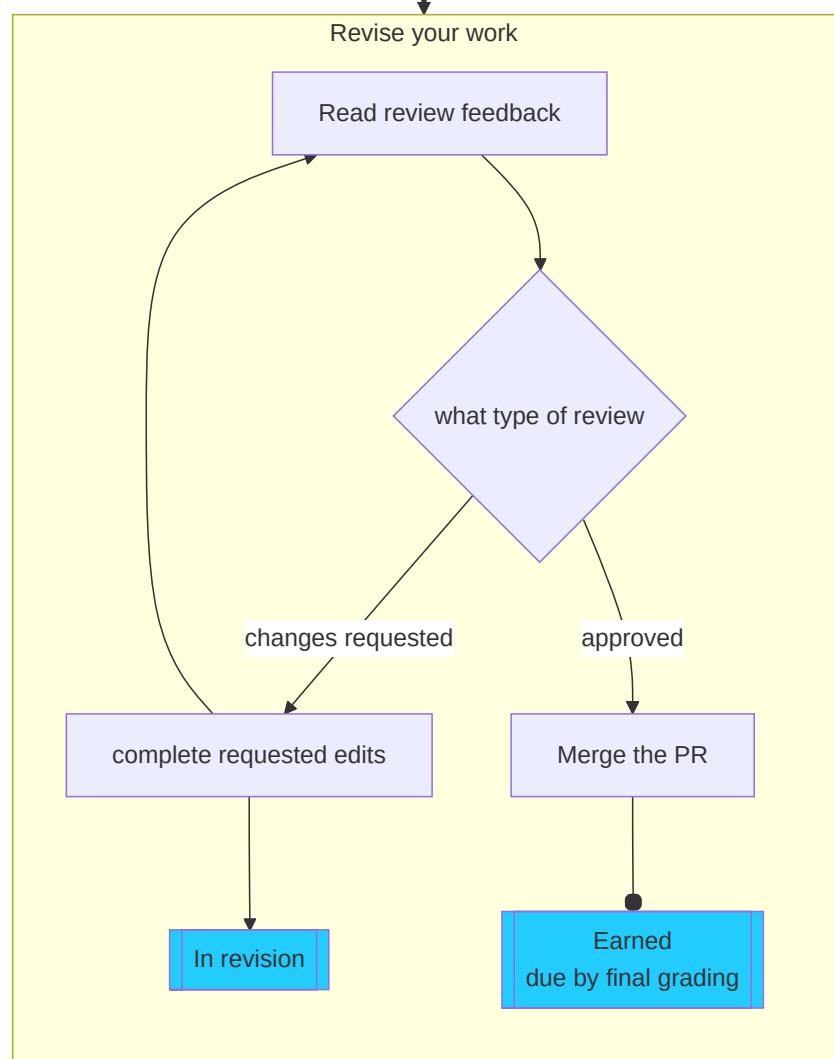
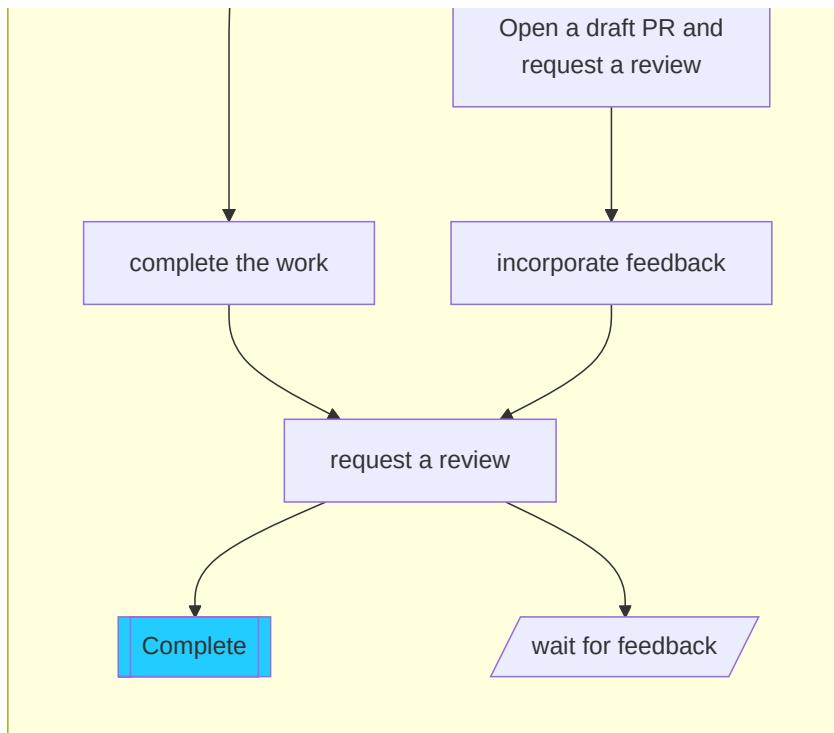


## Build Badges



Work on the badge

commit work to the branch



# Community Badges

You can log them either manually via files or with help of an action that a past student contributed!

## Logger Action

Your KWL repo has an action called “Community & Explore Badge Logger” that will help you

## Manual logging

These are the instructions from your `community_contributions.md` file in your KWL repo: For each one:

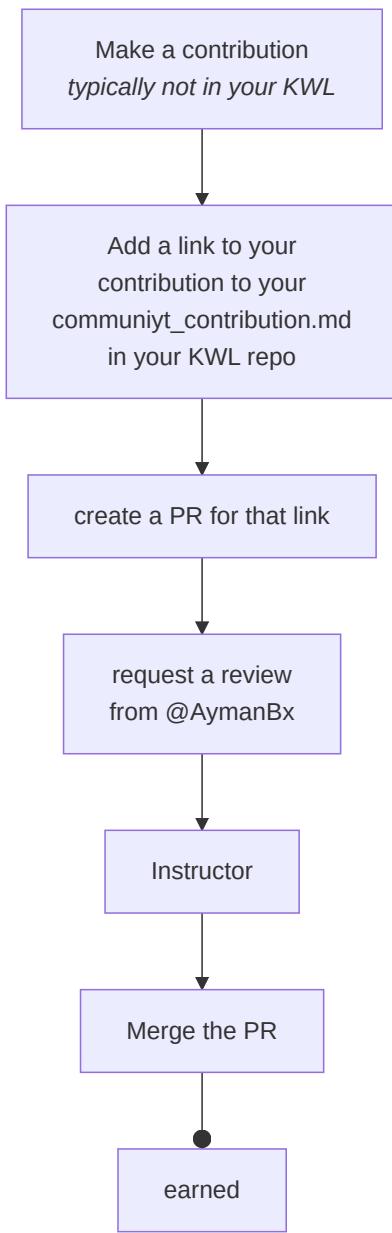
- In the `community\_contributions.md` file on your kwl repo, add an item in a bulleted list (start the line with - )
- Include a link to your contribution like `[text to display](url/of/contribution)`
- create an individual [pull request](#) titled “Community-shortname” where `shortname` is a short name for what you did. approval on this PR by instructor will constitute credit for your grade
- request a review on that PR from @AymanBx

### ⚠️ Important

You want one contribution per [PR](#) for tracking

### Note

You, to  
helps a  
organiz



## Detailed Grade Calculations

### Important

This page is generated with code and calculations, you can view them for more precise implementations of what the english sentences mean.

### Warning

These calculations may change a little bit and this page will be updated.

What is on the [Grading](#) page will hold true, but the detailed calculation here will update a little bit in ways that provide some more flexibility.

► Show code cell source

Grade cutoffs for total influence are:

► Show code cell source

letter	threshold
F	0
D	106
D+	124
C-	142
C	192
C+	210
B-	228
B	246
B+	264
A-	282
A	300

The total influence of each badge is as follows:

► Show code cell source

	badge	complexity	badge_type
0	experience	2	learning
1	lab	2	learning
2	review	3	learning
3	practice	6	learning
4	explore	9	learning
5	build	36	learning

## Bonuses

In addition to the weights for each badge, there also bonuses that will automatically applied to your grade at the end of the semester. These are for longer term patterns, not specific assignments. You earn these while working on other assignments, not separately.

### Important

the grade plans on the grading page and the thresholds above assume you earn the Participation and Lab bonuses for all grades a D or above and the Breadth bonus for all grades above a C.

Name	Definition	Influence	type
Participation	22 experience badges	18	auto
Lab	12 lab checkouts	18	auto
Breadth	If review + practice badges $\geq 18$ :	32	auto
Git-ing unstuck	fix large mistakes your repo using advanced git operations and submit a short reflection (allowable twice; instructor must approve)	9	event
Early bird	(review + practice) submitted by 02/21 $\geq 5$	9	event
Descriptive commits	all commits in KWL repo and build repos after penalty free zone have descriptive commit messages (not GitHub default or nonsense)	9	event
Curiosity	at least 15 experience reports have questions on time (before notes posted in evenings; instructor will log & award)	9	event
Community Star	10 community badges	18	auto
Hack the course - Contributor - Build	1 build that contributes to the course infrastructure/website +1 community or review	18	event
Hack the course - Contributor - Explore	1 explore that contributes to the course infrastructure/website + 2 community, with at least 1 review	18	event
Hack the course - Critic	5 total community badge, at least 2 reviews of other course contributions	9	event

Auto bonuses will be calculated from your other list of badges. Event bonuses will be logged in your KWL repo, where you get instructions when you meet the criteria.

#### Note

These bonuses are not pro-rated, you must fulfill the whole requirement to get the bonus. Except where noted, each bonus may only be earned once

#### Note

You cannot guarantee you will earn the Git-ing unstuck bonus, if you want to intentionally explore advanced operations, you can propose an explore badge, which is also worth 9.

## Bonus Implications

Attendance and participation is very important:

- 14 experience, 6 labs, and 9 practice is an F
- 22 experience, 13 labs, and 9 practice is a C-
- 14 experience, 6 labs, 9 practice and one build is a C-
- 22 experience, 13 labs, 9 practice and one build is a C+

Missing one thing can have a nonlinear effect on your grade. Example 1:

- 22 experience, 13 labs, and 18 review is a C
- 21 experience, 13 labs, and 18 review is a C-
- 21 experience, 13 labs, and 17 review is a D+
- 21 experience, 12 labs, and 17 review is a D

Example 2:

- 22 experience, 13 labs, and 17 practice is a C
- 22 experience, 13 labs, 17 practice, and 1 review is a B-
- 22 experience, 13 labs, and 18 practice is a B

The Early Bird and Descriptive Commits bonuses are straight forward and set you up for success. Combined, they are also the same amount as the participation and lab bonuses, so getting a strong start and being detail oriented all semester can give you flexibility on attendance or labs.

Early Bird, Descriptive commits, Community Star, and Git-ing Unstuck are all equal to the half difference between steps at a C or above. So earning any two can add a + to a C or a B for example:

- 22 experience, 13 labs, 18 practice, Descriptive Commits, and Early Bird is a B+
- 22 experience, 13 labs, 18 review, Descriptive Commits, and Early Bird is a C+

in these two examples, doing the work at the start of the semester on time and being attentive throughout increases the grade without any extra work!

If you are missing learning badges required to get to a bonus, community badges will fill in for those first. If you earn the Participation, Lab, and Breadth bonuses, then remaining community badges will count toward the community bonus.

For example, at the end of the semester, you might be able to skip some the low complexity learning badges (experience, review, practice) and focus on your high complexity ones to ensure you get an A.

The order of application for community badges:

- to make up missing experience badges
- to make up for missing review or practice badges to earn the breadth bonus
- to upgrade review to practice to meet a threshold
- toward the community badge bonus

To calculate your final grade at the end of the semester, a script will count your badges and logged event bonuses. The script can output as a yaml file, which is like a dictionary, for an example here we will use a dictionary.

see [cspt docs](#) for CLI version

```
example_student = {'experience': 22, 'lab': 13, 'review': 0, 'practice': 18,
                   'explore': 3,
                   'build': 0,
                   'community': 0,
                   'hack': 0,
                   'unstuck': 0,
                   'descriptive': 1,
                   'early': 1,
                   'question': 10}
```

```
badges_comm_applied = grade_calculation.community_apply(example_student)
badges_comm_applied
```

```
{'experience': 22,
 'lab': 13,
 'review': 0,
 'practice': 18,
 'explore': 3,
 'build': 0,
 'community': 0,
 'hack': 0,
 'unstuck': 0,
 'descriptive': 1,
 'early': 1,
 'question': 10}
```

```
grade_calculation.calculate_grade(badges_comm_applied)
```

'A'

```
grade_calculation.calculate_grade(badges_comm_applied, True)
```

300

## Schedule

### Overview

The following is a tentative outline of topics in an order, these things will be filled into the concrete schedule above as we go. These are, in most cases bigger questions than we can tackle in one class, but will give the general idea of how the class will go.

### How does this class work?

~ one week

We will start by introducing some basics of GitHub and setting expectations for how the course will work. This will include how you are expected to learn in this class which requires a bit about how knowledge production in computer science works and

getting started with the programming tools.

## What tools do Computer Scientists use?

Next we'll focus in on tools we use as computer scientists to do our work. We will use this as a way to motivate how different aspects of a computer work in greater detail. While studying the tools and how they work, we will get to see how some common abstractions are re-used throughout the fields and it gives a window and good motivation to begin considering how the computer actually works.

Topics:

- bash
- linux
- git
- i/o
- ssh and ssh keys
- number systems
- file systems

## What Happens When I run code?

Finally, we'll go in really deep on the compilation and running of code. In this part, we will work from the compilation through to assembly down to hardware and then into machine representation of data.

Topics:

- software system and Abstraction
- programming languages
- cache and memory
- compilation
- linking
- basic hardware components

## Recommended workload distribution

### Note

General badge deadlines are on the [detailed badge procedures](#) page.

To plan your time, I recommend expecting the following:

- 30 minutes, twice per week for prepare work (typically not this much).
- 1.5(review)-3(practice) hours, twice per week for the dated badges (including revisions).

For each explore :

- 30 min for proposal
- 7 hours for the project

For each build:

- 1.5 hour for the proposal (including revisions)
- 22 hours for the project
- 30 min for the final reflection

This is a four credit course, meaning we have approximately 4 hours of class + lab time per week( $75 \times 2 + 105 = 255$  minutes or 4.25 hours). By the [accreditation standards](#), students should spend a minimum of 2 hours per credit of work outside of class over 14 weeks. For a 4 credit class, then, the expected minimum number of hours of work outside of class you should be spending is 112 hours( $2 * 4 * 14$ ). With these calculations, given that there are 26 class sessions and only 18 review or practice are required, it is possible to earn an A with approximately 112 hours of work outside of class and lab time.

### Note

the first  
probab  
becaus  
revision  
likely ta

You do  
and ex  
details.

## Tentative Timeline

### ⚠ Warning

This section is not yet updated for Spring 2025.

This is a rough example.

This is the planned schedule, but is subject to change in order to adapt to how things go in class or additional questions that come up.

```
import pandas as pd
pd.read_csv('schedule.csv', index_col='date').sort_index()
```

	question	keyword	conceptual	practical	social	activity
date						
2025-	Welcome, Introduction, and Setup	intro	what is a system, why study tools	GitHub basics	class intros	create kwl repo in github, navigate github.com...
2025-	Course Logistics and Learning	logistics	github flow with issues	syllabus	working together and building common vocab	set up to work offline together, create a folder
2025-	Bash intro & git offline	terminal start	git structure, paths and file system	bash path navigation, git terminal authentication	why developers work differently than casual users	navigate files and clone a repo locally
2025-	How can I work with branches offline?	gitoffline	git branches	github flow offline, resolving merge conflicts	communication is important, git can help fix mi...	clone a repo and make a branch locally
2025-	When do I get an advantage from git and bash?	why terminal	computing mental model, paths and file structure	bash navigation, tab completion	collaboration requires shared language, shared...	work with bash and recover from a mistake with...
2025-	What *is* a commit?	merge conflicts	versions, git vlaues	merge conflicts in github, merge conflicts wit...	human and machine readable, commit messages ar...	examine commit objects, introduce plumbing com...
2025-	How do programmers communicate about code?	documentation	build, automation, modularity, pattern matching,	generate documentation with jupyterbook, gitig...	main vs master, documentation community	make a jupyterbook
2025-	What *is* git?	git structure	what is a file system, how does git keep track...	find in bash, seeing git config, plumbing/porc...	git workflows are conventions, git can be used...	examine git from multiple definitions and insp...
2025-	Why are these tools like this?	unix philosophy	unix philosophy, debugging strategies	decision making for branches	social advantages of shared mental model, diff...	discussion with minor code examples
2025-	How does git make a commit?	git internals	pointers, design and abstraction, intermediate...	inspecting git objects, when hashes are unique...	conventions vs requirements	create a commit using plumbing commands
2025-	How can I release and share my code?	git references	pointers, git branches and tags	git branches, advanced fixing, semver and conv...	advantages of data that is both human and mach...	make a tag and release
2025-	What is a commit number?	numbers	hashes, number systems	git commit numbers, manual hashing with git	number systems are derived in culture	discussion and use hashing algorithm
2025-	How can I automate things with bash?	bash scripting	bash is a programming language, official docs,...	script files, man pages, bash variables, bash ...	using automation to make collaboration easier	build a bash script that calculates a grade
2025-	How can I work on a remote	server	server, hpc, large files	ssh, large files, bash head, grep,	hidden impacts of remote	log into a remote server and work with

	question	keyword	conceptual	practical	social	activity
date						
2025-	server?			etc	computation	large f...
2025-	What is an IDE?	IDE	IDE parts	compare and contrast IDEs	collaboration features, developer communities	discussions and sharing IDE tips
2025-	How do I choose a Programming Language for a p...	programming languages	types of PLs, what is PL studying	choosing a language for a project	usability depends on prior experience	discussion or independent research
2025-	How can I authenticate more securely from a te...	server use	ssh keys, hpc system structure	ssh keys, interactive, slurm	social aspects of passwords and security	configure and use ssh keys on a hpc
2025-	What Happens when we build code?	building	building C code	ssh keys, gcc compiler	file extensions are for people, when vocabulary...	build code in C and examine intermediate outputs
2025-	What happens when we run code?	hardware	von neuman architecture	reading a basic assembly language	historical context of computer architectures	use a hardware simulator to see step by step o...
2025-	How does a computer represent non integer quan...	floats	float representation	floats do not equal themselves	social processes around standard developments, ...	work with float representation through fractio...
2025-	How can we use logical operations?	bitwise operation	what is a bit, what is a register, how to break...	how an ALU works	tech interviews look for obscure details somet...	derive addition from basic logic operations
2025-	What *is* a computer?	architecture	physical gates, history	interpreting specs	social context influences technology	discussion
2025-	How does timing work in a computer?	timing	timing, control unit, threading	threaded program with a race condition	different times matter in different cases	write a threaded program and fix a race condition
2025-	How do different types of storage work together?	memory	different type of memory, different abstractions	working with large data	privacy/respect for data	large data that has to be read in batches
2025-	How does this all work together	review	all	end of semester logistics	group work final	review quiz, integration/reflection questions
2025-	How did this semester go?	feedback	all	grading	how to learn better together	discussion

## Tentative Lab schedule

```
pd.read_csv('labschedule.csv', index_col='date').sort_index()
```

	topic	activity
date		
2025-01-27	unix philosophy	design a command line tool that would enable a...
2025-02-03	offline branches	plan for success, clean a messy repo
2025-02-10	git plumbing	git plumbing experiment
2025-02-19	scripting	releases and packaging
2025-02-24	GitHub Basics	syllabus quiz, setup
2025-03-03	os	hardware simulation
2025-03-10	tool familiarity	work on badges, self progress report
2025-03-17	remote, hpc	server work, batch scripts
2025-03-24	Machine representation	bits and floats and number libraries
2025-03-31	Compiling	C compiling experiments
2025-04-07	git plumbing	grade calculation script, self reflection
2025-04-14	working at the terminal	organization, setup kwl locally, manage issues
2025-04-21	hardware	self-reflection, work, project consultations

## Support Systems

### Mental Health and Wellness:

We understand that college comes with challenges and stress associated with your courses, job/family responsibilities and personal life. URI offers students a range of services to support your [mental health and wellbeing](#), including the [URI Counseling Center](#), [TELUS Health Student Support App](#), the [Wellness Resource Center](#), the [Psychological Consultation Center](#), the [URI Couple and Family Therapy Clinic](#), and [Well-being Coaching](#).

### Academic Enhancement Center

**Academic Enhancement Center** (for undergraduate courses): All Academic Enhancement Center support services for Spring 2025 begin on January 27th and are offered at no added cost to undergraduate students. Visit [AEC](#) for more information about our programs described below. Appointments can be scheduled in TracCloud located in [Microsoft 365](#).

- **STEM Tutoring:** Get peer tutoring for many 100 and 200 level STEM, Business, Nursing, and Engineering courses. Choose weekly or occasional sessions through TracCloud or visit the Drop-In Center in Carothers Library LL004. For more details visit [STEM & BUS Tutoring](#).
- **Academic Skills Development:** Meet one-on-one with a peer academic coach to build habits and strategies around time management, goal setting, and studying. Contact [Heather Price](#) for more information. [Click here](#) for more details. UCS 160 and UCS 161 are 1 credit courses designed to improve your academic skills and strategies. Consider enrolling in one of these courses! Contact [David Hayes](#) with any questions or to schedule a professional staff academic consultation. [Click here](#) for more details.

- The **Undergraduate Writing Center**: Receive peer writing support at any stage of your writing process. Schedule in-person or online consultations through TracCloud or stop by Roosevelt Hall Room 20 -new location! [Click here](#) for more details.

## General Policies

### Anti-Bias Statement:

We respect the rights and dignity of each individual and group. We reject prejudice and intolerance, and we work to understand differences. We believe that equity and inclusion are critical components for campus community members to thrive. If you are a target or a witness of a bias incident, you are encouraged to submit a report to the [URI Bias Resource Team](#). There you will also find people and resources to help.

### Disability, Access, and Inclusion Services for Students Statement

This course is specifically designed to use universal design principles. Many of the standard accommodations that the DAI office provides will not apply to this course, because of how it is designed: there are no exams for you to get extra time on, and no slides for you to get in advance. However, I am happy to work with you to help you understand how to use the build in support systems for the course.

URI wide:

Your access in this course is important. Please send me your Disability, Access, and Inclusion (DAI) accommodation letter early in the semester so that we have adequate time to discuss and arrange your approved academic accommodations. If you have not yet established services through DAI, please contact them to engage in a confidential conversation about the process for requesting reasonable accommodations in the classroom. DAI can be reached by calling: 401-874-2098, visiting: [web.uri.edu/disability](http://web.uri.edu/disability), or emailing: [dai@etal.uri.edu](mailto:dai@etal.uri.edu).

### Academic Honesty

Students are expected to be honest in all academic work. A student's name on any written work, quiz or exam shall be regarded as assurance that the work is the result of the student's own independent thought and study. Work should be stated in the student's own words, properly attributed to its source. Students have an obligation to know how to quote, paraphrase, summarize, cite and reference the work of others with integrity. The following are examples of academic dishonesty:

- Using material, directly or paraphrasing, from published sources (print or electronic) without appropriate citation
- Claiming disproportionate credit for work not done independently
- Unauthorized possession or access to exams
- Unauthorized communication during exams
- Unauthorized use of another's work or preparing work for another student
- Taking an exam for another student
- Altering or attempting to alter grades
- Fabricating or falsifying facts, data or references

- Facilitating or aiding another's academic dishonesty
- Submitting the same work for more than one course without prior approval from the instructors

### Tip

Most assignments are tested against LLMs and designed so that outsourcing it to an LLM will likely lead to a submission that is below the bar of credit.

## AI Use

All of your work must reflect your own thinking and understanding. The written work in English that you submit for review and practice badges must be your own work or content that was provided to you in class, it cannot include text that was generated by an AI or plagiarized in any other way. You may use auto-complete in all tools including, IDE-[integrated development environment GitHub](#) co-pilot (or similar, IDE embedded tool) for any code that is required for this course because the code is necessary to demonstrate examples, but language syntax is not the core learning outcome.

### Important

It is not okay to copy-paste and submit anything from an LLM chatbot interface in this course

If you are found to submit prismia responses that do not reflect your own thinking or that of discussion with peers as directed, the experience badge for that class session will be ineligible.

If work is suspected to be the result of inappropriate collaboration or AI use, you will be allowed to take an oral exam in lab time to contest and prove that your work reflects your own understanding.

The first time you will be allowed to appeal through an oral exam. If your appeal is successful, your counter resets. If you are found to have violated the policy then the badge in question will be ineligible and your maximum number of badges possible to be earned will be limited according to the guidelines below per badge type (you cannot treat the plagiarized badge as skipped). If you are found to have violated the policy a second time, then no further work will be graded for the remainder of the semester.

If you are found to submit work that is not your own for a *review or practice* badge, the review and practice badges for that date will be ineligible and the penalty free zone terms will no longer apply to the first six badges.

If you are found to submit work that is not your own for an *explore or build* badge, that badge will not be awarded and your maximum badges at the level possible will drop by 1/3 of the maximum possible (2 explore or 1 build) for each infraction.

## Attendance

"Attendance" is not explicitly checked, but participation in class through prismia is monitored, and lab checkouts and experience badges grade your engagement in the activities of lab and class respectively.

# Viral Illness Precautions Statement

The University is committed to delivering its educational mission while protecting the health and safety of our community. Students who are experiencing symptoms of viral illness should NOT go to class/work. The [Centers for Disease Control and Prevention \(CDC\)](#) recommends that all people who are experiencing viral illness should stay home and away from others until symptoms improve and they are fever free (without medications) for 24 hours. They should take added precautions for the next 5 days.

If you miss class once, you **do not need to notify me** in advance. You can follow the [makeup procedures](#) on your own.

## Excused Absences

Absences due to serious illness or traumatic loss, religious observances, military service, or participation in a university sanctioned event are considered excused absences.

**You do not need to notify me in advance.**

For *short absences* (1-2 classes), for any reason, you can follow the [makeup procedures](#), no extensions will be provided typically for this; if extenuating circumstances arise, then ask Any instructor.

For *extended excused absences*, (3 or more classes) email Ayman when you are ready to get caught up and she will help you make a plan for the best order to complete missed work so that you are able to participate in subsequent activities. Extensions on badges will be provided if needed for excused absences. In your plan, include what class sessions you missed by date.

For unexcused absences, the makeup procedures apply, but not the planning assistance via email, only regularly scheduled office hours, unless you have class during all of those hours and then you will be allowed to use a special appointment.

## Office Hours & Communication

### Announcements

Announcements will be made via [GitHub](#) Release. You can view them [online in the releases page](#) or you can get notifications by watching the [repository](#), choosing “Releases” under custom [see GitHub docs for instructions with screenshots](#). You can choose [GitHub](#) only or e-mail notification [from the notification settings page](#)

#### Warning

For the first week announcements will be made by BrightSpace too, but after that, all course activities will be only on GitHub.

## Sign up to watch

^

Watch the repo and then, after the first class, [claim a community badge](#) for doing so, using a link to these instructions as the “contribution” like follows.

- [watched the repo as per announcements](<https://compsys-progtools.github.io/spring2025/syllabus>)

put this on a [branch](#) called [watch\\_community\\_badge](#) and title your PR “Community-Watch”

## Help Hours

Day	Time	Location	Host
Monday	9am-12pm	Zoom	Ayman Sandouk
Tuesday	2:30pm-4:30pm	Tyler - Rm 139	Elijah Smith-Antonides
Wednesday	10am-12pm	Tyler - Rm 139	Trevor Moy

Online office hours locations and appointment links for alternative times are linked on the [GitHub Organization Page](#)

### Important

You can only see them if you are a “member”. To join, make sure that you have completed Lab 0.

## Tips

### TLDR

Contribute a TLDR set of tabs or mermaid visual to this section for a community badge.

## For assignment help

- use the badge issue for comments and @ mention instructors
- **send in advance, leave time for a response**
- **always** use issues in your repo for content directly related to assignments. If you [push \(changes to a repository\)](#) your partial work to the [repository](#) and then open an [issue](#), we can see your work and your question at the same time and download it to run it if we need to debug something
- use issues or discussions for questions about this syllabus or class notes. At the top right there's a [GitHub](#) logo ⓘ that allows you to open a [issue](#) (for a question) or suggest an edit (eg if you think there's a typo or you find an additional helpful resource related to something)

### Note

I check e-mail/github a small number of times per day, during work hours, almost exclusively. You might see me post to this site, post to BrightSpace, or comment on your assignments outside of my normal working hours, but I will not reliably see emails that arrive during those hours. This means that it is important to start assignments early.

### Should you e-mail your work?

No, request a [pull request](#) review or make an [issue](#) if you are stuck

# 1. Welcome, Introduction, and Setup

Today:

- intros
- what the *learning* goals of the course are
- see how in class time will work
- start learning git/github by doing

Not Today:

- syllabus review (on your own time/lab Monday)
- cours policy discussion (next week)

## 1.1. Introductions

- Ayman Sandouk
- Trevor Moy
- Elijah Smith-Antonides

## 1.2. Why think like a computer?

With Large Language Models (LLMs) able to write code from English (or other spoken languages, but LLMs are generally worse at non English)

Let's discuss some examples.

Many things in this course *are* things you will use **everyday** some of it is stuff that will help you in the trickiest times.

What differentiates LLM (Large Language Modules) from Computer Scientists?

## 1.3. GitHub

- This class is not a GitHub class
- GitHub is the main tool that we will be using in this course

- Not expecting you to be familiar with it
- Homework is submitted through GitHub in a non-traditional way
- Great practice for real-life software development with a team or even individual work
- More on that later

I look forward to getting to know you all better.

## 1.4. Prismia

- instead of slides
- you can message us
- we can see all of your responses
- emoji!

questions can be “graded”

- this is instant feedback
- participation will be checked
- correctness will not impact your final grade (directly)
- this helps both me and you know how you are doing

Questions can be multi-choice

or open ended

And I can share responses, grouped up

## 1.5. This course will be different

- no Brightspace
- 300 level = more independence
- I will give advice, but only hold you accountable to a minimal set
- High expectations, with a lot of flexibility

as an aside [another Professor describing](#) what she does not like about learning management systems (LMS).  
Brightspace is one, she talks about Canvas in the post, but they are similar.

I will not chase you.

I do not judge your reasons for missing class.

- **No need to tell me in advance**
- For 1 class no need to tell me why at all
- For 1 class, make it up and keep moving
- For longer absences, I will help you plan how to get caught up, and you must meet university criteria for excused absence

### 1.5.1. My focus is for you to learn

- that means, practice, feedback, and reflection
- you should know that you have learned
- you should be able to apply this material in other courses

### 1.5.2. Learning comes in many forms

- different types of material are best remembered in different ways
- some things are hard to explain, but watching it is very concrete

## 1.6. Learning is the goal

- producing outputs as fast as possible is not learning
- in a job, you may get paid to do things fast
- your work also needs to be correct, without someone telling you it is
- in a job you are trusted to know your work is correct, your boss does not check your work or grade you
- to get a job, you have to interview, which means explaining, in words, to another person how to do something

## 1.7. What about AI?

Large Language Models will change what programming looks like, but understanding is always going to be more effective than asking an AI. Large language models actually do not know anything, they just know what languages look like and generate text.

*if you cannot tell it when it's wrong, you do not add value for a company, so why would they pay you?*

## 1.8. This is a college course

- more than getting you one job, a bootcamp gets you one job
- build a long (or maybe short, but fruitful) career
- build critical thinking skill that makes you adaptable
- have options

## 1.9. “I never use what I learned in college”

- very common saying
- it's actually a sign of deep learning
- when we have expertise, we do not even notice when we apply it
- college is not about the facts, but the processes

## 1.10. Learning is hard

## 1.11. How does this work?

### 1.11.1. In class:

1. Memory/ understanding check
2. Review/ clarification as needed
3. New topic demo with follow along, tiny practice
4. Review, submit questions

### 1.11.2. Outside of class:

1. Read notes Notes to refresh the material, check your understanding, and find more details
2. Practice material that has been taught
3. Activate your memory of related things to what we will cover
4. Read articles/ watch videos to either fill in gaps or learn more details
5. Bring questions to class

## 1.12. Getting started

Your KWL chart is where you will start by tracking what you know now/before we start and what you want to learn about each topic. Then you will update it throughout the semester. You will also add material to the repository to produce evidence of your learning.

[Accept the assignment](#) to create your repository (aka repo)

We have a glossary!!

[repository](#)

### 1.12.1. What is a directory?

A “directory” in computing is a file system structure that acts like a container, organizing and managing files and other directories (often called folders) on a computer

**pro tip: links are often hints or more information**

Commits represent a message to your future self/teammates/viewers.  
They mark what significant change has been made between one "checkpoint" in the status of a project and t

## 1.13. What is this course about?

In your KWL chart, there are a lot of different topics that are not obviously related, so what is this course really about?

- practical exposure to important tools
- design features of those tool categories
- basic knowledge of many parts of the CS core
- focus on the connections

We will use learning the tools to understand how computer scientists think and work.

Then we will use the tools to examine the field of Computer Science top to bottom (possibly out of order).

### 1.13.1. How it fits into your CS degree

knowing where you've been and where we're going will help you understand and remember

## 1.14. In your degree

*this describes the BS; BA drops some of the math*

In CSC110, you learn to program in python and see algorithms from a variety of domain areas where computer science is applied.

Then in CSC 340 and 440 you study the algorithms more mathematically, their complexity, etc.

In CSC211, 212, you learn the foundations of computer science: general programming and data structures.

Then in 301, 305, 411, 412 you study different aspects of software design and how computers work.

In this class, we're going to connect different ideas. We are going to learn the tools used by computer scientists, deeply. You will understand why the tools are the way they are and how to use them even when things go wrong.

## 1.15. GitHub Docs are really helpful and have screenshots

- [editing a file](#)
- [pull request](#)

they pay people to update them so I direct you to theirs mostly instead of recreating them

Today we did the following:

1. Accept the assignment to create your repo: [KWL Chart](#)
2. Edit the README to add your name by clicking the pencil icon ([editing a file](#) step 2)
3. adding a descriptive commit message ([editing a file](#) step 5)
4. adding prior knowledge
5. created a new branch (named `prior_knowledge`) ([editing a file](#) step 7-8)
6. added a message to the Pull Request ([pull request](#) step 5)
7. Creating a pull request ([pull request](#) step 6)
8. Clicking Merge Pull Request

[join a team](#)

we will use this for discussions and see more in lab, but having you join now makes it easier for me to make you a member and give you access to a special member view

## 1.16. Git and GitHub terminology

We also discussed some of the terminology for git. We will also come back to these ideas in greater detail later.

### 1.16.1. GitHub Actions

GitHub allows us to run scripts within our repos, the feature is called GitHub Actions and the individual items are called workflows.

Navigate to your actions tab

### 1.16.2. Get Credit for Today's class

\*\*Run your Experience Reflection (inclass) action on your kwl repo \*\*

*talk with peers to make sure you remember what the right way to click on it is*

On the created PR, go to the `Files` section, edit the file, and commit the changes.

### 1.16.3. Fix your repo

So, it is apparently a weird bug in GitHub, that the actions were not working,

but it is easy (though a little annoying) to fix.

### 1.16.4. Make the edits

On each file in the .github/workflows folder that ends in .yml edit in some small way

#### Important

if the file has the word “reviewer” in, change the reviewer from “brownsarahm” to “AymanBx”

else: make any small change (eg add an additional blank line in a place where there is a blank line already).

### 1.16.5. Run forgotten badge action

1. Go to the actions tab of your repo
2. Select the action that has the name `Forgotten Badge (Late, but was in class)`
3. In the blue banner that appears click `Run Workflow`
4. leave the branch set to main
5. Enter the date `2025-01-23`

6. Wait a minute or so for it to run, when it has a green checkmark, go to your PR tab
7. select the PR with the title Experience Report 2025-01-23
8. Go to the files tab of that PR and edit it (use the 3 dots menu in the top right of the file box)
9. fill in the information and commit to the same branch (do not open an additional PR)
10. assign @instructors to review your PR.

For screenshots, see the [Manually running a workflow, on GitHub](#)

## 1.17. Prepare for next class

1. (for lab Monday) Read the syllabus section of the course website carefully and explore the whole course [website](#)
2. (for lab Monday) Bring questions about the course
3. (for class Tuesday) Think about one thing you've learned really well (computing or not). Be prepared to discuss the following: How do you know that you know it? What was it like to first learn it? (nothing written to submit, but you can use the issue to take notes if you would like)

## 1.18. Badges

[Review](#) [Practice](#)

1. [accept this assignment](#) and join the existing team to get access to more features in our course organization.
2. Post an introduction to your classmates [on our discussion forum](#) (include link to your comment in PR comment, must accept above to see)
3. Read the notes from today's class carefully
4. Fill in the first two columns of your KWL chart (content of the PR; named to match the badge name)

## 1.19. We have a Glossary!!

For example, the term we used above:

[repository](#)

 Tip

In class, on prismia, I will sometimes link like above, but you can also keep the page open if that is helpful for you.

In the course site, glossary terms will be linked as in the following list.

Key terms for the first class:

- [repository](#)
- [git](#)
- [github](#)
- [PR](#)

## 1.20. Questions after class

### 1.20.1. Should I be doing some practice/review badges between classes?

- Before each class, you need to complete the tasks assigned in the `prepare` issue
- At the end of each class, run the `experience report (in class)` action and fill out the file at your own time
- After each class, you get to pick one of the two (Practice/Review) and do that one. You only get graded for one of them

## 2. More orientation

Today we will:

- continue getting familiar with the structure of GitHub
- clarify more how the course will flow
- practice with new vocabulary

*Last class was a lot of new information, today we will reinforce that mostly, and add only a little*

### 2.1. Warm up

1. Navigate to your KWL repo
2. Find the issues tab
3. Open the prepare-2025-01-28 issue and discuss the questions with your classmates at your table
4. If you have issues that currently say Error: not found (prepare 01-30 and practice 01-28)

\*hint: my KWL repo URL is: <https://github.com/compsys-progtools/spring25-kwl-AymanBx>

What do you associate “learn new vocabulary” with?

Note: it is actually *always* the first step of learning, or joining a community.

What are GitHub issues for?

- [x] bug reporting and tracking
- [ ] proposing changes to the code by comparing two branches
- [ ] discussing things tangentially related to the code

What are GitHub issues for in our class?

- [ ] discussing things tangentially related to the code
- [ ] proposing changes to the code by comparing two branches
- [x] Assignment or issue that needs to be fixed in our repo/project

What are Pull Requests for?

- [ ] bug reporting and tracking
- [x] proposing changes to the code by comparing two branches

- [ ] discussing things tangentially related to the code

What are Pull Requests for in our class?

- [ ] bug reporting and tracking
- [x] Request an instructor to view my work and approve it or comment on it
- [ ] discussing things tangentially related to the code

Go to your PR tab

What is an experience badge?

- [ ] A way to prove I was in class (take attendance)
- [ ] A program that means at the end of the semester I get a medal for each one I have
- [x] A way to remind my future self and show my instructor what I've learned from class and ask questions

## 2.2. How do we work with experience badges

Checklist:

1. Merge prepare work into this PR
2. Link prepare issue to this PR
3. Complete experience report
4. Add activity completion evidence per notes

We fixed the `forgottenexperience.yml` file and then [ran it manually](#).

## 2.3. Making up for action issue last week

Yesterday we fixed the issue with our actions and were able to run the Forgotten badge action

What date did you make your experience badge for?

1. Date is supposed to be 2025-01-23
2. Redo and copy the content from the old one

Then we edited the file it created to add a title on the line with one `#` (should be line 10) using the 3 dots menu in the top right of the file on the `files changed` tab of the PR.

Which of the following is true? *hint: look at your experience badge from yesterday (and chat with neighbors)*

- [ ] once a PR is open you cannot add commits to either branch involved
- [ ] once a PR is open if you add commits to the proposing branch, you have to open a new PR
- [x] once a PR is open if you add commits to the proposing branch, they are visible in the existing PR

### 2.3.1. Remember

Your experience pull request (badge) already had a changed file in it And then you edited the file again to answer the existing prompts

#### Note

When you add more commits to a branch that has a PR, it automatically updates the PR.

### 2.3.2. Where does this file exist?

Find the message that says `github-actions wants to merge 1 commit into main from experience-<somenumber>`

### 2.3.3. What does this experience- represent?

This `is` a branch kf your repo that has one file that `is` new (different `from` the main branch where that fi  
This file `is` the experience report that you are asked to fill out after every lecture

### 2.3.4. How do I know the location of the file?

#### Note

Here we are learning *by example* and then *synthesizing* that into more concrete facts.

**my goal is to teach you to get better at learning in that way, bc it is what employers will expect**

To do this:

- I set up opportunities for you to *do* the things that give you the opportunity
- highlight important facts about what just happened
- ask you questions to examine what just happened

This is why attendance/participation is a big part of your grade.

Experience badges are evidence of having learned.

There is a time breakdown in the syllabus that suggests and recommends a good way to distribute your time in the semester for the class.

Take a minute to think about how you use your time and what that breakdown means for how you will plan.

Then we will use the tools to examine the field of Computer Science top to bottom (possibly out of order).

## 2.4. Programming is Collaborative

There are two very common types of collaboration

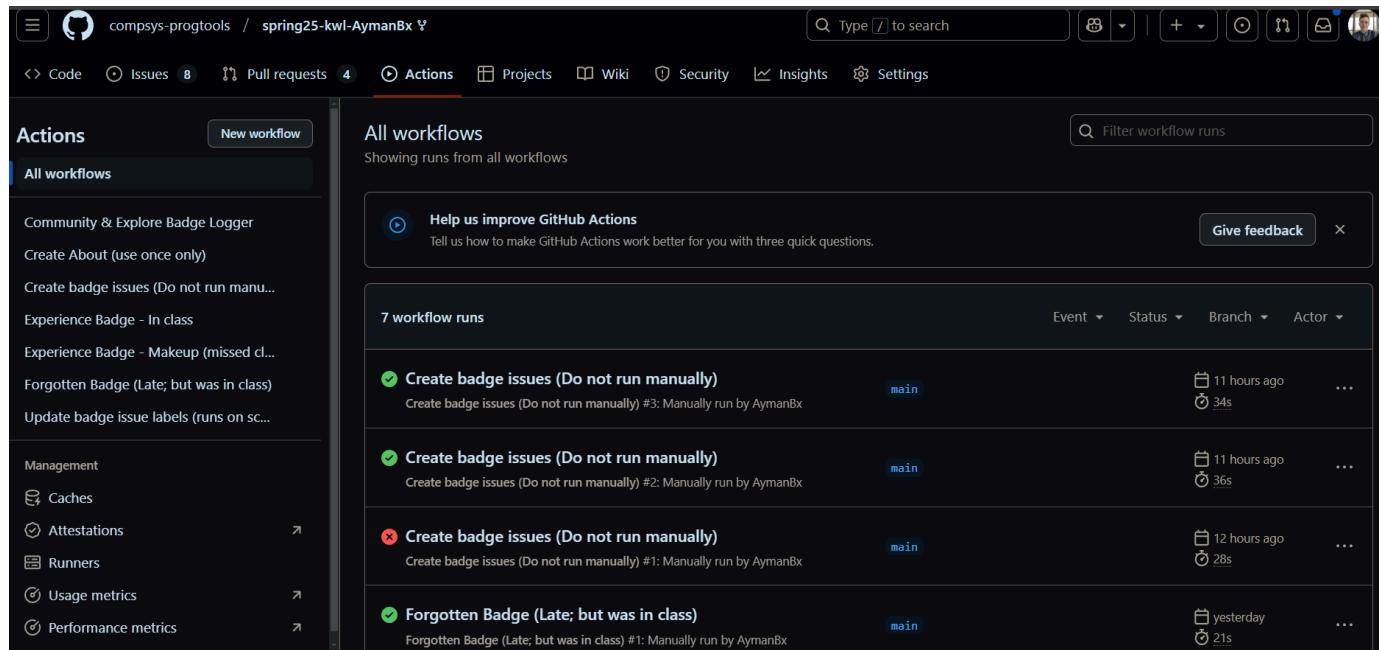
- code review (working independently and then reviewing)
- pair programming (sitting together and discussing while writing)

We are going to build your skill in the *code review* model. This means you need to collaborate, but collaboration in school tends to be more stressful than it needs to. If students have different goals or motivation levels it can create conflict. So there will be some chances for collaboration where people can show up at the level they want without impacting others.

You can also do build badges collaboratively, for a closer collaboration, but those are your choice.

## 2.5. GitHub Actions Tab

GitHub allows us to run scripts within our repos, the feature is called GitHub Actions and the individual items are called workflows.



The screenshot shows the GitHub Actions tab for a repository named 'compsys-progtools / spring25-kwl-AymanBx'. The left sidebar lists various GitHub features like Code, Issues, Pull requests, Actions, Projects, Wiki, Security, Insights, and Settings. The 'Actions' section is selected. The main area displays 'All workflows' with a search bar and a 'Help us improve GitHub Actions' card. Below is a table of '7 workflow runs' with columns for Event, Status, Branch, Actor, and three dots. The runs are:

Event	Status	Branch	Actor	...
Create badge issues (Do not run manually)	main	11 hours ago	34s	
Create badge issues (Do not run manually)	main	11 hours ago	36s	
Create badge issues (Do not run manually)	main	12 hours ago	28s	
Forgotten Badge (Late; but was in class)	main	yesterday	21s	
Update badge issue labels (runs on sc...)				
Management				
Caches				
Attestations				
Runners				
Usage metrics				
Performance metrics				

this should be different from yours, because I tested things in mine before making your PRs

On my actions page, in the screenshot above, how many **successful workflow runs** are shown?

- [ ] 4
- [ ] 5
- [x] 3

On my actions page, in the screenshot above, how many **total workflow runs** are shown?

- [x] 4
- [ ] 5
- [ ] 3

On my actions page, in the screenshot above, how many workflows are **available to run**?

- [ ] 4
- [x] 7

- [ ] 5

Your time to practice:

1. Navigate to your actions tab
2. Run the “Create About (use only once)” Action.

You should have:

- fixed action files
- Labs issues closed
- experience report for 2025-01-23

## 2.6. Prepare for next class

1. View all existing issues (Prepare, View)
2. Select one out of the two for each date and complete the tasks in it. (Pick whether you want to do practice or review for a certain date)
3. Make sure you ask for out review on any open pull request (other than feedback)
4. Choose where you want to save files for this class locally on your computer and make note of that location. (nothing to submit; but we will be working locally and you need to have a place)
5. Think about how you think about files and folders in a computer. What do you know about how they are organized? how they're implemented? (nothing to submit)

## 2.7. Badges

**Review**      **Practice**

*the text in ( ) below is why each step is assigned*

1. review today's notes after they are posted, both rendered and the raw markdown versions. Include links to both views in your badge PR comment. (to review)
2. [“Watch” the course website repo](#), specifically watch Releases under custom (to get notifications)
3. map out your computing knowledge and add it to your kwl chart repo. this can be an image that you upload or a text-based outline in a file called prior-knowledge-map. (optional) try mapping out using [mermaid](#) syntax, we'll be using other tools that will facilitate rendering later (what we will learn will connect a lot of ideas, mapping out where you start, sets you up for success)

## 2.8. Questions after class

# 3. Why Systems?

## 3.1. 5 minutes

## 3.2. Questions?

Issues, Pull requests, Prepare, Practice, Review, Lab...

## 3.3. Working offline

Today more clear motivation for each thing we do and more context.

Today we will learn to work with GitHub offline, this requires understanding some about file systems and how content is organized on computers.

We will learn:

- relative and absolute paths
- basic bash commands for navigating the file system
- authenticating to GitHub on a terminal
- how to clone a repo
- how fetch and checkout work

## 3.4. Let's get organized

For class you should have a folder on your computer where you will keep all of your materials.

We will start using the terminal today, by getting all set up.

Open a terminal window. I am going to use `bash` commands

- if you are on mac, your default shell is `zsh` which is mostly the same as bash for casual use. you can switch to bash to make your output more like mine using the command `bash` if you want, but it is not required.
- if you are on windows, your **GitBash** terminal will be the least setup work to use `bash` (preferred)
- if you have WSL (if you do not, no need to worry) you should be able to set your linux shell to `bash` (I believe it already is set to bash)

If you use `pwd` you can see your current path

```
pwd
```

```
Users/ayman
```

It outputs the absolute path of the location that I was at.

we start at home `~`

We can change directory with `cd`

if we use `cd` without a path, it goes back to home `cd ~` would do the same

We can make a new directory with `mkdir`

What you want to have is a folder for class (mine is systems) in a place you can find it

You might:

- make a systems folder in your Documents folder
- make an inclass folder in the CSC311 folder you already made
- use the CSC311 folder as your in class working space

If I run the following commands, what do I expect as the output?

```
cd  
pwd
```

- [ ] cannot tell, do not know where you started
- [x] /c/Users/ayman
- [ ] /c/Users/ayman/Documents
- [ ] the same as wherever you were before

When you use `pwd` what type of path does it return?

- [x] absolute
- [ ] relative

The first slash `/` represents the `/(root)` directory, which is the starting place for the search

But what does it mean when a path starts with `.` or `..`?

To go back one step in the path, (one level up in the tree) we use `cd ..`

```
cd ..
```

`..` is a special file that points to a specific relative path, of one level up. (In other words, parent folder/directory)

use `pwd`, `cd` and `pwd` to illustrate what “one level up” means

Did you notice the difference?

```
cds Documents/
```

```
bash: cds: command not found
```

notice that command not found is the error when there is a typo

```
cd Documents/
pwd
```

```
/c/Users/ayman/Documents
```

```
mkdir systems
```

```
/c/Users/ayman/Documents/systems
```

```
cd ..
pwd
```

```
/c/Users/ayman/Documents/
```

If we give no path to `cd` it brings us to home.

```
cd
pwd
```

```
/c/Users/ayman
```

Then we can go back.

```
cd Documents/systems/
pwd
```

```
/c/Users/ayman/Documents/systems
```

Do you have any content in the folder?

```
ls
```

Could be empty if you had created it before you would see the content you had in it

We can use two levels up at once like this:

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

```
/c/Users/ayman
```

```
cd Documents/systems/
```

We can the **tab** to complete once we have a unique set of characters. If what we have is not unique enough yet, bash will do nothing when you press tab once, but if you press it multiple times it will show you the options:

```
cd Do
```

Press **tab** twice. The consol outputs:

```
Documents/ Downloads/  
cd Do
```

```
cd Doc
```

Press **tab**... The consol auto completes the path

```
cd Documents/
```

What character is always at the start of absolute paths?

- [ ] :
- [x] /
- [ ] \*
- [ ] ]

What type of path is **..../Downloads** ?

- [ ] absolute
  - [x] relative
- 

## 3.5. A toy repo for in class

this repo will be for *in class* work, you will not get feedback inside of it, unless you ask, but you will answer questions in your kwl repo about what we do in this repo sometimes

only work in this repo during class time or making up class, unless specifically instructed to

Preferred:

1. [view the template](#)
  2. click the green “use this template” button in the top right
  3. make `compsys-progtools` the owner
  4. set the name to `gh-inclass-<your gh username>` replacing the `<>` part with your actual name
- 

Backup: [accept the assignment](#)

## 3.6. Authenticating with GitHub

We have two choices to Download a repository:

1. clone to maintain a link using git
2. download zip to not have to use git, but have no link

we want option 1 because we are learning git

For a public repo, it won't matter, you can use any way to download it that you would like, but for a private repo, we need to be authenticated.

### 3.6.1. Authenticating with GitHub

There are many ways to authenticate securely with GitHub and other git clients. We're going to use easier ones for today, but we'll come back to the third, which is a bit more secure and is a more general type of authentication.

1. ssh keys (we will do this later)
  2. `gh` CLI with `gh auth login`
- 

we will do option 2 for today

### 3.6.1.1. GitBash (windows mostly)

- `git clone` and paste your URL from GitHub
- then follow the prompts, choosing to authenticate in Browser.

### 3.6.1.2. Native terminal ( MacOS X/Linux/WSL)

- GitHub CLI: enter `gh auth login` and follow the prompts.
- then `git clone` and paste your URL from github

### 3.6.1.3. If nothing else works

Create a [personal access token](#). This is a special one time password that you can use like a password, but it is limited in scope and will expire (as long as you choose settings well).

Then proceed to the clone step. You may need to configure an identity later with [`git config`](#)

## 3.6.2. Cloning a repository

We will create a local copy by cloning

Type `pwd` first to make sure you're in the Systems folder

```
git clone https://github.com/compsys-progtools/gh-inclass-AymanBx
```

```
Cloning into 'gh-inclass-AymanBx'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 8 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (8/8), done.
```

Confirm it worked with:

```
ls
```

```
gh-inclass-AymanBx
```

We see the new folder that matches our repo name

## 3.7. What is in a repo?

We can enter that folder

```
cd gh-inclass-AymanBx/
```

When we compare the local directory to GitHub

```
ls
```

Notice that the `.github/workflows` that we see on GitHub is missing, that is because it is *hidden*. All file names that start with `.` are hidden.

We can actually see the rest of the files or folders with the `-a` for **all option** or **flag**. Options are how we can pass non required parameters to command line programs.

```
ls -a
```

```
..          .git  
..          .github
```

We also see some special “files”, `.` the current location and `..` up one directory

## 3.8. How do I know what git knows?

`git status` is your friend.

```
git status
```

```
On branch main  
Your branch is up to date with 'origin/main'.  
nothing to commit, working tree clean
```

this command compares your working directory (what you can see with `ls -a`) and all subfolders except the `.git` directory to the current state of your `.git` directory (more on that later ...).

## 3.9. Making a branch with GitHub and working offline

First on an issue, create a branch using the link in the development section of the right side panel. See the [github docs](#) for how to do that.

"create an about file"

Then it gives you two steps to do. We are going to do them one at a time so we can see better what they each do.

First we will update the `.git` directory without changing the working directory using [git fetch](#). We have to tell git fetch where to get the data from, we do that using a name of a [remote](#).

```
git fetch origin
```

```
From https://github.com/compsys-progtools/gh-inclass-AymanBx
 * [new branch]      1-create-an-about-file -> origin/1-create-an-about-file
```

We can look at the repo to see what has changed.

```
git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

This says nothing, because remember git status tells us the relationship between our working directory and the .git repo.

Next, we switch to that branch.

```
git checkout 1-create-an-about-file
```

```
branch '1-create-an-about-file' set up to track 'origin/1-create-an-about-file'.
Switched to a new branch '1-create-an-about-file'
```

and verify what happened

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.

nothing to commit, working tree clean
```

Run your Experience Badge (inclass) action. [Github how to](#)

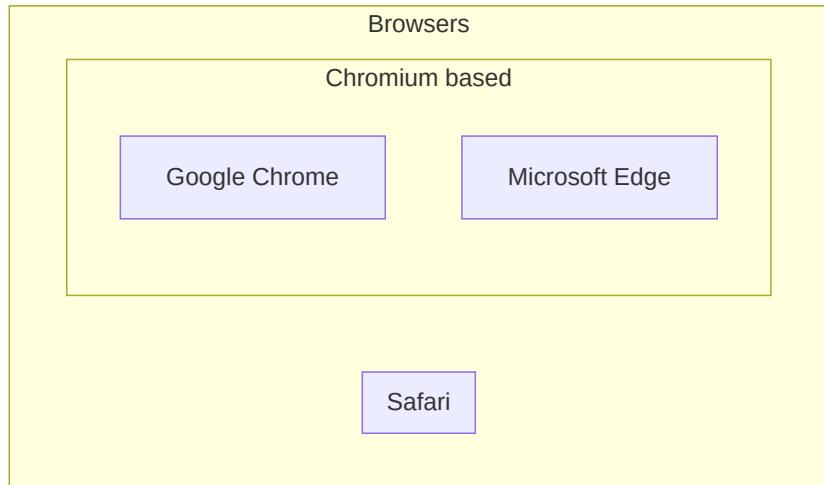
1. Go to the Actions tab
2. Click the Experience Badge (inclass) on the left hand side
3. Click the run workflow button on the right
4. Click the run workflow button in the popup

## 3.10. Prepare for next class

1. Find the glossary page for the course website, link it below. Review the terms for the next class: shell, terminal, bash, git, zsh, powershell, GitHub. Make a diagram using [mermaid](#) to highlight how these terms relate to one another. Put this in a file called `terminal-vocab.md` on a branch linked to this issue.
2. Check your kwl repo before class and see if you have received feedback, reply or merge accordingly.

## 3.11. Example

Example “venn diagram “ with [mermaid subgraphs](#)



## 3.12. Badges

**Review**    **Practice**

Any steps in a badge marked **lab** are steps that we are going to focus in on during the next lab time. Remember the goal of lab is to help you complete the work, not add additional work. The lab checkout will include some other tasks and then we will encourage you to work on this badge while we are there to help. Lab checkouts are checked only for completion though, not correctness, so steps of activities that we want you to really think about and revise if incorrect will be in a practice or review badge.

1. Read the notes. If you have any questions, post an issue on the course website repo.
2. Using your terminal, download your KWL repo. Include the command used in your badge PR.
3. Try using setting up git using your favorite IDE or GitHub Desktop. Make a file gitoffline.md and include some notes of how it went. Was it hard? easy? what did you figure out or get stuck on? Is the terminology consistent or does it use different terms?

4. **lab** Explore the difference between git add and git commit: try committing and pushing without adding, then add and push without committing. Describe what happens in each case in a file called gitcommit.md. Compare what happens based on what you can see on GitHub and what you can see with git status.

### 3.13. Questions after class

## 4. Git Offline

### 4.1. Absolute Path vs Relative Path

Navigate to your inclass repo on your terminal

```
pwd
```

```
/c/Users/ayman
```

Which command will help me navigate between folders?

- [ ] ls
- [x] cd
- [ ] pwd
- [ ] mkdir

#### Note

Remember: cd stands for **c**hange **d**irectory

What type of path do I use with the command **cd**?

- [ ] absolute path only
- [ ] relative path only
- [x] Either one will work

Using absolute path means you know how to navigate to your desired folder starting from the root **/**

The **/** is the starting point to where all your files and folders can be found

Using you **relative** path, means that you know how to reach your desired folder relative to where you are right now

How do I get to Tyler 052?

Absolute path: USA/Rhode Island/Kingston/Flag rd/Green house rd/Red building/052  
Relative path: leave Ranger 302 (...)/  
leave Ranger Hall (...)/ Through quad/ pass by engineering/ Red building/ 052

The `/` between folder names means “from here, go to” We can separate the `cd` command that contains `/` into multiple commands

```
cd Documents/  
cd systems/
```

```
pwd
```

```
/c/Users/ayman/Documents/systems
```

Let's try another way

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

```
/c/Users/ayman
```

```
cd Documents/systems/
```

```
pwd
```

```
/c/Users/ayman/Documents/systems
```

We got to the same destination

Watch your steps carefully for this one (Don't hit `Enter`)

```
cd gh-
```

Hit `Tab`

```
cd gh-inclass-AymanBx
```

Now you can hit `Enter`

### Note

The terminal determined that the existing folders/files that begin with "gh-" are limited to only one and helped you complete it with the press of `tab`

## getting to GitHub from your local system

Now on the other side of things, navigate to your inclass repo on GitHub

the step below requires that you have the `gh` CLI.

```
gh repo view --web
```

What files/folders can you see on there?

```
.github/workflows
```

How many branches do you have?

```
main  
1-create-an-about-file
```

Select the 1-create-an-about-file

Same files so far (No changes made yet)

Back to your terminal

Let's go back to comparing files between online and local repos

```
ls -a
```

### Note

We used `-a` as an option for showing us hidden files/folders (anything that starts with a `.`)

```
.          .git  
..         .github
```

We'll notice a few things here

We have three extra files/folders

We know what `..` refers to by now. It's a pointer to the parent directory

The `.` is a pointer to where I am right now (the current directory)

Let's test out this theory

```
pwd
```

```
/c/Users/ayman/Documents/systems/gh-inclass-AymanBx
```

```
cd .  
pwd
```

```
/c/Users/ayman/Documents/systems/gh-inclass-AymanBx
```

Our place didn't change. It checks out!

## 4.2. .git

The third difference was the `.git` folder

`.git` folder is created by the `git` tool that we use on our terminal to hold special information about the project we're working on

Everytime we start a command with the word `git` we're calling that tool and then telling it what we want it to do

So far we saw `git status` and `git fetch`

`status` & `fetch` were the commands that we asked git to execute

```
.git is a black box ( until furhter notice ;- )  
It holds tracking information about what changes were made locally and online
```

We use `git fetch` to update the .git box with changes that happed on the GitHub repo

Let's try it

```
git fetch
```

If you got nothing then your .git box is already up to date with the online **repo** (project)

Check the current status of your project

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.

nothing to commit, working tree clean
```

`git status` is your friend, we will use it all the time to keep track of our movements

Notice two key lines here

`Your branch is up to date with 'origin/1-create-an-about-file'.` is telling you that your local project matches the online one at the moment (as far as it knows)

More importantly, `nothing to commit, working tree clean` is telling you that your local repo (everything out side of the .git folder) matches the tracking information of git (everything inside the .git folder)

**Much** more on that later

Lastly, one more small thing we noticed

Github showed us `.github/workflows` whereas `ls -a` only showed my `.github`

As discussed, the `/` tells us that this there is a file/folder within a folder

Github combined them in one because there are no other files/folders on the .github folder

How do we prove it?

```
ls .github
```

```
workflows/
```

Listing the contents of .github showed us that it contains the folder "workflows" with it

### Note

Conclusion: The `ls` command can take an **argument** that is a path and it will list the contents of the folder passed in that path for us

## 4.3. Creating a file on the terminal

The `touch` command creates an empty file.

```
touch about.md
```

We can use `ls` to see our working directory now.

```
ls
```

```
about.md
```

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.
```

```
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    about.md
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Notice: Working tree is not clean anymore.

There is one “untracked file”

This means the working directory (everything outside .git) had changes but those changes were not **added** to the .git box to be **tracked**

```
nano about.md
```

What year are you in? When do you expect to graduate?

Ctrl + s (Only windows users) Ctrl + x Enter (Only mac users would need this)

we used the [nano text editor](#). `nano` is simpler than other text editors that tend to be more popular among experts, `vim` and `emacs`. Getting comfortable with nano will get you used to the ideas, without putting as much burden on your memory. This will set you up to learn those later, if you need a more powerful terminal text editor.

We put some content in the file, any content then saved and exit.

On the nano editor the `^` stands for control.

and we can look at the contents of it.

Now we will check again with git.

`cat` concatenates the contents of a file to standard out, where all of the content that is shown on the terminal is.

### Note

Standard out is a special file in your device that your programs/commands executed will put their output into. The terminal prints the content of standard out (aka: std out), hence we get the output printed on the terminal

```
cat about.md
```

and we can see the contents

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    about.md

nothing added to commit but untracked files present (use "git add" to track)
```

## 4.4. git add

In this case both say to `git add` to track or to include in what will be committed. Under untracked files it says `git add <file>...`, in our case this would look like `git add about.md`. However, remember we learned that the `.` that is always in every directory is a special “file” that points to the current directory, so we can use that to add **all** files. Since we have only one, the two are equivalent, and the `.` is a common shortcut, because most of the time we want to add everything we have recently worked on in a single commit.

`git add` puts a file in the “staging area” we can use the staging area to group files together and put changes to multiple files in a single commit. This is something we **cannot** do on GitHub in the browser, in order to save changes at all, we have to commit. Offline, we can save changes to our computer without committing at all, and we can group many changes into a single commit.

We will use `.` as our “file” to stage everything in the current working directory.

```
git add .
```

And again, we will check in with git

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.
```

```
Changes to be committed:
(use "git restore --staged <file>..." to unstage)
  new file:   about
```

Now that one file is marked as a new file and it is in the group “to be committed”. Git also tells us how to undo the thing we just did.

Notice: git tells you that if you changed your mind on that file and you don’t want it staged to be committed to the next “checkpoint” anymore, you can simply take it out of the staging area using the command `git restore --staged <file_name>`

#### 💡 Try this yourself

Try making a change, adding it, then restoring it. Use git status to see what happens at each point

## 4.5. git commit

Next, we will commit the file. We use `git commit` for this. The `-m` option allows us to put our commit message directly on the line when we commit. Notice that unlike committing on GitHub, we do not choose our branch with the `git commit` command. We have to be “on” that branch before the `git commit`.

#### ℹ️ Note

A commit is a “checkpoint” in your project that you want to save, to be able to go back at any point in time. A commit saves the status of all the files in the project that had been modified since the last check point and **staged** for the new commit

```
git commit -m "create about - closes #1"
```

We used a [closing keyword](#) so that it will close the issue.

#### ⚠️ Warning

When you make your first commit you will need to do some [config](#) steps to set your email and user name.

```
[1-create-an-about-file c7375fa] create about - closes #1
 1 file changed, 3 insertions(+)
 create mode 100644 about.md
```

one more check

```
git status
```

```
On branch 1-create-a-readme
Your branch is ahead of 'origin/1-create-an-about-file' by 1 commit.
  (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

! Important  
If you find yourself in the middle of a file, you can press Esc to exit and return to the terminal. If you want to edit the file again, you can press Esc and then i to enter insert mode. You can also use the vi command to edit files.

#### Note

Now your working tree is clean again, but your **local** branch (as reflected inside the black box .git) now looks different from what's on the online GitHub repo (Sometimes referred to as the "Upstream")

## 4.6. git push

Git suggests that we use the **push** command to update the online repo with the local one.

And push to send to [github.com](https://github.com)

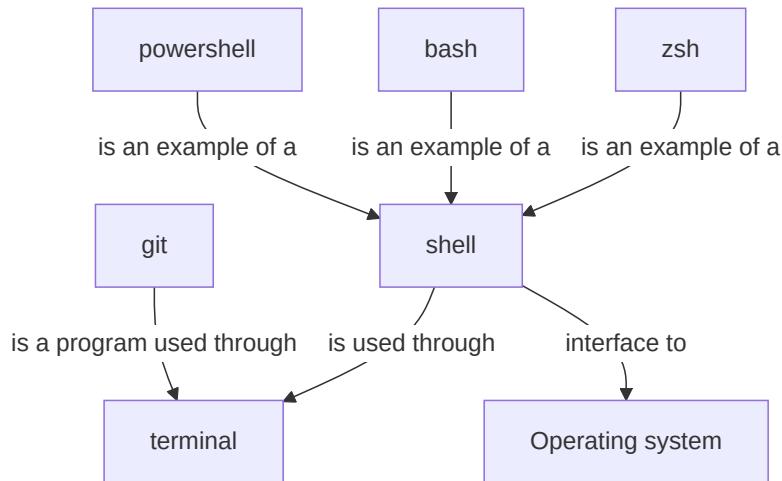
```
git push
```

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 12 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 341 bytes | 341.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/compsys-progtools/gh-inclass-AymanBx
  98ca2d6..9e8a
```

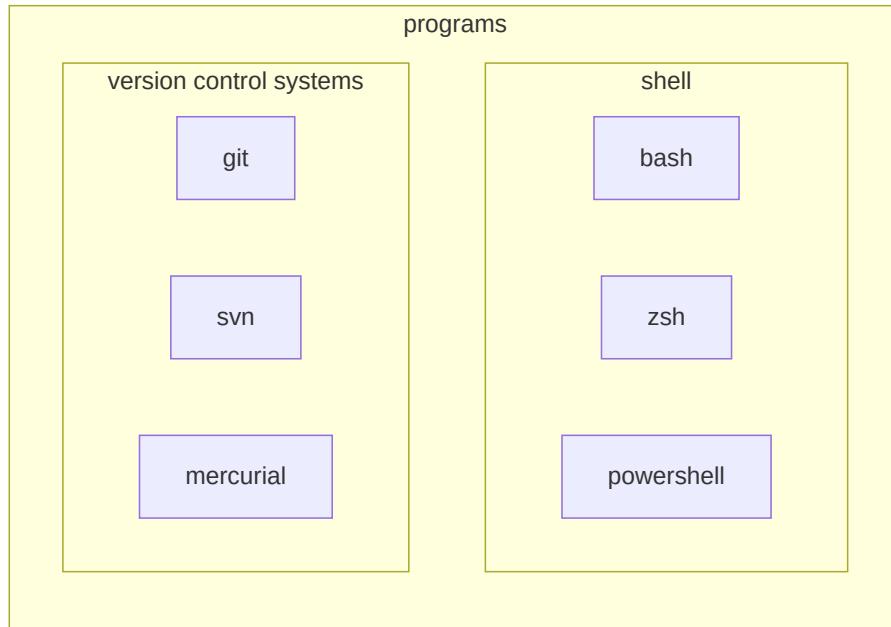
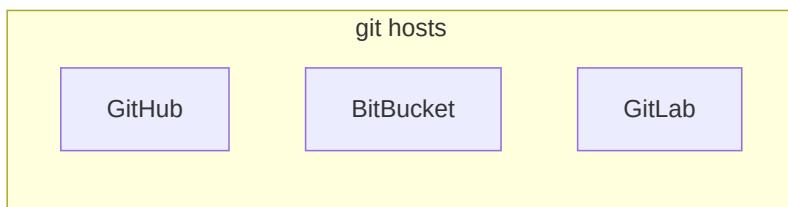
Now check out your online repo. Make sure you're on the "1-create-an-about-file" branch

[about.md](#) should exist now on there!

## 4.7. Summary



Another way to think about things (and adds some additional examples to help you differentiate between categories and examples of categories)



Today's bash commands:

command	explanation
<code>pwd</code>	print working directory
<code>cd &lt;path&gt;</code>	change directory to path
<code>mkdir &lt;name&gt;</code>	make a directory called name
<code>ls</code>	list, show the files
<code>touch</code>	create an empty file

We also learned some git commands

command	explanation
<code>status</code>	describe what relationship between the working directory and git
<code>clone &lt;url&gt;</code>	make a new folder locally and download the repo into it from url, set up a remote to url
<code>add &lt;file&gt;</code>	add file to staging area
<code>commit -m 'message'</code>	commit using the message in quotes
<code>push</code>	send to the remote

## 4.8. Prepare for next class

- [ ] Make sure you're inclass repo has two branches on github `main` & `1-create-an-about-file`
- [ ] Make sure you're inclass repo has two branches locally as well using the command `git branch` on your terminal when you're inside the inclass folder
- [ ] Make sure you can see the `about.md` file on github when you select the `1-create-an-about-file` branch and not the main branch

## 4.9. Badges

Review Practice

- [ ] Review the notes from 02-04
- [ ] Ask questions about what we did in the Experience report for 02-04

## 4.10. Questions after class

## 5. How do git branches work?

We're going to continue what we've started last week and get more practice on the interaction between git and GitHub

## 5.1. Reminder

Last week we

- Created an issue on GitHub (create an about file)
- Created a branch out of that issue (Development)
- Used the command `git fetch` to **fetch** all new changes from GitHub into the `.git` folder
- git told us that there was a new branch created called `1-create-an-about-file`
- We created a local branch that matches the name of the GitHub branch (upstream) with the command `git checkout 1-create-an-about-file`
- We created a new file through the terminal using the `touch` command
- We edited the file using a text editor available to us on the terminal named `nano`
- We added the changed file to the staging area using `git add`
- We committed the change to our local repo using `git commit -m <message>`
- We pushed the changes to the online repo using `git push`

Let's pull up our project on the terminal first

```
cd ~/Documents/systems/gh-inclass-AymanBx
```

Let's also pull up the folder in the file explorer

Always remember to check up on the status of your project

```
git status
```

```
On branch 1-create-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.
nothing to commit, working tree clean
```

Let's take a look at our files

```
ls -a
```

```
.
.
.
about.md
```

## 5.2. It's not magic

What happens if we switch branches? Keep an eye on the folder on the side while you switch branches

```
git switch main
```

Let's take a look at our files now

```
ls -a
```

```
.      .git  
..     .github
```

We can't see the file `about.md`

```
git status
```

```
On branch main  
Your branch is up to date with 'origin/main'.  
  
nothing to commit, working tree clean
```

Obviously no changes have been done to main both locally or on GitHub yet which makes sense

And finally, Let's pull up the repo online

```
gh repo view --web
```

If we're on main we can't see the `about.md` file. Let's change the branch to `1-create-an-about-file`

Now we can see the about file on the online inclass repo but only in the `create-about` branch

How do we apply the changes (new file) to main?

- [ ] commit
- [ ] git add
- [x] pull request & merge
- [ ] close the issue

If GitHub prompts you, select `compare and pull request` Or click on `contribute` and then `open pull request`

Merge pull request

Back to code section. Now we can see the [about.md](#) file!

We can also see the the number of issues went down by 1 (from 1 to none) That's because of two reasons, one of them is because the branch was created from the development section of the issue (effectively liking the issue to the branch and later to the pr). The other reason is because of the commit message that we wrote when we created the about file. We'll see this actually take affect in a few minutes.

On the terminal

```
ls -a
```

```
.      .git  
..      .github
```

The local repo is still unaware of the changes.

### 5.3. Sync

```
git fetch
```

```
remote: Enumerating objects: 4, done.  
remote: Counting objects: 100% (4/4), done.  
remote: Compressing objects: 100% (2/2), done.  
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)  
Unpacking objects: 100% (3/3), 925 bytes | 154.00 KiB/s, done.  
From https://github.com/compsys-progtools/gh-inclass-AymanBx  
  98ca2d6..4f26167  main      -> origin/main
```

```
git status
```

```
On branch main  
Your branch is behind 'origin/main' by 1 commit, and can be fast-forwarded.  
(use "git pull" to update your local branch)
```

```
nothing to commit, working tree clean
```

```
ls -a
```

```
.      .git  
..      .github
```

git is aware of some changes that were made. But the changes still aren't reflected on the file system locally.

git is so nice. It always gives us pointers as to where to go from here...

```
git pull
```

```
Updating 98ca2d6..4f26167
Fast-forward
 about.md | 3 +
 1 file changed, 3 insertions(+)
 create mode 100644 about.md
```

And finally. One last time

```
ls -a
```

```
.          .git
..          .github
about.md
```

## 5.4. Closing an issue with a commit message

Let's create a new issue (if you don't already have one) called [Create a README](#) Pay attention to what number that issue is assigned (Marked with a <#>)

Back to the code section. Click on add a README button and use the following content

contents:

```
# GitHub Practice

Name: <your name here>
```

Before you commit, take a look at the issues tab above (don't open it). What is the number showing next to the word [issues](#) ?

Commit directly to main. Add to the commit message [closes #<number\\_of\\_issue>](#)

Now take another look at the number of issues.

What happened?

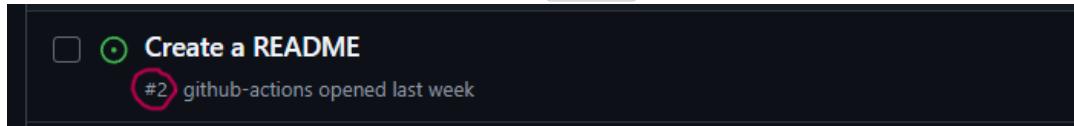
We put the special keyword `closes` in the commit message when we committed the change.

"closes #2"

What does that mean? What is #2?

#2 is referring to the issue [Create a README](#). It is labeled #2 on GitHub

Go check it out! Open the issues tab and click on `Closed`



What does that mean?

When we use the `closes` keyword in a commit message, GitHub recognizes that the commit message said `closes #2` and close the issue for us!

Other words that may work: `fixes`, `resolves`, etc.

## 5.5. Branches do not sync automatically

Now we go back to the main branch

```
git checkout main
```

```
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.
```

It thinks that we are up to date because it does not know about the changes to `origin/main` yet.

```
ls
```

```
about.md
```

the file is missing. It said it was up to date with origin main, but that is the most recent time we checked github only. It's up to date with our local record of what is on GitHub, not the current GitHub.

Next, we will update locally, with `git fetch`

```
git fetch
```

```
remote: Enumerating objects: 1, done.  
remote: Counting objects: 100% (1/1), done.  
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)  
Unpacking objects: 100% (1/1), 911 bytes | 455.00 KiB/s, done.  
From https://github.com/compsys-progtools/gh-inclass-AymanBx  
 0e7c990..0c12714  main      -> origin/main
```

Here we see 2 sets of messages. Some lines start with “remote” and other lines do not. The “remote” lines are what `git` on the GitHub server said in response to our request and the other lines are what `git` on your local computer said.

So, here, it counted up the content, and then sent it on GitHub's side. On the local side, it unpacked (remember git compressed the content before we sent it). It describes the changes that were made on the GitHub side, the main branch was moved from one commit to another. So it then updates the local main branch accordingly (“Updating 6a12db0...caeacb5”).

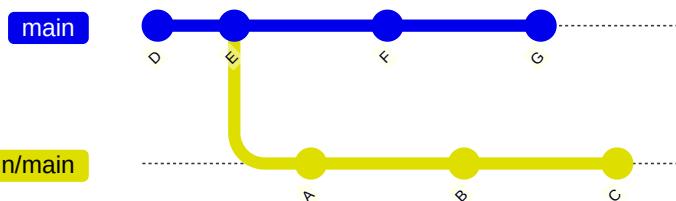
We can see that if this updates the working directory too:

```
ls
```

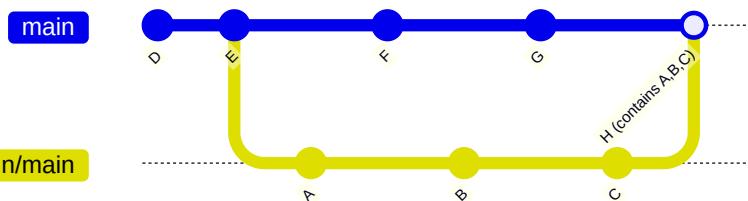
no changes yet. `fetch` updates the .git directory so that git knows more, but does not update our local file system.

```
about.md
```

### 5.5.1. Git Merge

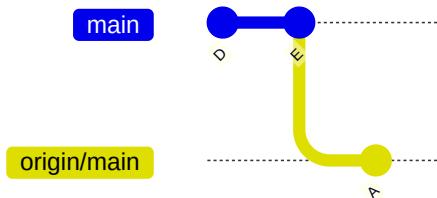


After merge, it looks like this:

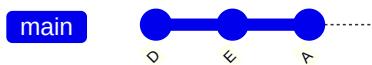


There is a new commit with the content.

In our case it is simpler:

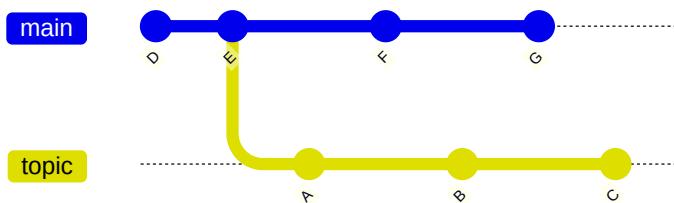


so it will fast forward

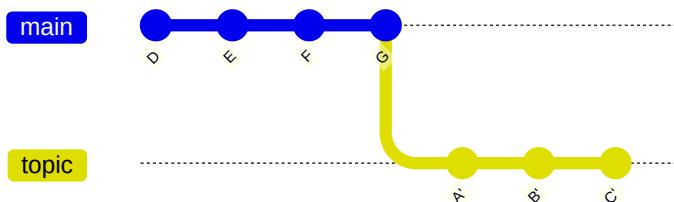


## 5.5.2. git rebase

If a repo is like this:



after:



## 5.6. Git Pull

remember `git pull` does:

1. `git fetch`
2. `git merge` (default, if possible) or `git rebase` (if settings or with option)

```
git pull
```

```
Updating 0e7c990..0c12714
Fast-forward
 README.md | 3
---
 1 file changed, 3 insertions(+)
 create mode 100644 README.md
```

Now, we can check again

```
ls
```

```
README.md      about.md
```

and it looks as expected

## 5.7. making a new branch locally

We've used `git checkout` to switch branches before. But last time we had created the branch on GitHub first and applied `git fetch`

If we use `checkout` and an unkown branch name

```
git checkout my_branch
```

```
error: pathspec 'my_branch' did not match any file(s) known to git
```

it give us an error, this does not work

`git checkout` expects there to be a branch known in the .git directory. Either a local one or one that was fetched from a remote (upstram/online) repository. If it were the latter, git creates a new local branch that [tracks](#) the online branch (meaning it links them so you can push and pull between them)

To create a branch at the same time, we use the `-b` option. with `-b` we can make a new branch and switch to it at the same time.

```
git checkout -b my_branch
```

```
Switched to a new branch 'my_branch'
```

so we see it is done!

What was this a shortcut for?

First let's go back to main

```
git checkout main
```

Or

```
git switch main
```

```
Switched to branch 'main'  
Your branch is up to date with 'origin/main'.
```

`create` does not exist

```
git branch create fun_fact
```

```
fatal: not a valid object name: 'fun_fact'
```

so it tried to treat create as a name and finds that as extra

This version gives us two new observations

```
git branch fun_fact; git checkout fun_fact
```

```
Switched to branch 'fun_fact'
```

It switches, but does not say it's new. That is because it made the branch first, then switched.

The `;` allowed us to put 2 commands in one line.

We can view a list of branches:

```
git branch
```

```
1-create-a-readme  
main  
* fun_fact  
my_branch
```

or again look at the log

```
git log
```

```
<enter log here>
```

branches are pointers, so each one is located at a particular commit.

the `-r` option shows us the remote ones

```
git branch -r
```

```
origin/1-create-a-readme
origin/HEAD -> origin/main
origin/main
```

## 5.8. Merge Conflict

```
nano about.md
```

we used the [nano text editor](#). `nano` is simpler than other text editors that tend to be more popular among experts, `vim` and `emacs`. Getting comfortable with nano will get you used to the ideas, without putting as much burden on your memory. This will set you up to learn those later, if you need a more powerful terminal text editor.

this opens the nano program on the terminal. it displays reminders of the commands at the bottom of the screen and allows you to type into the file right away.

Add any fun fact on the line below your content. Then, write out (save), it will prompt the file name. Since we opened nano with a file name (`about.md`) specified, you will not need to type a new name, but to confirm it, by pressing enter/return.

```
cat about.md
```

```
Second semester Masters
```

```
Expected graduation May 2026
- Got my BS in 2023
```

```
git status
```

```
On branch fun_fact
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   about.md

no changes added to commit (use "git add" and/or "git commit -a")
```

```
git add about.md
```

```
git status
```

```
On branch fun_fact
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   about.md
```

we are going to do it without the `-m` on purpose here to learn how to fix it

```
git commit
```

```
[fun_fact 70759fd] add fun fact
 1 file changed, 1 insertion(+)
```

```
git status
```

```
On branch fun_fact
nothing to commit, working tree clean
```

without a commit message it puts you in vim. Read the content carefully, then press `a` to get into ~insert~ mode. Type your message and/or uncomment the template.

When you are done use `escape` to go back to command mode, the ~insert~ at the bottom of the screen will go away. Then type `:wq` and press enter/return.

What this is doing is adding a temporary file with the commit message that git can use to complete your commit.

Now let's go back to `main`

```
git switch main
```

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Let's look at the contents of `about.md`

```
cat about.md
```

```
Second semster Masters
```

```
Expected graduation May 2026
```

The file `about.md` in `main` is still unaware of our changes in the `fun_fact` branch

How can we apply those changes to `main` locally?

```
git merge fun_fact
```

```
Updating 8bd4ea3..8040553
```

```
Fast-forward
 about.md | 1 +
 1 file changed, 1 insertion(+)
```

Let's check if that worked:

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 1 commit.
 (use "git push" to publish your local commits)
```

```
nothing to commit, working tree clean
```

```
cat about.md
```

```
Second semster Masters
```

```
Expected graduation May 2026
 - Got my BS in 2023
```

We can see the updated `about.md` on main now

## 5.9. Merge conflicts

We are going to *intentionally* make a merge conflict here.

This means we are learning two things:

- what *not* to do if you can avoid it
- how to fix it when a merge conflict occurs

Merge conflicts are not **always** because someone did something wrong; it can be a conflict in the simplest term because two people did two types of work that were supposed to be independent, but turned out not to be.

First, in your browser edit the [about.md](#) file to have a different fun fact.

```
Second semster Masters  
Expected graduation May 2026  
- I graduated from highschool abroad
```

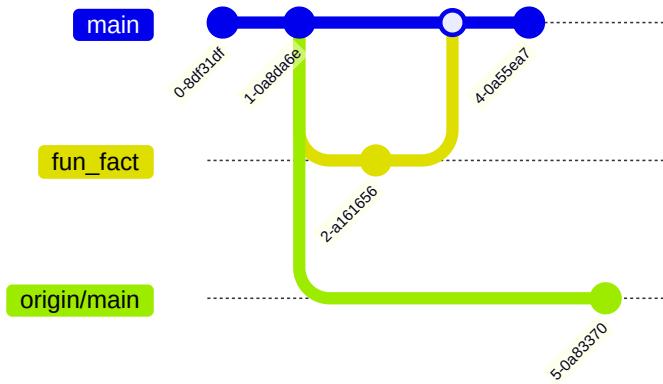
Commit changes directly to main

Now let's try updating our local repo with a [pull](#) command

```
git pull
```

```
hint: You have divergent branches and need to specify how to reconcile them.  
hint: You can do so by running one of the following commands sometime before  
hint: your next pull:  
hint:  
hint:   git config pull.rebase false  # merge  
hint:   git config pull.rebase true   # rebase  
hint:   git config pull.ff only      # fast-forward only  
hint:  
hint: You can replace "git config" with "git config --global" to set a default  
hint: preference for all repositories. You can also pass --rebase, --no-rebase,  
hint: or --ff-only on the command line to override the configured default per  
hint: invocation.  
fatal: Need to specify how to reconcile divergent branches.
```

Now it cannot work because the branches have diverged. This illustrates the fact that our two versions of the branch [main](#) and [origin/main](#) are two separate things.



git gave us some options, we will use [rebase](#) which will apply our local commits *after* the remote commits.

```
git pull --rebase
```

```

Auto-merging about.md
CONFLICT (content): Merge conflict in about.md
error: could not apply 62dcf61... local Added fun fact
hint: Resolve all conflicts manually, mark them as resolved with
hint: "git add/rm <conflicted_files>", then run "git rebase --continue".
hint: You can instead skip this commit: run "git rebase --skip".
hint: To abort and get back to the state before "git rebase", run "git rebase --abort".
hint: Disable this message with "git config advice.mergeConflict false"
Could not apply 62dcf61... local Added fun fact

```

it gets most of it, but gets stopped at a conflict.

```
git status
```

```

interactive rebase in progress; onto 462402f
Last command done (1 command done):
  pick 62dcf61 local Added fun fact
No commands remaining.
You are currently rebasing branch 'main' on '462402f'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:  about.md

no changes added to commit (use "git add" and/or "git commit -a")

```

this highlights what file the conflict is in

we can inspect this file

```
nano about.md
```

```
Second semster Masters
```

```
Expected graduation May 2026
<<<<< HEAD
- I got my BS in 2023
=====
- I graduated from highschool abroad
>>>>> "local main"
```

We have to manually edit it to be what we want it to be. We can take one change the other or both.

```
nano about.md
```

In some situations if it's actually the same line of code edited in two different ways you might choose to keep one and throw out the other. In other cases it might be two different lines of code (or fun facts) occupying the same line in the file. In this case, we will choose to keep both, so my file looks like this in the end.

```
Second semster Masters
```

```
Expected graduation May 2026
- I got my BS in 2023
- I graduated from highschool abroad
```

```
git status
```

```
interactive rebase in progress; onto 462402f
Last command done (1 command done):
  pick 62dcf61 aded fun fact
No commands remaining.
You are currently rebasing branch 'main' on '462402f'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

Unmerged paths:
  (use "git restore --staged <file>..." to unstage)
  (use "git add <file>..." to mark resolution)
    both modified:  about.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Now, we do git add and commit

```
git commit -a -m 'keep both changes'
```

```
[detached HEAD c3e68a0] keep both changes
 1 file changed, 2 insertions(+)
```

and check again

```
git status
```

```
interactive rebase in progress; onto 462402f
Last command done (1 command done):
  pick 62dcf61 local Added fun fact
No commands remaining.
You are currently editing a commit while rebasing branch 'main' on '462402f'.
(use "git commit --amend" to amend the current commit)
(use "git rebase --continue" once you are satisfied with your changes)

nothing to commit, working tree clean
```

Now, we follow the instructions again, and continue the rebase to combine our branches

```
git rebase --continue
```

```
Successfully rebased and updated refs/heads/main.
```

Once we rebase and everything is done, we can push.

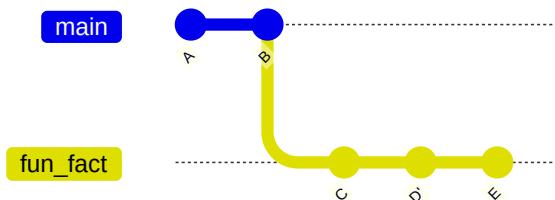
```
git push
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 309 bytes | 309.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/compsys-progtools/gh-inclass-AymanBx.git
  462402f..c3e68a0  main -> main
```

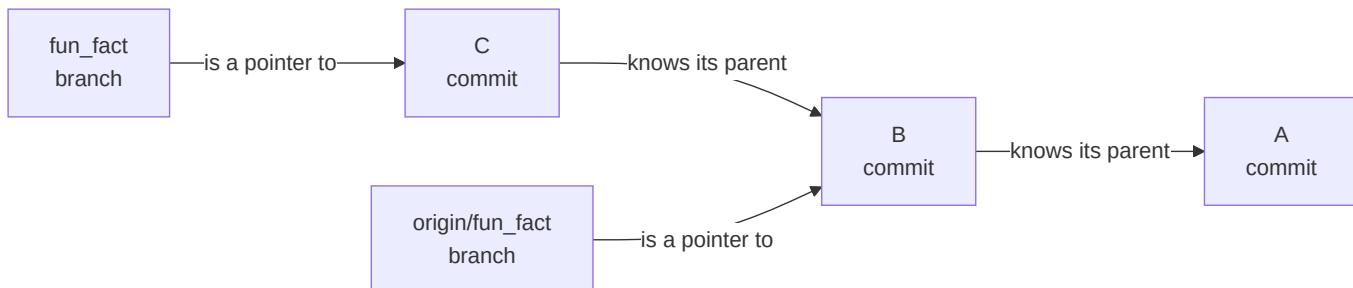
## 5.10. Summary

- branches do not sync automatically
- branches are pointers to commits
- every commit knows its parents
- if two different commits have the same parent, when we try to merge we will have divergent branches
- divergent branches can be merged by different strategies
- a merge conflict occurs if, when merging branches, a single file has been edited in two different ways

We often visualize git using graphs like subway maps:



However you can also think of what we learned today like this:



Over the next few weeks we will keep refining this understanding.

### 5.10.1. New bash commands

command	explanation
<code>cat</code>	concatenate a file to standard out (show the file contents)

### 5.10.2. New git commands

command	explanation
<code>git merge</code>	merge specified branch into the one git is currently on
<code>git branch</code>	list branches in the repo
<code>git branch new_name</code>	create a <code>new_name</code> branch
<code>git checkout -b new_Name</code>	create a <code>new_name</code> branch and switch to it
<code>git pull</code>	apply or fetch and apply changes from a remote branch to a local branch

## 5.11. Prepare for Next Class

1. Bring git questions or scenarios you want to be able to solve to class on Thursday (in your mind or comment here if that helps you remember)
2. Try read and understand the workflow files in your KWL repo, the goal is not to be sure you understand every step, but to get an idea about the big picture ideas and just enough to complete the following. Try to modify files, on a prepare branch, so that your name is already filled in when your experience badge (inclass)/Forgotten/ and Makeup action runs. We will give the answer in class, but especially do not do this step on the main branch it could break your action.

## 5.12. Badges

Review

Practice

1. Create a merge conflict in your github in class repo and resolve it using your favorite IDE,. Describe how you created it, show the files, and describe how your IDE helps or does not help in ide\_merge\_conflict.md. Give advice for when you think someone should resolve a merge conflict manually vs using an IDE. (if you do not regularly use an, IDE, try VSCode)
2. Read more details about [git branches](#)(you can also use other resources) add branches.md to your KWL repo and describe how branches work, in your own words. Include one question you have about branches or one scenario you think they could help you with.

## 5.13. Questions After Today's Class

### 5.13.1. Can you have conflicts between local branches?

Absolutely you can. If any two branches whether both are local, both online or one of each edit the same file and try to merge you will get a conflict

### 5.13.2. What happens where there are multiple people working offline and try to commit all at the same time?

Same as you would expect. There's no such thing as exactly at the same time. One commit will get pushed before the other one and the other one will either get added to it if it doesn't conflict with the first one or the branch will diverge as we saw in class and the person pushing the second commit will have to fix it.

### 5.13.3. How do I know when to use `git switch` and when to use `git checkout`?

- `git switch` always good for switching between branches and it's safe because it won't create a new branch if one doesn't already exist.
- `git checkout`: switches if a local branch exists. Creates a new branch ONLY if an online branch with the exact same name exists (and it links both branches to each other)
- `git checkout -b`: literally creates a new branch first and then switches to it. It executes two commands: `git branch <name>` + `git checkout <name>`

## 6. Terminal

### 6.1. Course Workflow

Make sure you check out [this flowchart](#) that explains class workflow to make sure you're following along correctly

## 6.2. What's happening on this terminal?

Someone asked "How do we make sure a pr's name is good enough for us to get credit on the badge?"

Answer: Well, you can use the same tool that I use for grading [courseutils](#)

Let's try it together

```
$ cspt checktitle "pracitce 2-11"
```

```
Usage: cspt [OPTIONS] COMMAND [ARGS]...
Try 'cspt --help' for help.
```

```
Error: No such command 'checktitle'.
```

Hmmm... What went wrong?

Looks like I have the **command** `checktitle` wrong.

Well, how do we do that?

```
$ cspt --help
```

```
Usage: cspt [OPTIONS] COMMAND [ARGS]...
```

```
Options:
  --help Show this message and exit.
```

```
Commands:
```

badge	log an additional badge that is not in the
badgecounts	check if early bonus is met from output of
combinecounts	combine two yaml files by adding all keys a
createtoyfiles	from a yaml source file create a set of toy
earlybonus	check if early bonus is met from output of
eventbonus	award a bonus,
exportac	export ac files for site from lesson
exporthandout	export prismia version of the content
exportprismia	export prismia version of the content
getassignment	get the assignment text formatted
getbadgedate	cli for calculate badge date
glossify	overwrite a file with all glossary terms us
grade	calculate a grade from yaml that had keys o
issuestat	generate script to appy course issue status
issuestatus	generate the activity file csv file for the
kwlcsv	award a bonus,
logquestion	transform input file to a gh markdown check
mkchecklist	process select non dates
parsedate	check json output for titles that will not
prefixlist	transform output from mac terminal export t
processexport	list PR titles from json or - to use std in
progressreport	check a single title
titlecheck	

Of course I'm not going to memorize it if I don't need to. I'm a software engineer, I use different tools on the terminal daily. I memorize most of them because of practice. What's important is not to memorize commands but to know where to find them and how to learn about them.

This line `Usage: cspt [OPTIONS] COMMAND [ARGS]...` is what I like to call a signature line.

It tell us what the shape of a command in `cspt` would look like.

If we dissect the line:

- `cspt`: The name of the program/tool
- `[OPTIONS]`: Mostly optional **option/flag** that you want your command to include
- `COMMAND`: There a list of commands that this tool will execute for you.  
Pick one
- `[ARGS]`: If the command requires an argument (mostly a type of input) pass it/add it here

```
$ cspt titlecheck "pracitce 2-11"
```

```
Usage: cspt titlecheck [OPTIONS]
Try 'cspt titlecheck --help' for help.

Error: Got unexpected extra argument (pracitce 2-11)
```

Ok, my command is still missing something, or has something extra (it says extra agrs, but is that really the problem?)

```
$ cspt titlecheck --help
```

```
Usage: cspt titlecheck [OPTIONS]
      check a single title

Options:
  -t, --pr-title TEXT    title to check as string
  -g, --ghpr FILENAME   pass title as file, or gh pr view output
  --help                  Show this message and exit.
```

Looks like we're missing an option. Aha, not so optional this time...

Notice in this help message we see `-t`, `--pr-title` `TEXT` `title to check as string` If we dissect this line:

- `-t` & `--pr-title`: are representations of the same option. The developer of this tool ([Dr. Sarah Brown](#)) has given the user two ways of using this option, a short way `-t` and a long way `--pr-title`
- `Text`: This is the argument we're passing this command. In this case this is a string with the pull request title
- `title to check as string`: Description

Question: Will there always be a short and a long way to set a flag (option)? Answer: Not necessarily, this is a convention that a lot of developers abide by. Similar to when you write a variable name in your programming language of choice: In python programmers use `snake_case` In C & C++ programmers mostly use `camelCase` And these are not rules that if you don't abide by your program will crash. These are naming conventions that are just known and used by most programmers.

```
$ cspt titlecheck -t "pracitce 2-11"
```

```
missing a badge type keyword.
```

**Understanding this piece comes with practice.**

This of a command like `commit`. We've used `git commit` as is before, and the command executed, but we still needed to include a commit message and that's why "git" forced us into `vim` to type a commit message. So, the option `-m` was partially optional, but when we use it, we simply add the message right after in double-quotes in the same command and that saves us the trouble.

We've also used the option `-a` with `git commit` last class. And we that we simply asked `git` to allow us to skip doing `git add .` and for it to do it for us. Or when we created a new local branch with the `checkout` command we used the option `-b` to ask git to create a new branch and then `checkout/switch` to it.

Getting closer. I misspelled `Practice`.

```
$ cspt titlecheck -t "practice 2-11"
```

missing or poorly formatted date

```
$ cspt titlecheck -t "practice 2-11-2025"
```

good

```
$ cspt titlecheck -t "practice 2025-2-11"
```

good

```
$ cspt titlecheck -t "practice 2/11/2025"
```

good

We can use the `--help` / `-h` option with more tools

```
$ gh -h
```

Work seamlessly with GitHub from the command line.

#### USAGE

```
gh <command> <subcommand> [flags]
```

#### CORE COMMANDS

auth:	Authenticate gh and git with GitHub
browse:	Open the repository in the browser
codespace:	Connect to and manage codespaces
gist:	Manage gists
issue:	Manage issues
org:	Manage organizations
pr:	Manage pull requests
project:	Work with GitHub Projects.
release:	Manage releases
repo:	Manage repositories

#### GITHUB ACTIONS COMMANDS

cache:	Manage Github Actions caches
run:	View details about workflow runs
workflow:	View details about GitHub Actions workflows

#### EXTENSION COMMANDS

classroom:	Extension classroom
------------	---------------------

#### ALIAS COMMANDS

co:	Alias for "pr checkout"
-----	-------------------------

#### ADDITIONAL COMMANDS

alias:	Create command shortcuts
api:	Make an authenticated GitHub API request
completion:	Generate shell completion scripts
config:	Manage configuration for gh
extension:	Manage gh extensions
gpg-key:	Manage GPG keys
label:	Manage labels
ruleset:	View info about repo rulesets
search:	Search for repositories, issues, and pull requests
secret:	Manage GitHub secrets
ssh-key:	Manage SSH keys
status:	Print information about relevant issues, pull requests, and notifications across repositories
variable:	Manage GitHub Actions variables

#### HELP TOPICS

actions:	Learn about working with GitHub Actions
environment:	Environment variables that can be used with gh
exit-codes:	Exit codes used by gh
formatting:	Formatting options for JSON data exported from gh
mintty:	Information about using gh with MinTTY
reference:	A comprehensive reference of all gh commands

#### FLAGS

--help	Show help for command
--version	Show gh version

#### EXAMPLES

```
$ gh issue create
$ gh repo clone cli/cli
$ gh pr checkout 321
```

#### LEARN MORE

Use 'gh <command> <subcommand> --help' for more information about a command.  
Read the manual at <https://cli.github.com/manual>

A shell is the outermost layer of an operating system. It is the way for users to communicate or interact with the operating system and make commands

It is very powerful

The terminal has programs installed on it such as `nano`, `cat`, `git` and more... And it has commands that it recognizes `echo`, `cd`, `pwd`, etc...

```
$ echo hello
```

```
hello
```

```
$ 5+2
```

```
bash: 5+2: command not found
```

```
$ 5+2
```

Again, I don't have the syntax here memorized, but I can get there I'm sure...

```
bash: 5+2: command not found
```

```
$ $(5+2)
```

```
bash: 5+2: command not found
```

```
$ $((5+2))
```

```
bash: 7: command not found
```

Aha!! Notice: this time we got the syntax for mathematical operation correct in `bash`. Because this time `bash` this time evaluated the result and then told us this is not a command

Let's try this one last time

```
$ echo $((5+2))
```

```
7
```

With this command, we asked bash to `echo` (print out) the evaluated value of `5+2`

`$( (5+2))` here was an argument.

```
cat -h
```

```
cat: unknown option -- h
Try 'cat --help' for more information.
```

Hmmm, `cat` doesn't have the `-h` option. What if...

```
cat --help
```

```
Usage: cat [OPTION]... [FILE]...
Concatenate FILE(s) to standard output.
```

With no FILE, or when FILE is -, read standard input.

```
-A, --show-all      equivalent to -VET
-b, --number-nonblank  number nonempty output lines, overrides -n
-e                equivalent to -VE
-E, --show-ends    display $ at end of each line
-n, --number       number all output lines
-s, --squeeze-blank suppress repeated empty output lines
-t                equivalent to -VT
-T, --show-tabs   display TAB characters as ^I
-u                (ignored)
-v, --show-nonprinting  use ^ and M- notation, except for LFD and TAB
--help            display this help and exit
--version         output version information and exit
```

Examples:

```
cat f - g  Output f's contents, then standard input, then g's contents.
cat        Copy standard input to standard output.
```

```
GNU coreutils online help: <https://www.gnu.org/software/coreutils/>
Report any translation bugs to <https://translationproject.org/team/>
Full documentation <https://www.gnu.org/software/coreutils/cat>
or available locally via: info '(coreutils) cat invocation'
```

AH!! `cat` didn't have the **short option** for the option `--help`

Someone asked "How can a terminal command `gh repo view --web` open a browser tab?

`gh` (program/tool) `repo` (command) `view` (subcommand) `--web/ -w` (options/flags)

## 6.3. Prepare for next class

No prepare work for next class

## 6.4. Badges

**Review**

**Practice**

1. [] Look over the class notes and answer the following in a file called `cat.md`

What happens if we typed the command

```
cat
```

exactly as is. Feel free to test it on your terminal.

Tell me what the `--help` message told you regarding the command being run like so. And give me a brief explanation in your own words of what that means.

## 7. What is a commit?

### 7.1. Viewing Commit History

Navigate to your in-class repo

We can see commits with `git log`

```
git log
```

```
commit 162a47b31c7a73969ce9cbcefd206c279a37433d (HEAD -> main, origin/main, origin/HEAD)
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date: Thu Feb 13 11:32:48 2025 -0500
```

Create README.md

```
commit b0b24aa53537557c582b6f61d409e9b2ee91333f
Merge: 8040553 9942c3c
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Feb 11 13:35:41 2025 -0500
```

Keeping both fun facts

```
commit 9942c3c00dcde51d52a1c110413522043eca07cd
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date: Tue Feb 11 13:24:03 2025 -0500
```

Update about.md

```
commit 804055399f6565aca3cb188fe771ef2dca99f959 (fun_fact)
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Feb 11 13:20:41 2025 -0500
```

Added fun fact

```
commit 8bd4ea38fe31186b9e5d0c1e19c9aef748c6dce6 (my_branch)
Merge: e427044 9e8a8b6
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date: Tue Feb 11 12:55:21 2025 -0500
```

Merge pull request #2 from compsys-progtools/1-create-an-about-file

Created an about file. Closes #1

```
commit e427044744061f5f38b0c3d1ff76b56552e8355d
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Feb 11 00:43:36 2025 -0500
```

removed file

.

.

.

The logs are sorted in a latest to oldest manner

this is a program, we can use enter/down arrow to move through it and then `q` to exit.

Mine will look a bit different than yours because I filled out my [README.md](#) later

### 7.1.1. How does that help?

Let's compare

```
git switch fun_fact
```

```
Switched to branch 'fun_fact'
```

```
git log
```

```
commit 804055399f6565aca3cb188fe771ef2dca99f959 (HEAD -> fun_fact)
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Tue Feb 11 13:20:41 2025 -0500

    Added fun fact

commit 8bd4ea38fe31186b9e5d0c1e19c9aef748c6dce6 (my_branch)
Merge: e427044 9e8a8b6
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date:   Tue Feb 11 12:55:21 2025 -0500

    Merge pull request #2 from compsys-progtools/1-create-an-about-file

    Created an about file. Closes #1

commit e427044744061f5f38b0c3d1ff76b56552e8355d
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Tue Feb 11 00:43:36 2025 -0500

    removed file
```

Notice what the parts of each log are

- commit: The specific checkpoint of the state of your project. It holds some metadata within it
- Author: The author of the commit.
  - Notice if you look at the different commits in your logs, you will find two different authors.
  - One that is <your\_username>@users.noreply.github.com
  - The other is Your name and email (however you have configured them locally in git)
- Date: Date and time of the commit
- Content: The message you (or GitHub) added to the commit explaining what happened in that particular commit

What do we notice when we compare both logs?

In branch `fun_fact` the latest commit that branch knows about is the one with the commit message "Added fun fact".

Everything below matches mostly logs of the `main` branch. Whereas the logs in `main` show two more commits that are more recent (three in my case)

- One with the message "Update [about.md](#)" and the author shows my GitHub user tag (with the @user)
- The other has the message "Keeping both fun facts" with the author being my locally configured user data (Yes, I used my GitHub username as my name because at the time I configured my username locally I didn't know they didn't HAVE to match)
- We also notice a `Merge` tag on that last commit with some numbers next to it

Let's try to make sense of the number in Merge:

```
8040553 9942c3c
```

Notice the individual commit identifiers (hash) for the two previous commits **9942c3c00dcde51d52a1c110413522043eca07cd** **804055399f6565aca3cb188fe771ef2dca99f959**

One more thing we will notice:

- At the latest commit in the `fun_fact` branch we see next to the commit “hash” `(HEAD -> fun_fact)`
- Next to the commit hash below it we only see `(my_branch)`

Whereas in the `main` branch logs we see:

- Next to the latest commit `(HEAD -> main, origin/main, origin/HEAD)`
- Next to the latest commit `fun_fact` branch knows we see `(fun_fact)`. Notice `HEAD` has been moved
- Next to the commit hash below it we only see `(my_branch)`

Branches are pointers, each one is located (pointing) at a particular commit.

Instead of storing important things in variables that only have scope of how long the program is active git stores its important information in files that it reads each time we run a command. All of git's data is in the `.git` directory.

Everything the git program uses is stored in the `.git` directory, you can think of that like all of the variables the program would need if it ran all the time.

```
ls .git/
```

```
COMMIT_EDITMSG  REBASE_HEAD      index          packed-refs
FETCH_HEAD      config           info           refs
HEAD            description       logs           objects
ORIG_HEAD       hooks           objects
```

the ones in all caps are simple pointers and the others are other formats.

```
cd .git/HEAD
```

```
bash: cd: .git/HEAD: Not a directory
```

## 7.2. What is a commit?

## 7.2.1. Defining terms

A commit is the most important unit of git. Later we will talk about what git as a whole is in more detail, but understanding a commit is essential to understanding how to fix things using git.

In CS we often have multiple, overlapping definitions for a term depending on our goal.

In intro classes, we try really hard to only use one definition for each term to let you focus.

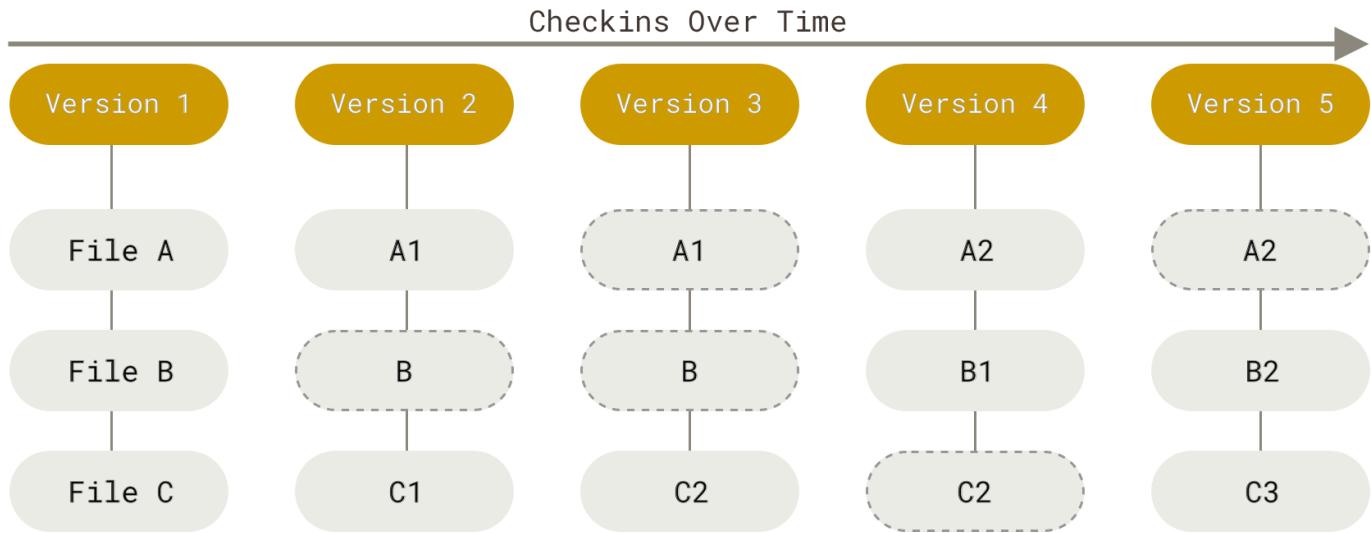
Now we need to contend with multiple definitions

These definitions could be based on

- what it conceptually represents
- its role in a larger system
- what its parts are
- how it is implemented

for a commit, today, we are going to go through all of these, with lighter treatment on the implementation for today, and more detail later.

## 7.3. Conceptually, a commit is a snapshot



git takes a full *snapshot* of the repo at each commit.

Under the hood, it only makes a *new* copy of files that have changed because it uses the same technique to store each snapshot, so any files that have not changed, do not create new files inside of git.

## 7.4. A commit's role is central to git

a commit is the basic unit of what git manages

All other git things are defined relative to commits

- branches are pointers to commits that move
  - tags are pointers to commits that do not move
  - trees are how file path/organization information is stored for a commit
  - blobs are how files contents are stored when a commit is made
- 

## 7.5. Parts of a commit

We will learn about the structure of a commit by inspecting it.

First we will go back to our `gh-inclass` repo

```
bash
:tags: ["skip-executio
```

```
cd Documents/inclass/systems/gh-inclass-sp24-brownsarahm/
```

```
---
```

```
We can use `git log` to view past commits
```

```
```bash
git log
```

```
<log_here>
```

here we see some parts:

- hash (the long alphanumeric string)
- (if merge)
- author
- time stamp
- message

but we know commits are supposed to represent some content and we have no information about that in this view

the hash is the *unique identifier* of each commit

we can view individual commits with `git cat-file` and at least 4 characters of the hash or enough to be unique. We will try 4 characters and I will use the last (my second to last) visible commit above (`b0b24aa53537557c582b6f61d409e9b2ee91333f`)

`git cat-file` has different modes:

- `-p` for pretty print

- `-t` to return the type

```
```with `git cat-file` and at least 4
characters of the hash or enough to be unique. We will try 4 characters
and I will use the last (my second to last) visible commit above (`b0b24aa53537557c582b6f61d409e9b2ee9133

`git cat-file` has different modes:
- `-p` for pretty print
- `-t` to return the type

```{code-cell} bash
git cat-file -p 1e2a
```

```
:emphasize-lines: 2,3
tree a9ebb362bb6ccac3d4cd637d4afa34d39a874a9b
parent 804055399f6565aca3cb188fe771ef2dca99f959
parent 9942c3c00dcde51d52a1c110413522043eca07cd
author AymanBx <ayman_sandouk@uri.edu> 1739298941 -0500
committer AymanBx <ayman_sandouk@uri.edu> 1739298941 -0500
```

Keeping both fun facts

Here we see the actual parts of a commit file:

- a pointer to a tree
- a pointer to two parent commits (because this was a merge) (highlighted)
- author info with timestamp
- committer info with timestamp
- commit message

### 7.5.1. What is the PGP signature?

[Signed commits](#) are extra authentication that you are who you say you are.

The commits that are labeled with the `verified` tag on [GitHub.com](#)

If we pick a commit from the history on GitHub that does not have `verified` on it, then we can see it does not have the PGP signature

## 7.6. Commit parents help us trace back

kind of like a linked list

```
:emphasize-lines: 2
tree 657c82a4b3c0aca11df1d3f9f3db46f067753120
parent 8bd4ea38fe31186b9e5d0c1e19c9aef748c6dce6
author AymanBx <ayman_sandouk@uri.edu> 1739298041 -0500
committer AymanBx <ayman_sandouk@uri.edu> 1739298041 -0500

Added fun fact
```

### 7.6.1. Commit trees are the hash of the content

This separation is helpful.

The snapshot is stored via a tree, we can use `git cat-file` to look at the tree object too.

The tree being a separate object from the overall commit allows us to be able to “edit” a message or “change” the parent of a commit; we actually make a *new* commit with the same tree.

let's look at the tree for that commit.

```
```bash
:tags: ["skip-execution"]
git cat-file -p 0689
```

```
console
:emphasize-lines: 4
040000 tree 263fb9d22090e88edd2bf1847c24c3511de91b49      .github
100644 blob 9fdc6b1b8d6b0916ef50b0a37e8c31999117016d      .gitignore
100644 blob 9ece5efa25710c8fad7d9f210928785b5362b06f      CONTRIBUTING.md
100644 blob 2d232a2231c650dc4094606797fe0bd3e0ce4c65      LICENSE.md
100644 blob b8eb6e89c6295e574ee5e3363d51c917a16797ff      README.md
040000 tree f596404cd28ea4bad49ff73fb4884049ab0e31f2      docs
100644 blob 39d5708913a6c708d1a505cde6da544785c086a6      setup.py
040000 tree 8c3cc97ca6446c270ca0b8f7d4ce640a6e81e468      src
040000 tree d3980efccf4856f0c61a6a16ed40be53
```230a5      tests
```

in this we have several columns:

- mode (indicates normal file or directory in the working directory)
- `git` object type (block or tree)
- hash of the object
- its file name in the working directory

The highlighted line for `LICENSE.md` we all have the same hash (as long as you picked a commit and tree after that file was created). This is because the hash is of the *contents* and the files all do have the same contents

## 7.6.2. Trees point to blobs of the file content

We can also use `git cat-file` to view a blob.

```
```o use `git cat-file` to view a blob.  
```{code-cell} bash  
:tags: ["skip-execution"]  
git cat-file -p 2d23
```

```
console  
the info on how the code  
```\n be reused
```

```
bash  
:tags: ["skip  
```\necution"]
```

```
++{"lesson_part":"main"}
```

## 7.7. Commits are implemented as files

commits are stored in the `.git` directory as files. git itself *is* a file system, or a way of storing information.

Everything the git program uses is stored in the `.git` directory, you can think of that like all of the variables the program would need if it ran all the time.

```
bash  
:tags: ["skip-execut  
```\n]  
ls .git
```

```
console  
COMMIT_EDITMSG  REBASE_HEAD      index          packed-refs  
FETCH_HEAD      config           info            refs  
HEAD           description       logs  
ORIG_HEAD       objects
```

the ones in all caps are simple pointers and the others are other formats.

Most of the content is in the `objects` folder, git objects are the items that get stored.

Recall, we had seen the `HEAD` pointer before

```
```{code-cell} bash
:tags: ["skip-execution"]
cat .git/HEAD
```

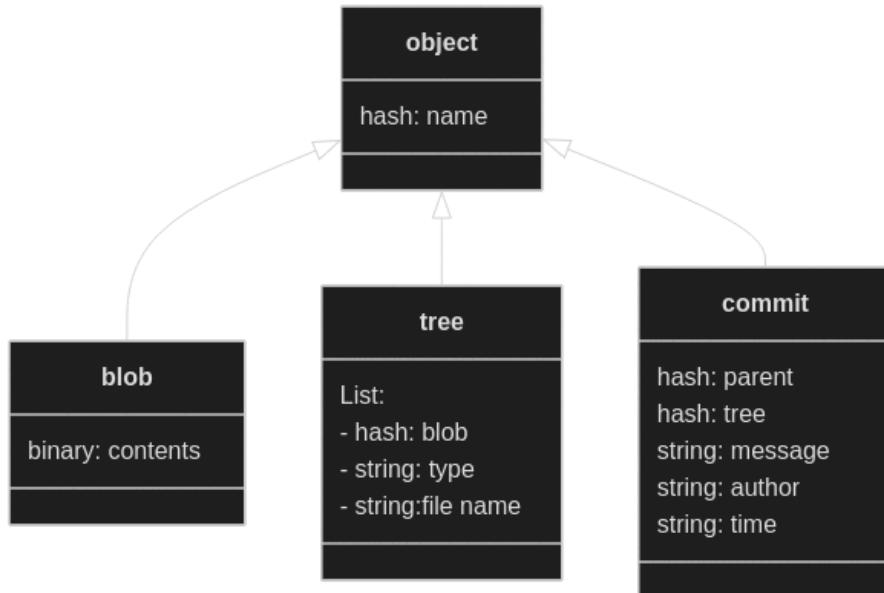
```
console
ref: refs/heads/main
````organization
```

which stores our current branch

```
bash
:tags: ["skip-execution"]
ls
````t/objects/
```

```
console
06      29      46      72      93      ab      c7      e9
0c      2d      4c      76      94      b0      ca      f1
0e      38      5b      7a      99      b1      cb      f5
10      39      5f      7c      9d      b8      d2      f9
19      3a      62      85      9e      c0      d3      info
1e      3c      63      87      9f      c2      d8      pack
1f      3d      66      8c      a3      c3      dd
25      45
````91      a8      c5      e0
```

We see a lot more folders here than we had commits. This is because there are three types of objects.



```
bash
:tags: ["skip-execution"]
cat .git/objects/29/245e4b9cce937fb9e50bc3762a
```c6a7a12c3
```

```
console
x%?A
?0Fa?9?nt!?] *(
??x?1??`Ld2???V?????eS/???P???1?aLL?EUT???!=?????fu??~?
???.???x?TItP???| )?>?'#?F¤hÝ?%?Cu?§.?
```Gb?????|Ez8
```

```
bash
git cat
```le -t 2924
```

```
```onsole
blob
```

```
bash
:tags: ["skip-execution"]
git cat
```le -p 2924
```

```
console
<last blob
```nent here>
```

```
bash
:tags: ["skip-execut
``"]"
git log
```

```
```nsole
<log>
```

```
bash
:tags: ["skip-execution
```git status
```

```
console
On branch organization
Your branch is ahead of 'origin/organization' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, work
``` tree clean
```

```
bash
:tags: ["skip-e
```ution"]
ls
```

```
console
CONTRIBUTING.md README.md      scratch.ipynb   src
LICENSE.md       docs
```up.py        tests
```

## 7.8. Prepare for next class

Examine an open source software project and fill in the template below in a file called software.md in your kwl repo on a branch that is linked to this issue. You do not need to try to understand how the code works for this exercise, but instead focus on how the repo is set up, what additional information is in there beyond the code. You may pick any mature open source project, meaning a project with recent commits, active PRs and issues, multiple contributors. In class we will have a discussion and you will compare what you found with people who examined a different project. Coordinate with peers (eg using the class discussion or in lab time) to look at different projects in order to discuss together in class.

```
## Software Reflection

Project : <markdown link to repo>

## README

<!-- what is in the readme? how well does it help you -->

## Contents

<!-- denote here types of files (code, what languages, what other files) -->

## Automation

<!-- comment on what types of stuff is in the .github directory -->

## Documentation

<!-- what support for users? what for developers? code of conduct? citation? -->

## Hidden files and support
<!-- What type of things are in the hidden files? who would need to see those files vs not? -->
```

Some open source projects if you do not have one in mind:

- [pandas](#)

- [numpy](#)
- [GitHub CLI](#)
- [Rust language](#)
- [vs code](#)
- [TypeScript](#)
- [Swift](#)
- [Jupyter book](#)
- [git-novice lesson](#)

## 7.9. Experience Report Evidence

redirect your [history](#) to a file [log-2024-02-08.txt](#) and include it with your experience report.

## 7.10. Badges

[Review](#)

[Practice](#)

1. Export your git log for your KWL main branch to a file called gitlog.txt and commit that as exported to the branch for this issue. **note that you will need to work between two branches to make this happen.** Append a blank line, [## Commands](#), and another blank line to the file, then the command history used for this exercise to the end of the file.
2. In commit-def.md compare two of the four ways we described a commit today in class. How do the two descriptions differ? How does defining it in different ways help add up to improve your understanding?

## 8. When do I get an advantage from git and bash?

### 8.1. What is a GPG signature?

[Check it out](#)

It's a way of assuring anyone viewing commits on GitHub that those commits came from known, safe sources

There are ways to configure the signature locally so that local commits are signed and verified. This is an extra way for collaborators to secure that the commit was made by the person they know and are working with and not someone else.

### 8.2. When do I get an advantage from git and bash?

so far we have used git and bash to accomplish familiar goals, and git and bash feel like just extra work for familiar goals.

Today, we will start to see why git and bash are essential skills: they give you efficiency gains and time traveling super powers (within your work, only, sorry)

## 8.3. Important references

Use these for checking facts and resources.

- [bash](#)
- [git](#)

## 8.4. Setup

First, we'll go back to our github inclass folder

```
cd Documents/systems/github-inclass-AymanBx/
```

And confirm we are where we want to be

```
pwd
```

```
/c/Users/Ayman/Documents/systems/github-inclass-AymanBx/
```

and make sure we are up to date:

```
git status
```

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

We checked if there were any uncommitted/unstaged changes that happened locally

How do we check for changes that occurred online?

Then we will use fetch to see if we are really up to date

```
git fetch
```

And let's check again to confirm

```
git status
```

```
$ git status
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

Let's check for existing prs in the repo

Find your PR that I opened for you today that has the title, "2/20 in class activity"

Note this is optional, and only works with the  cli is installed

```
gh pr list
```

```
Showing 1 of 1 open pull requests in compsys-progtools/gh-inclass-AymanBx
```

```
#4 Feb 20 in class activity organizing_ac about 2 hours ago
```

```
gh pr view
```

```
Feb 20 in class activity #3
```

```
Open • AymanBx wants to merge 1 commit into main from organizing_ac • about 2 hours ago  
+17 -0 • No checks
```

```
this draft PR https://github.blog/2019-02-14-introducing-draft-pull-requests/ adds some example files t  
during class on 9/19. \n\n wait until class time (or when you make up class by following the notes) to  
with this PR
```

```
View this pull request on GitHub: https://github.com/compsys-progtools/gh-inclass-AymanBx/pull/3
```

Since there only is one pr in your repo (most of you) it will show you details of that pr (Title, body, etc.)

If that doesn't work try using the number shown next to the pr when we listed the prs

```
gh pr view #3
```

```
Feb 20 in class activity #3
```

```
Open • AymanBx wants to merge 1 commit into main from organizing_ac • about 2 hours ago  
+17 -0 • No checks
```

```
this draft PR https://github.blog/2019-02-14-introducing-draft-pull-requests/ adds some example files t  
during class on 9/19. \n\n wait until class time (or when you make up class by following the notes) to  
with this PR
```

```
View this pull request on GitHub: https://github.com/compsys-progtools/gh-inclass-AymanBx/pull/3
```

Next get the files for today's activity:

1. Find your PR that I opened for you today that has the title, "2/20 in class activity"
2. Mark it ready for review to change from draft
3. Merge it

Let's open the PR in the browser:

```
gh repo view --web
```

```
Opening github.com/intcompsys-progtools/github-inclass-AymanBx in your browser.
```

and merge them there.

To get added to your main branch.

Let's make sure we're on main

```
git checkout main
```

Now we bring the files to the local repo:

```
git fetch
```

```
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 19 (delta 0), reused 18 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (19/19), 2.55 KiB | 84.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-AymanBx
 162a47b..581ac5e  main      -> origin/main
 * [new branch]    organizing_ac -> origin/organizing_ac
```

And let's check for changes

```
git status
```

```
On branch main
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)

nothing to commit, working tree clean
```

```
ls
```

```
README.md      about.md
```

Notice that it updated the .git, but not our working directory. It said [Your branch is behind 'origin/main' by 2 commits](#)

```
cat about.md
```

But git knows we are behind and tells us how to update

```
git pull
```

```
$ git pull
Updating 162a47b..581ac5e
Fast-forward
 API.md          | 1 +
 CONTRIBUTING.md | 1 +
 LICENSE.md      | 1 +
 _config.yml     | 1 +
 _toc.yml        | 1 +
 abstract_base_class.py | 1 +
 alternative_classes.py | 1 +
 example.md      | 1 +
 helper_functions.py | 1 +
 important_classes.py | 1 +
 philosophy.md   | 1 +
 scratch.ipynb   | 1 +
 setup.py        | 1 +
 tests_alt.py    | 1 +
 tests_helpers.py | 1 +
 tests_imp.py    | 1 +
 tsets_abc.py    | 1 +
17 files changed, 17 insertions(+)
create mode 100644 API.md
create mode 100644 CONTRIBUTING.md
create mode 100644 LICENSE.md
create mode 100644 _config.yml
create mode 100644 _toc.yml
create mode 100644 abstract_base_class.py
create mode 100644 alternative_classes.py
create mode 100644 example.md
create mode 100644 helper_functions.py
create mode 100644 important_classes.py
create mode 100644 philosophy.md
create mode 100644 scratch.ipynb
create mode 100644 setup.py
create mode 100644 tests_alt.py
create mode 100644 tests_helpers.py
create mode 100644 tests_imp.py
create mode 100644 tsets_abc.py
```

this message tells us what updates were made

Let's check what we got now:

```
$ ls
```

```
API.md          abstract_base_class.py  setup.py
CONTRIBUTING.md alternative_classes.py  tests_alt.py
LICENSE.md      example.md            tests_helpers.py
README.md       helper_functions.py   tests_imp.py
_config.yml     important_classes.py  tsets_abc.py
_toc.yml        philosophy.md        scratch.ipynb
about.md
```

## 8.5. Branches review

We can get a list of the branches we have locally

```
git branch
```

and we can get help to see what options that has with `git branch --help`

or see them with their upstream information using the `-vv` for verbose option

```
git branch -vv
```

```
1-create-an-about-file 9e8a8b6 [origin/1-create-an-about-file] Created an about file. Closes #1
  fun_fact              8040553 Added fun fact
* main                  162a47b [origin/main: behind 2] Create README.md
  my_branch              8bd4ea3 Merge pull request #2 from compsys-progtools/1-create-an-about-file
```

## 8.6. Organizing a project (working with files)

A common question is about how to organize projects. While our main focus in this class session is the `bash` commands to do it, the *task* that we are going to do is to organize a hypothetical python project

Put another way, we are using organizing a project as the *context* to motivate practicing with bash commands for moving files.

A different the instructor might go through a slide deck that lists commands and describes what each one does and then have examples at the end. Instead, we are going to focus on organizing files, and I will introduce the commands we need along the ways.

next we are going to pretend we worked on the project and made a bunch of files

I gave a bunch of files, each with a short phrase in them.

- none of these are functional files
- the phrases mean you can inspect them on the terminal

### Note

file extensions are for people; they do not specify what the file is actually written like

these are all *actually* plain text files

what command can we use to view the contents of a file named `my.file`

- [ ] shw my.file (short for show)
- [ ] prt my.file (short for print)
- [x] cat my.file (short for concatenate)
- [ ] dis my.file (short for display)

```
$ cat setup.py
```

```
file with function with instructions for pip
```

`cat` concatenates the contents of a file to stdout, which is a special file that our terminal reads

(think about in C you can write to STDOUT or STDERR, some IDEs have separate visual panels for these two places)

create a new branch from main.

```
git checkout -b organization
```

```
$ git checkout -b organization
Switched to a new branch 'organization'
```

What advantage does making a new branch here give us?

- [ ] its required for changes
- [X] allows us to test things before putting them on main
- [ ] it creates a beautiful tree eventually

```
git stauts
```

```
git: 'stauts' is not a git command. See 'git --help'.
```

```
The most similar command is
status
```

Typo...

```
git status
```

```
On branch organization
nothing to commit, working tree clean
```

## 8.7. Files, Redirects, git restore

```
cat README.md
```

```
# Practice
My name is Ayman
```

Echo allows us to send a message to stdout.

```
echo "age=27"
```

```
age = 27
```

Let's connect what we are about to learn to something you have probably seen before.

Think about a time you opened a file within a program that you wrote. For example

- `fopen` in C
- or `open` in Python

in both cases one parameter is the file to open, what other parameters have you used?

Typically, when we write to a file, in programming, we also have to tell it what *mode* to open the file with, and some options are:

- read
- write
- append

## References

- [C language docs from IBM](#)
- [Python official docs](#)

C is not an open source language in the typical sense so there is no “official” C docs. But multiple companies over the years have created compilers for the C language, so they created documentation to explain the language from their perspective.

We can **redirect** the contents of a command from stdout to a file in `bash`. Like file operations while programming there is a similar concept to this mode.

There are two types of redirects, like there are two ways to write to a file, more generally:

- overwrite (`>`)
- append (`>>`)

We can add contents to files with `echo` and `>>`

```
$ echo "age=27" >> README.md
```

Then we check the contents of the file and we see that the new content is there.

```
cat README.md
```

```
# Practice  
My name is Ayman  
age = 27
```

let's see what changes happened

```
$ git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

We can redirect other commands too:

```
git status >> curgit
```

Notice, We didn't get the output from the `git status` command this time

But we can see this created a new file

```
ls
```

```
API.md           abstract_base_class.py  setup.py
CONTRIBUTING.md alternative_classes.py  test_alt.py
LICENSE.md       ==curgit==               test_help.py
README.md        helper_functions.py   test_im.p.y
_config.yml      important_classes.py  tests_abc.py
about.md         example.md
```

and we can look at its contents too

```
cat curgit
```

```
On branch organization
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

this is not a file we actually want, which gives us a chance to learn another new bash command: `rm` for remove

```
rm curgit
```

Note that this is a true, full, and complete DELETE, this does not put the file in your recycling bin or the apple trash can that you can recover the file from, it is **gone** for real.

We will see soon a way around this, because git can help.

use `rm` with great care

Let's confirm that it's gone

```
ls
```

```
API.md      abstract_base_class.py  setup.py
CONTRIBUTING.md  alternative_classes.py  tests_alt.py
LICENSE.md    example.md          tests_helpers.py
README.md     helper_functions.py  tests_imp.py
_config.yml   important_classes.py tsets_abc.py
_toc.yml      philosophy.md      scratch.ipynb
about.md
```

Now we have made some changes we want, so let's commit our changes.

```
git commit -a -m 'add age to readme'
```

```
warning: in the working copy of 'README.md', LF will be replaced by CRLF the next time Git touches it
[organization 4ccfb5f] added age to readme
 1 file changed, 1 insertion(+)
```

```
git status
```

```
On branch organization
nothing to commit, working tree clean
```

Now, let's go back to thinking about redirects. We saw that with two `>>` we appended to the file. With just *one* what happens?

```
echo "age = 27" > README.md
```

We check the file now

```
cat README.md
```

```
age = 27
```

It wrote over. This would be bad, we lost content, but this is what git is for!

It is *very very* easy to undo work since our last commit.

This is good for times when you have something you have an idea and you do not know if it is going to work, so you make a commit before you try it. Then you can try it out. If it doesn't work you can undo and go back to the place where you made the commit.

To do this, we will first check in with git

```
git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that it tells us what to do (use "git restore <file>..." to discard changes in working directory). The version of [README.md](#) that we broke is in the working directory but not committed to git, so git refers to them as "changes" in the working directory.

```
$ git restore README.md
```

this command has no output, so we can use git status to check first

```
git status
```

```
On branch organization
nothing to commit, working tree clean
```

and it looks like it did before the > line. and we can check the file too

```
$ cat README.md
```

```
# Practice

My name is Ayman
age = 27
```

Back how we wanted it!

Now we will add some text to the readme

```
echo "|file | contents |
> | --| --
> | abstract_base_class.py | core abstract classes for the project |
> | helper_functions.py | utility functions that are called by many classes |
> | important_classes.py | classes that inherit from the abc |
> | alternative_classes.py | classes that inherit from the abc |
> | LICENSE.md | the info on how the code can be reused|
> | CONTRIBUTING.md | instructions for how people can contribute to the project|
> | setup.py | file with function with instructions for pip |
> | test_abc.py | tests for constructors and methods in abstract_base_class.py|
> | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> | tests_imp.py | tests for constructors and methods in important_classes.py|
> | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> | API.md | jupyterbook file to generate api documentation |
> | _config.yml | jupyterbook config for documentation |
> | _toc.yml | jupyter book toc file for documentation |
> | philosophy.md | overview of how the code is organized for docs |
> | example.md | myst notebook example of using the code |
> | scratch.ipynb | jupyter notebook from dev |" >> README.md
```

this explains each file a little bit more than the name of it does. We see there are sort of 5 groups of files:

- about the project/repository
- code that defines a python module
- test code
- documentation
- extra files that “we know” we can delete.

We also learn something about bash: using the open quote `"` then you stay inside that until you close it. when you press enter the command does not run until after you close the quotes

```
$ cat README.md
```

```
# Practice

My name is Ayman
age = 27
|file | contents |
> | --| --
> | abstract_base_class.py | core abstract classes for the project |
> | helper_functions.py | utilty funtions that are called by many classes |
> | important_classes.py | classes that inherit from the abc |
> | alternative_classes.py | classes that inherit from the abc |
> | LICENSE.md | the info on how the code can be reused|
> | CONTRIBUTING.md | instructions for how people can contribute to the project|
> | setup.py | file with function with instructions for pip |
> | test_abc.py | tests for constructors and methods in abstract_base_class.py|
> | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> | tests_imp.py | tests for constructors and methods in important_classes.py|
> | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> | API.md | jupyterbook file to generate api documentation |
> | _config.yml | jupyterbook config for documentation |
> | _toc.yml | jupyter book toc file for documentation |
> | philosophy.md | overview of how the code is organized for docs |
> | example.md | myst notebook example of using the code |
> | scratch.ipynb | jupyter notebook from dev |
```

```
$ ls
```

```
API.md      abstract_base_class.py  setup.py
CONTRIBUTING.md  alternative_classes.py  tests_alt.py
LICENSE.md    example.md          tests_helpers.py
README.md     helper_functions.py  tests_imp.py
_config.yml   important_classes.py  tests_abc.py
_toc.yml      philosophy.md
about.md      scratch.ipynb
```

## 8.8. Getting organized

First, we'll make a directory with `mkdir`

```
mkdir docs
```

```
$ ls
```

```
API.md      abstract_base_class.py scratch.ipynb
CONTRIBUTING.md alternative_classes.py setup.py
LICENSE.md    docs/           tests_alt.py
README.md     example.md       tests_helpers.py
_config.yml   helper_functions.py tests_imp.py
_toc.yml      important_classes.py tsets_abc.py
about.md      philosophy.md
```

next we will move a file there with `mv`

```
mv example.md docs/
```

what this does is change the path of the file from `.../github-inclass-AymanBx/example.md` to  
`.../github-inclass-AymanBx/docs/example.md`

This doesn't return anything, but we can see the effect with `ls`

```
$ ls
```

```
API.md      helper_functions.py
CONTRIBUTING.md important_classes.py
LICENSE.md    philosophy.md
README.md     scratch.ipynb
_config.yml   setup.py
_toc.yml      tests_alt.py
about.md      tests_helpers.py
abstract_base_class.py tests_imp.py
alternative_classes.py tsets_abc.py
docs/
```

We can also use `ls` with a relative or absolute path of a directory to list the location instead of our current working directory.

```
$ ls docs/
```

```
example.md
```

### 8.8.1. Moving multiple files with patterns

let's look at the list of files again.

```
$ ls
```

```
API.md           helper_functions.py
CONTRIBUTING.md important_classes.py
LICENSE.md       philosophy.md
README.md        scratch.ipynb
_config.yml      setup.py
_toc.yml         tests_alt.py
about.md         tests_helpers.py
abstract_base_class.py tests_imp.py
alternative_classes.py tsets_abc.py
docs/
```

What patterns are there in the file names that relate to what the file is for?

(use your readme or cat the files)

We can use the `*` [wildcard operator](#) to move all files that match the pattern. We'll start with the two `.yml` ([yaml](#)) files that are both for the documentation.

```
mv *.yml docs/
```

Again, we confirm it worked by seeing that they are no longer in the working directory.

```
$ ls
```

```
API.md           important_classes.py
CONTRIBUTING.md philosophy.md
LICENSE.md       scratch.ipynb
README.md        setup.py
about.md         tests_alt.py
abstract_base_class.py tests_helpers.py
alternative_classes.py tests_imp.py
docs/
helper_functions.py
```

and that they are in `docs`

```
$ ls docs/
```

```
_config.yml _toc.yml example.md
```

We see that most of the test files start with `tests_` but one starts with `tsets_`. We can fix this!

We can use `mv` to change the name as well. This is because “moving” a file and is really about changing its path, not actually copying it from one location to another and the file name is a part of the path.

```
$ mv tsets_abc.py tests_abc.py
```

```
$ ls
```

```
API.md           important_classes.py
CONTRIBUTING.md philosophy.md
LICENSE.md       scratch.ipynb
README.md        setup.py
about.md         test_abc.py
abstract_base_class.py tests_alt.py
alternative_classes.py tests_helpers.py
docs/            tests_imp.py
helper_functions.py
```

This changes the path from `..../tsets_abc.py` to `.../tests_abc.py` to. It is doing the same thing as when we use it to move a file from one folder to another folder, but changing a different part of the path.

Now we make a new folder:

```
$ mkdir tests
```

and move all of the test files there:

this is why good file naming is important even if you have not organized the whole project yet, you can use the good conventions to help yourself later.

```
$ mv test_* tests/
```

```
$ ls
```

```
API.md           important_classes.py
CONTRIBUTING.md philosophy.md
LICENSE.md       scratch.ipynb
README.md        setup.py
about.md         tests/
abstract_base_class.py tests_alt.py
alternative_classes.py tests_helpers.py
docs/            tests_imp.py
helper_functions.py
```

Nothing changed because my pattern doesn't match any existing patterns. I typed `test_*` instead of `tests_*`

```
$ mv tests_* tests/
```

```
$ ls
```

```
API.md           abstract_base_class.py  philosophy.md
CONTRIBUTING.md alternative_classes.py scratch.ipynb
LICENSE.md       docs/                  setup.py
README.md        helper_functions.py   tests/
about.md         important_classes.py
```

```
$ ls tests
```

```
test_abc.py  tests_helpers.py  
tests_alt.py  tests_imp.py
```

## 8.9. Hidden files

We are going to make a special hidden file and an extra one. We will use the following command:

```
touch .secret .gitignore
```

We also learn 2 things about `touch` and `bash`:

- `touch` can make multiple files at a time
- lists in `bash` are separated by spaces and do not require brackets

The list `.secret .gitignore` is a single `ARG` that was passed to the command

```
$ ls
```

```
API.md      abstract_base_class.py  philosophy.md  
CONTRIBUTING.md  alternative_classes.py  scratch.ipynb  
LICENSE.md    docs/                 setup.py  
README.md     helper_functions.py   tests/  
about.md      important_classes.py
```

We can't see the files. Why is that?

```
$ ls -a
```

```
./          CONTRIBUTING.md      helper_functions.py  
../         LICENSE.md        important_classes.py  
.git/       README.md        philosophy.md  
.github/    about.md        scratch.ipynb  
.gitignore  abstract_base_class.py  setup.py  
.secrets    alternative_classes.py  tests/  
API.md      docs/
```

We need the `-a` option to ask the command to show all files

lets put some content in the secret (it will actually *not* be going to GitHub)

```
echo "my dev secret" >> .secret
```

```
$ cat .secrets
```

```
my dev secret
```

Let's check the status of our project

```
$ git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:  _config.yml
    deleted:  _toc.yml
    deleted:  example.md
    deleted:  tests_alt.py
    deleted:  tests_helpers.py
    deleted:  tests_imp.py
    deleted:  tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    .secrets
    docs/
    tests/

no changes added to commit (use "git add" and/or "git commit -a")
```

### ⚠ Attention

Notice that from git's perspective, the files we moved were deleted from their original path, and saw them as new files in their respective folders. git just doesn't recognize moves (changes of paths)

Also notice that it's only showing folders as "untracked" files. It isn't showing the separate files in them and that's because all the files in them are new and untracked yet.

.gitignore lets us *not* track certain files

Making it so that git doesn't track that file's changes anymore...

### ℹ Note

This is very important, in a lot of real-life projects we tend to have certain files that are private to one developer or the group of developers that we don't want the general public (or anyone other than the developers of the project for that matter) to be able to see them. Such as API keys (like passwords), different logs, or different user configurations, etc.

let's ignore that `.secret` file

```
echo ".secret" >> .gitignore
```

```
$ git status
```

```
On branch organization
Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
 deleted: _config.yml
 deleted: _toc.yml
 deleted: example.md
 deleted: tests_alt.py
 deleted: tests_helpers.py
 deleted: tests_imp.py
 deleted: tsets_abc.py

Untracked files:
(use "git add <file>..." to include in what will be committed)
 .gitignore
 docs/
 tests/
```

Notice that .secrets is gone from the list of untracked files.

.gitignore works on patterns too!

```
mkdir my_secrets
cd my_secrets
```

```
touch a b c d e f
```

```
$ cd ..
```

```
$ git status
```

```
On branch organization
Changes not staged for commit:
(use "git add/rm <file>..." to update what will be committed)
(use "git restore <file>..." to discard changes in working directory)
 deleted: _config.yml
 deleted: _toc.yml
 deleted: example.md
 deleted: tests_alt.py
 deleted: tests_helpers.py
 deleted: tests_imp.py
 deleted: tsets_abc.py

Untracked files:
(use "git add <file>..." to include in what will be committed)
 .gitignore
 docs/
 my_secrets/
 tests/
```

```
echo "my_secrets/*" >> .gitignore
```

```
$ git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted: _config.yml
    deleted: _toc.yml
    deleted: example.md
    deleted: tests_alt.py
    deleted: tests_helpers.py
    deleted: tests_imp.py
    deleted: tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    docs/
    tests/
```

## 8.10. Copying a file

cp copies

```
cp README.md docs/overview.md
```

Notice that we copied the contents of the file and put it in a newly named file called `overview.md` all at once

```
$ ls docs
```

```
_config.yml _toc.yml example.md overview.md
```

```
$ cat docs/overview.md
```

```
# Practice
```

```
My name is Ayman
age = 27
```

## 8.11. Experience Report Evidence

Save your history with:

```
history > activity-2025-02-20.md
```

then append your git status, and the contents of your github-in-class and github-in-class/docs with to help visually separate the parts.

```
echo "***--" >> activity-2025-02-20.md
git status >> activity-2025-02-20.md
echo "***--" >> activity-2025-02-20.md
ls >> activity-2025-02-20.md
echo "***--" >> activity-2025-02-20.md
ls docs/ >> activity-2025-02-20.md
```

then edit that file (on terminal, any text editor, or an IDE) to make sure it only includes things from this activity.

## 8.12. Prepare for next class

1. Think through and make some notes about what you have learned about design so far. Try to answer the questions below in `design_before.md`. If you do not now know how to answer any of the questions, write in what questions you have.

- What past experiences with making decisions about design of software do you have?
- what experiences studying design do you have?
- What processes, decisions, and practices come to mind when you think about designing software?
- From your experiences as a user, how would you describe the design of command line tools vs other GUI tools?

## 8.13. Badges

[Review](#) [Practice](#)

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on. In this case, in lab, we will check that you know what to do, but if we want you to do revisions those will be done through the badge.

1. Update your KWL chart with the new items and any learned items. And add a new row: [bash redirects](#)
2. Clone the course website. Append the commands used and the contents of your [spring2025/.git/config](#) to a `terminal_review.md` (hint: history outputs recent commands and redirects can work with any command, not only echo). Edit the [README.md](#), commit, and try to push the changes. Describe what the error means and which [GitHub Collaboration Feature](#) you think would enable you to push? (answer in the [terminal\\_review.md](#))

fork is the answer, **must** be one of the things highlighted in the link

## 8.14. Questions after class

### 8.14.1. Are all hidden files listed in `.gitignore` and that's how you determine a file is hidden, or are there other ways a file can be hidden?

No, a file is hidden if its name begins with a `.`. And that's unrelated to what's in [.gitignore](#). What's in [.gitignore](#) hidden or not just ends up not being tracked by [git](#).

### 8.14.2. how am I suppose to remember most of these commands learned today in

## class

We have a table of commands learned on the course website. But realistically, you're not supposed to memorize them like the back of your hand. You're supposed to practice using them and understand them better and that's when you start remembering them better.

## 9. Unix Philosophy

Today we will continue organizing the github inclass repo as our working example.

Navigate to your inclass repo

Now we will add some text to the readme file

```
echo "|file | contents |
> | --| --
> | abstract_base_class.py | core abstract classes for the project |
> | helper_functions.py | utilty funtions that are called by many classes |
> | important_classes.py | classes that inherit from the abc |
> | alternative_classes.py | classes that inherit from the abc |
> | LICENSE.md | the info on how the code can be reused|
> | CONTRIBUTING.md | instructions for how people can contribute to the project|
> | setup.py | file with function with instructions for pip |
> | test_abc.py | tests for constructors and methods in abstract_base_class.py|
> | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> | tests_imp.py | tests for constructors and methods in important_classes.py|
> | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> | API.md | jupyterbook file to generate api documentation |
> | _config.yml | jupyterbook config for documentation |
> | _toc.yml | jupyter book toc file for documentation |
> | philosophy.md | overview of how the code is organized for docs |
> | example.md | myst notebook example of using the code |
> | scratch.ipynb | jupyter notebook from dev |" >> README.md
```

### 9.1. What happens if we copy to a folder that has a file of the same name?

cat docs/overview.md

```
# Practice

My name is Ayman
age = 27
```

```
touch overview.md
echo "Hello hello..." > overview.md
cat overview.md

+++{"lesson_part": "main"}
```

Hello hello...

```
ls
```

```
API.md           helper_functions.py
CONTRIBUTING.md important_classes.py
LICENSE.md       my_secrets/
README.md        overview.md
about.md         philosophy.md
abstract_base_class.py scratch.ipynb
alternative_classes.py setup.py
docs/            tests/
```

```
cat docs/overview.md
```

```
# Practice
```

```
My name is Ayman
age = 27
```

```
cp overview.md docs/overview.md
```

```
cat docs/overview.md
```

```
Hello hello...
```

### ➊ Note

Terminal is not going to tell you "There's already a file with the same name pick the one you want to keep or keep both etc..." Terminal expects you to be aware and responsible of your actions.

Let's fix it with the new content we just added to readme

```
cat README.md
```

```
# Practice

My name is Ayman
age = 27
|file | contents |

> | --| -- |

> | abstract_base_class.py | core abstract classes for the project |
> | helper_functions.py | utilty funtions that are called by many classes |
> | important_classes.py | classes that inherit from the abc |
> | alternative_classes.py | classes that inherit from the abc |
> | LICENSE.md | the info on how the code can be reused|
> | CONTRIBUTING.md | instructions for how people can contribute to the project|
> | setup.py | file with function with instructions for pip |
> | test_abc.py | tests for constructors and methods in abstract_base_class.py|
> | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> | tests_imp.py | tests for constructors and methods in important_classes.py|
> | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> | API.md | jupyterbook file to generate api documentation |
> | _config.yml | jupyterbook config for documentation |
> | _toc.yml | jupyter book toc file for documentation |
> | philosophy.md | overview of how the code is organized for docs |
> | example.md | myst notebook example of using the code |
> | scratch.ipynb | jupyter notebook from dev |
```

```
cp README.md docs/overview.md
```

```
cat docs/overview.md
```

```
# Practice

My name is Ayman
age = 27
|file | contents |

> | --| -- |

> | abstract_base_class.py | core abstract classes for the project |

> | helper_functions.py | util functions that are called by many classes |

> | important_classes.py | classes that inherit from the abc |

> | alternative_classes.py | classes that inherit from the abc |

> | LICENSE.md | the info on how the code can be reused |

> | CONTRIBUTING.md | instructions for how people can contribute to the project |

> | setup.py | file with function with instructions for pip |

> | test_abc.py | tests for constructors and methods in abstract_base_class.py |

> | tests_helpers.py | tests for constructors and methods in helper_functions.py |

> | tests_imp.py | tests for constructors and methods in important_classes.py |

> | tests_alt.py | tests for constructors and methods in alternative_classes.py |

> | API.md | jupyterbook file to generate api documentation |

> | _config.yml | jupyterbook config for documentation |

> | _toc.yml | jupyter book toc file for documentation |

> | philosophy.md | overview of how the code is organized for docs |

> | example.md | myst notebook example of using the code |

> | scratch.ipynb | jupyter notebook from dev |
```

### ! Attention

Not only did we copy the **content** of the README file to a new location. We gave the new file a different name at the same time. If using File Explorer you'd have to **Ctrl + C** from one place , **Ctrl + V** into the new place, then rename the file from [README.md](#) to [overview.md](#). We did all of that in one command!!

```
git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   README.md
    deleted:   _config.yml
    deleted:   _toc.yml
    deleted:   example.md
    deleted:   tests_alt.py
    deleted:   tests_helpers.py
    deleted:   tests_imp.py
    deleted:   tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    docs/
    overview.md
    tests/

no changes added to commit (use "git add" and/or "git commit -a")
```

If I edited multiple files in a repo, how can I stage all of them?

- [ ] git add f1, f2, f3
- [x] git add .
- [ ] git add -a
- [ ] git add

`git add .` is the common way to add everything. Why?

Because the `.` is a special to the **current** directory, so everything that has been changed inside this directory gets staged with this command.

What about the command `git add f1, f2, f3`?

Well, the only thing wrong about this command is something we've talked about before. Lists in `bash` are separated by a **space** `,` not a **comma** `,`

So if we did `git add f1 f2 f3` that would've worked. The only thing that's annoying about this method is the waste of time when writing the name of every single file that I want to add when I can simply "select all" with the special character `.`

But if we were staging more than one file all at once but NOT all the file then this is a perfect way to do so.

And finally, what about `git add -a`?

Well, let's check is there such an option `-a`?

```
git add -h
```

```

usage: git add [<options>] [--] <pathspec>...

-n, --[no-]dry-run    dry run
-v, --[no-]verbose   be verbose

-i, --[no-]interactive
                     interactive picking
-p, --[no-]patch     select hunks interactively
-e, --[no-]edit      edit current diff and apply
-f, --[no-]force     allow adding otherwise ignored files
-u, --[no-]update    update tracked files
--[no-]renormalize  renormalize EOL of tracked files (implies -u)
-N, --[no-]intent-to-add
                     record only the fact that the path will be added later
-A, --[no-]all        add changes from all tracked and untracked files
--[no-]ignore-removal ignore paths removed in the working tree (same as --no-all)
--[no-]refresh        don't add, only refresh the index
--[no-]ignore-errors  just skip files which cannot be added because of errors
--[no-]ignore-missing check if - even missing - files are ignored in dry run
--[no-]sparse          allow updating entries outside of the sparse-checkout cone
--[no-]chmod (+|-)x    override the executable bit of the listed files
--[no-]pathspec-from-file <file>
                     read pathspec from file
--[no-]pathspec-file-nul
                     with --pathspec-from-file, pathspec elements are separated with NUL character

```

There isn't such an option as `-a`, BUT, there is an option `-A` or `--all` that `add changes from all tracked and untracked files`

Now we know 😊

Let's remove the new file we created for the experiment

```
rm overview.md
```

If I edited multiple files in a repo, why would I want NOT to stage all of them at once?

We might not want to add everything all at once because we think specific changes ought to have their own commit

```
git add README.md
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    modified:   README.md

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:   _config.yml
    deleted:   _toc.yml
    deleted:   example.md
    deleted:   tests_alt.py
    deleted:   tests_helpers.py
    deleted:   tests_imp.py
    deleted:   tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    docs/
    tests/
```

```
git commit -m "Added overview of files in readme.md"
```

```
[organization 93d4ae3] Added overview of files in readme.md
  1 file changed, 37 insertions(+)
```

```
git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:   _config.yml
    deleted:   _toc.yml
    deleted:   example.md
    deleted:   tests_alt.py
    deleted:   tests_helpers.py
    deleted:   tests_imp.py
    deleted:   tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    docs/
    tests/

no changes added to commit (use "git add" and/or "git commit -a")
```

Notice that the other files remain unchanged.

Let's keep staging the changes that we made in steps that make sense.

```
ls docs/
```

```
_config.yml _toc.yml example.md overview.md
```

We moved a bunch of files related to documentation to the `docs/` folder. Let's stage and commit these changes separate than other changes

What is a quick way for me to add all yaml files?

```
git add *.yml
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:   _config.yml -> docs/_config.yml
    renamed:   _toc.yml -> docs/_toc.yml

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:   example.md
    deleted:   tests_alt.py
    deleted:   tests_helpers.py
    deleted:   tests_imp.py
    deleted:   tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    docs/example.md
    docs/overview.md
    tests/
```

Notice how when we added **all** yaml files. git added the new ones and the deleted ones (staged the deletion of the ones that were moved) It then recognized the moving of the files into the docs folder as “renaming” with two different paths being the old and new names.

Notice also how now it specifies that there are two more files that are new to the docs/ directory and aren't staged. One of which was moved from the main directory ([example.md](#))

We also notice that two new files start with docs/ Is there a quick way to add all of them?

```
git add docs/*
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:  _config.yml -> docs/_config.yml
    renamed:  _toc.yml -> docs/_toc.yml
    new file: docs/example.md
    new file: docs/overview.md

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:  example.md
    deleted:  tests_alt.py
    deleted:  tests_helpers.py
    deleted:  tests_imp.py
    deleted:  tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    tests/
```

```
git add example.md
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
    renamed:  _config.yml -> docs/_config.yml
    renamed:  _toc.yml -> docs/_toc.yml
    renamed:  example.md -> docs/example.md
    new file: docs/overview.md

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:  tests_alt.py
    deleted:  tests_helpers.py
    deleted:  tests_imp.py
    deleted:  tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    tests/
```

```
git commit -m "Organized all documentation files."
```

```
[organization 99c73d4] Organized all documentation files.
 4 files changed, 41 insertions(+)
 rename _config.yml => docs/_config.yml (100%)
 rename _toc.yml => docs/_toc.yml (100%)
 rename example.md => docs/example.md (100%)
 create mode 100644 docs/overview.md
```

```
git status
```

```

On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted:  tests_alt.py
    deleted:  tests_helpers.py
    deleted:  tests_imp.py
    deleted:  tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    tests/

      no changes added to commit (use "git add" and/or "git commit -a")

```

git log

```

commit 99c73d4f2c0c87c3aef7a61366e56e3894a040a5 (HEAD -> organization)
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Tue Feb 25 13:02:20 2025 -0500

    Organized all documentation files.

commit 93d4ae322683efa703dedbb9acb8431807a38af
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Tue Feb 25 12:53:36 2025 -0500

    Added overview of files in readme.md

commit 4ccfb5fe36073d15eaa4271d5f1355c0d9e1635b
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Thu Feb 20 13:13:59 2025 -0500

    added age to readme

commit 581ac5ef92ba5b4a647f30fc91a9d405e6900007 (origin/main, origin/HEAD, main)
Merge: 162a47b 7b56104
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date:   Thu Feb 20 11:19:10 2025 -0500

    Merge pull request #3 from compsys-progtools/organizing_ac

```

## 9.2. Discussion

At your tables:

- share your responses to the work in the prep work for today's topic
- discuss any similarities or differences
- if any, send back questions you have
- if any, send points you want to share to the whole class

How did the processes, decisions, and practices that people at your table brought up compare to what you thought of?

## 9.3. Design

In CS we tend to be more implicit about design, but that makes it hard to learn and keep track of.

Today's goal is to be more explicit, discussing some principles and seeing how you have already interacted with them.

Today we're going to do a bit more practical stuff, but we are also going to start to get into the philosophy of how things are organized.

Understanding the context and principles will help you:

- remember how to do things
- form reliable hypotheses about how to fix problems you have not seen before
- help you understand when you should do things as they have always been done and when you should challenge and change things.

### 9.3.1. Why should we study design?

- it is easy to get distracted by implementation, syntax, algorithms
- but the core *principles* of design organize ideas into simpler rules

### 9.3.2. Why are we studying developer tools?

The best way to learn design is to study examples [Schon1984, Petre2016], and some of the best examples of software design come from the tools programmers use in their own work.

#### Software design by example

*note*

- we will talk about some history in this course

This is because:

- I think that history is important context for making decisions
- when people are deeply influential, ignoring their role in history is not effective, we cannot undo what they did
- we do not have to admire them or even say their names to acknowledge the work
- computing technology has been used in Very Bad ways and in Definitely Good ways

## 9.4. Unix Philosophy

sources:

- [wiki](#)
- [a free book](#)
- composability over monolithic design
- social conventions

The tenets:

1. Make it easy to write, test, and run programs.

2. Interactive use instead of batch processing.
3. Economy and elegance of design due to size constraints ("salvation through suffering").
4. Self-supporting system: all Unix software is maintained under Unix.

For better or [worse](#) unix philosophy is dominant, so understanding it is valuable.

This critique is written that unix is not a good system for "normal folks" not in its effectiveness as a context for *developers* which is where \*nix (unix, linux) systems remain popular.

context:

*"normal folks" is the author's term; my guess is that it is attempting to describe a typical, nondeveloper computer user terminology to refer to people has changed a lot over time; when we study concepts from primary sources, we have to interpret the document through the lens of what was normal at the time the document was written, not to excuse bad behavior but to not be distracted and see what is there*

## 9.5. Philosophy in practice, using pipes

Today we will work in your main repo

```
cd spring2025-kwl-AymanBx/
```

To check if your [gh](#) CLI is working:

```
gh
```

Work seamlessly with GitHub from the command line.

#### USAGE

gh <command> <subcommand> [flags]

#### CORE COMMANDS

auth: Authenticate gh and git with GitHub  
browse: Open the repository in the browser  
codespace: Connect to and manage codespaces  
gist: Manage gists  
issue: Manage issues  
org: Manage organizations  
pr: Manage pull requests  
project: Work with GitHub Projects.  
release: Manage releases  
repo: Manage repositories

#### GITHUB ACTIONS COMMANDS

cache: Manage GitHub Actions caches  
run: View details about workflow runs  
workflow: View details about GitHub Actions workflows

#### EXTENSION COMMANDS

classroom: Extension classroom

#### ALIAS COMMANDS

co: Alias for "pr checkout"

#### ADDITIONAL COMMANDS

alias: Create command shortcuts  
api: Make an authenticated GitHub API request  
attestation: Work with artifact attestations  
completion: Generate shell completion scripts  
config: Manage configuration for gh  
extension: Manage gh extensions  
gpg-key: Manage GPG keys  
label: Manage labels  
ruleset: View info about repo rulesets  
search: Search for repositories, issues, and pull requests  
secret: Manage GitHub secrets  
ssh-key: Manage SSH keys  
status: Print information about relevant issues, pull requests, and notifications across repositor  
variable: Manage GitHub Actions variables

#### HELP TOPICS

actions: Learn about working with GitHub Actions  
environment: Environment variables that can be used with gh  
exit-codes: Exit codes used by gh  
formatting: Formatting options for JSON data exported from gh  
mintty: Information about using gh with MinTTY  
reference: A comprehensive reference of all gh commands

#### FLAGS

--help Show help for command  
--version Show gh version

#### EXAMPLES

gh issue create  
gh repo clone cli/cli  
gh pr checkout 321

#### LEARN MORE

Use `gh <command> <subcommand> --help` for more information about a command.  
Read the manual at <https://cli.github.com/manual>  
Learn about exit codes using `gh help exit-codes`

When we use it without a subcommand, it gives us help output a list of all the commands that we can use. If it is *not* working, you would get a `command not found` error.

Let's also pull up the repo on the browser

```
gh repo view --web
```

Opening [github.com/compsys-progtools/spring25-kwl-AymanBx](https://github.com/compsys-progtools/spring25-kwl-AymanBx) in

Let's inspect the action file that creates the issues for your badges.

### Note

We can also inspect the file on the terminal

```
cat .github/workflows/getassignment.yml
```

```
name: Create badge issues (Do not run manually)
on:
  workflow_dispatch

jobs:
  check-contents:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v4

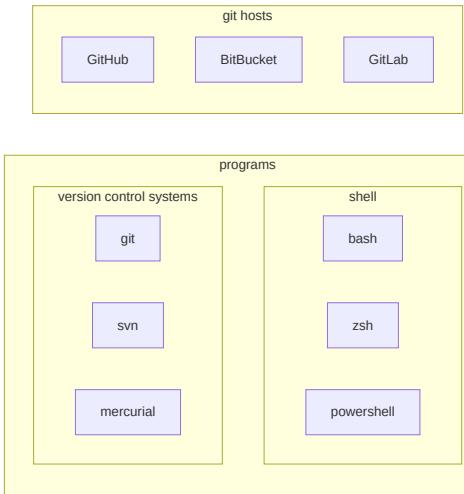
    # Install dependencies
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: 3.12

    - name: Install Utils
      run: |
        pip install git+https://github.com/compsys-progtools/
    - name: Get badge requirements
      run: |
        # prepare badge lines
        pretitle="$(cspt getbadgedate --prepare)"
        cspt getassignment --type prepare | gh issue create --
        # review badge lines
        rtitle="$(cspt getbadgedate --review)"
        cspt getassignment --type review | gh issue create --
        # practice badge lines
        pratitle="$(cspt getbadgedate --practice)"
        cspt getassignment --type practice | gh issue create
    env:
      GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    # edit the run step above for the level(s) you want.
    # You should keep the prepare, because they are required for
    #   You may choose to get only the review or only the pract
    # added comment to fix
```

In the course site, glossary terms will be linked as in the following list.

Key terms that were disambiguated today

- [terminal](#)
- [shell](#)
- [git](#)
- [bash](#)



A lot of this is familiar, but we are focused today on the three highlighted lines

Here we see a few key things:

- on the last line it uses a pipe `|` to connect a command `sysgetassignment` to the `gh issue create` command.
- the `gh issue create` command uses the `--body-file` option with a value `-`; this uses std in, but since this is to the right of the pipe, it puts the output of the first command into this option
- the 2nd to last line creates a variable (we will learn this more later) and the last line uses that variable
- the `Install Utils` step, installs the tool that we've talked about before called courseutils...

We know that this action is what is used to create badge issues, so this custom code must be what gets information from the course website to get the assignments and prepare them for your badge issues.

To confirm it worked, we use its base command

```
cspt
```

```
Usage: cspt [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  badgecounts      check if early bonus is met from output of `gh pr list...
  combinecounts    combine two yaml files by adding values
  createtoyfiles   from a yaml source file create a set of toy files with...
  earlybonus       check if early bonus is met from output of `gh pr list...
  exportac          export ac files for site from lesson
  exporthandout     export prismia version of the content
  exportprismia     export prismia version of the content
  getassignment     get the assignment text formatted
  getbadgedate      cli for calculate badge date
  grade             calculate a grade from yaml that had keys of...
  issuestat         generate script to apply course issue statuse updates
  issuestatus       generate the activity file csv file for the site...
  kwlcsv            transform input file to a gh markdown checklist, if the...
  mkchecklist        transform output from mac terminal export to myst...
  parsedate          process select non dates
  prfixlist          check json output for titles that will not be counted...
  processexport      transform output from mac terminal export to myst...
  progressreport     list PR titles from json or - to use std in that have...
  titlecheck         check a single title
```

Let's try the one we saw in the action

```
cspt getassignment --type prepare
```

```
404: Not Found
```

With default settings, it says 404.

We know that it goes online. So, for example, if you turn your wifi off and then try it again, you will get a different error.

To learn more about this, lets use the `--help` option.

A lot of CLI tools have this option. In this program, that I made, the Python library `click` that I used to make this, provides the `help` option automatically to show the documentation or lets me as the developer add help text to options.

```
cspt getassignment --help
```

```
Usage: cspt getassignment [OPTIONS]
       get the assignment text formatted

Options:
  --type TEXT  type can be {prepare, review, or practice}; default prepare
  --date TEXT  date should be YYYY-MM-DD of the tasks you want; default most
               recently posted
  --help       Show this message and exit.
```

Now we can see that it can take some options.

If we do not provide a date, I can tell you (and I should add to the documentation), it uses today's date. Since we ran this on a day with class, before the badges were posted, the file it looked for did not exist, so we got 404.

We can use the options to get the last posted prepare work

```
cspt getassignment --type prepare --date 2024-09-26
```

```
- [ ] Think through and make some notes about what you have learned about design so far. Try to answer th
```
- What past experiences with making decisions about design of software do you have?
- what experiences studying design do you have?
- What processes, decisions, and practices come to mind when you think about designing software?
- From your experiences as a user, how you would describe the design of command line tools vs other GUI t
```
```

```
gh issue create --help
```

```
Create an issue on GitHub.
```

```
Adding an issue to projects requires authorization with the `project` scope.  
To authorize, run `gh auth refresh -s project`.
```

#### USAGE

```
gh issue create [flags]
```

#### ALIASES

```
gh issue new
```

#### FLAGS

-a, --assignee	login	Assign people by their login. Use "@me" to self-assign.
-b, --body	string	Supply a body. Will prompt for one otherwise.
-F, --body-file	file	Read body text from file (use "-" to read from standard input)
-e, --editor		Skip prompts and open the text editor to write the title and body in. The first
-l, --label	name	Add labels by name
-m, --milestone	name	Add the issue to a milestone by name
-p, --project	title	Add the issue to projects by title
--recover	string	Recover input from a failed run of create
-T, --template	file	Template file to use as starting body text
-t, --title	string	Supply a title. Will prompt for one otherwise.
-w, --web		Open the browser to create an issue

#### INHERITED FLAGS

--help	Show help for command
-R, --repo	[HOST/]OWNER/REPO Select another repository using the [HOST/]OWNER/REPO format

#### EXAMPLES

```
$ gh issue create --title "I found a bug" --body "Nothing works"  
$ gh issue create --label "bug,help wanted"  
$ gh issue create --label bug --label "help wanted"  
$ gh issue create --assignee monalisa,hubot  
  
$ gh issue create --project "Roadmap"  
$ gh issue create --template "bug_report.md"
```

#### LEARN MORE

```
Use `gh <command> <subcommand> --help` for more information about a command.  
Read the manual at https://cli.github.com/manual  
Learn about exit codes using `gh help exit-codes`
```

## 9.5.1. Interactive Design

```
gh issue create
```

```
Creating issue in compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template
```

```
? Title tst  
? Body <Received>  
? What's next? Submit  
https://github.com/compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template/issues/43
```

```
gh issue list
```

```
Showing 1 of 1 open issue in compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template
```

ID	TITLE	LABELS	UPDATED
#43	tst		less than a minute ago

```
gh issue view 43
```

```
tst compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template#43
Open • AymanBx opened about 1 minute ago • 0 comments
```

No description provided

View this issue on GitHub: <https://github.com/compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template/issues/43>

Comment also allows us to work interactively.

```
gh issue comment 43
```

It uses `nano` which we have already been using, so you are well prepared for this!

```
- Press Enter to draft your comment in nano...
? Submit? Yes
https://github.com/compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template/issues/43#issuecomment-1142200000
```

We can look again

```
gh issue view 43
```

```
tst compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template#43
Open • AymanBx opened about 2 minutes ago • 1 comment
```

No description provided

AymanBx • 0m • Newest comment

lksjflakjfladksjlf;kwhf;kwh

View this issue on GitHub: <https://github.com/compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template/issues/43>

We can also close issues

```
gh issue close 43
```

```
✓ Closed issue compsys-progtools/compsys-progtools--spring2025-kwl-kwl-template#43 (tst)
```

## 9.6. Prepare for Next Class

Read the [README.md](#) file in [courseutils](#). In a file courseutils write a summary of your findings.

```

# Course Utils

## Short description
<!-- Short description of what you think this tool is for -->

## Use cases
<!-- These are bullet points. Add as many as you need -->
*
*

## Summary
<!-- Any thoughts about the tool -->

```

## 9.7. Badges

**Review** **Practice**

1. Read today's notes when they are posted. There are important tips and explanation to be sure you did.
2. Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Review the open source project you looked at for the `software.md` file from before and decide if it primarily adheres to or deviates from the unix philosophy. Add a `## Unix Philosophy <Adherence/Deviation>` section to your `software.md`, setting the title to indicate your decision and explain your decision in that section (pick one). Provide at least two specific examples supporting your choice, using links to specific lines of code or specific sections in the documentation that support your claims.

## 9.8. Experience Report Evidence

## 9.9. Questions After Today's Class

### 9.9.1. why would we not stage all files in a repo all at once

In case you wanted to separate the work you've made into different commits (checkpoints) to possibly deploy each one separately and make sure your main doesn't break. And if main does break, you can tell which commit caused it to fail to run or deploy because you don't have too many changes all in one commit.

## 10. Unix Philosophy In Practice

### 10.1. Philosophy in practice, using pipes

Last class we experimented with creating issues from the terminal with a command from the `gh` CLI (Command Line Interface)

We also looked at the `getassignment.yml` workflow and we noticed how badges get posted.

One thing we saw here was being able to write a script to automate a bunch of actions in our repo easily

But the main thing we noticed was being able to use two commands from two separate tools together at once to complete a job

Think of the line we saw

```
cspt getassignment --type practice | gh issue create --title $pratitle --label practice --body-file -
```

Here we saw a few key things:

- It uses a pipe `|` to connect a command `cspt getassignment` to the `gh issue create` command.
- the `gh issue create` command uses the `--body-file` option with a value `-` which means to use stdin
- but since this is to the right of the pipe, it puts the output of the first command into this option

We know that this action is what is used to create badge issues, so this custom code must be what gets information from the course website to get the assignments and prepare them for your badge issues.

One of the Unix Philosophy ideas were to make sure the input and output of a program are text streams to make it easier to connect different programs with each other and make them work together

"Expect the output of every program to become the input to another, as yet unknown, program. Don't clutter output with extraneous information. Avoid stringently columnar or binary input formats. Don't insist on interactive input"

### 10.1.1. Installing from source

As we saw in the action, we can install python packages from source via git. let's install the new version of the code

```
pip install git+https://github.com/compsys-progtools/courseut
```

For those of you who are having issues with the environment `externally managed environment` remember to run the virtual environment we talked about in lab

Remember, the term `pipe` refers to a method to achieve "interprocess" communication. Not necessarily only connecting the input of one process to the output of the other. Interprocess communication is used in many different situations, some of which might be when dividing a task into multiple sub-processes (multiprocessing) to use the full potential of the device and make the task quicker, one might need to construct a way for the subprocesses to communicate with each other to divide the work between them correctly, or to report results to each other or to the parent process. There are more use cases of pipes and multiprocessing than the two examples mentioned.

#### ! Attention

Do the following step outside of either one of your repos. A good place to run this command would be the `Systems` folder or `CSC311` folder. As long as its not inside either repo folders.

Create a Python `virtual environment`

```
python venv -m venv
```

Run the virtual environment

macos/linux

windows

```
source venv/bin/activate
```

```
Collecting git+https://github.com/compsys-progtools/courseutils@main
  Cloning https://github.com/compsys-progtools/courseutils (to revision main) to /private/var/folders/8g/
    Running command git clone --filter=blob:none --quiet https://github.com/compsys-progtools/courseutils /
    Resolved https://github.com/compsys-progtools/courseutils to commit 395ac8754c55a119ffa2b3670afabfe4bae
      Preparing metadata (setup.py) ... done
Requirement already satisfied: Click in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pandas in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: lxml in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pyyaml in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: numpy in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: requests in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: html5lib in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: six>=1.9 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: webencodings in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: python-dateutil>=2.8.2 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pytz>=2020.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: tzdata>=2022.7 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: charset-normalizer<4,>=2 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: idna<4,>=2.5 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: urllib3<3,>=1.21.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: certifi>=2017.4.17 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Building wheels for collected packages: syscourseutils
  Building wheel for syscourseutils (setup.py) ... done
  Created wheel for syscourseutils: filename=syscourseutils-1.0.6-py3-none-any.whl size=21580 sha256=9ef0
  Stored in directory: /private/var/folders/8g/px8bm7bj0_j31j71yh6mfd_r0000gn/T/pip-ephem-wheel-cache-6z\

Successfully built syscourseutils
Installing collected packages: syscourseutils
  Attempting uninstall: syscourseutils
    Found existing installation: syscourseutils 1.0.6
    Uninstalling syscourseutils-1.0.6:
      Successfully uninstalled syscourseutils-1.0.6
Successfully installed syscourseutils-1.0.6
```

To confirm it worked, we use its base command

```
cspt
```

```

Usage: cspt [OPTIONS] COMMAND [ARGS]...

Options:
  --help  Show this message and exit.

Commands:
  badgecounts      check if early bonus is met from output of `gh pr list...
  combinecounts    combine two yaml files by adding values
  createtoyfiles   from a yaml source file create a set of toy files with...
  earlybonus       check if early bonus is met from output of `gh pr list...
  exportac         export ac files for site from lesson
  exporthandout    export prismia version of the content
  exportprismia    export prismia version of the content
  getassignment    get the assignment text formatted
  getbadgedate    cli for calculate badge date
  grade           calculate a grade from yaml that had keys of...
  issuestat        generate script to apply course issue status updates
  issuestatus      generate the activity file csv file for the site...
  kwlcsv          transform input file to a gh markdown checklist, if the...
  mkchecklist      process select non dates
  parsedate        check json output for titles that will not be counted...
  prefixlist       transform output from mac terminal export to myst...
  processexport    list PR titles from json or - to use std in that have...
  progressreport   check a single title

```

Remember, this tool searches for the assignment information on the course website, so if the assignments weren't posted yet we get an error message

Let's try the one we saw in the action

```
cspt getassignment --type prepare
```

```
404: Not Found
```

With default settings, it says 404.

We know that it goes online. So, for example, if you turn your wifi off and then try it again, you will get a different error code.

To learn more about this, let's use the `--help` option.

A lot of CLI tools have this option. In this program, that Prof. Sarah Brown made, the Python library [click](#) that is used to make this, provides the `help` option automatically to show the documentation or lets the developer add help text to options.

```
cspt getassignment --help
```

```

Usage: cspt getassignment [OPTIONS]

get the assignment text formatted

Options:
  --type TEXT  type can be {prepare, review, or practice}; default prepare
  --date TEXT  date should be YYYY-MM-DD of the tasks you want; default most
               recently posted
  --help       Show this message and exit.

```

Now we can see that it can take some options.

If we do not provide a date, I can tell you (and I should add to the documentation), it uses today's date and attempts to get the next class' prepare info. Since we ran this on a day with class, before the badges were posted, the file it looked for did not exist, so we got 404.

We can use the options to get the last posted prepare work

```
cspt getassignment --type prepare --date 2025-02-25
```

```
- [ ] Think through and make some notes about what you have learned about design so far. Try to answer the following questions:  
...  
- What past experiences with making decisions about design of software do you have?  
- What experiences studying design do you have?  
- What processes, decisions, and practices come to mind when you think about designing software?  
- From your experiences as a user, how would you describe the design of command line tools vs other GUI tools?
```

And let's take a look at the `gh issue create` command options

```
gh issue create --help
```

```
Create an issue on GitHub.
```

```
Adding an issue to projects requires authorization with the `project` scope.  
To authorize, run `gh auth refresh -s project`.
```

#### USAGE

```
gh issue create [flags]
```

#### ALIASES

```
gh issue new
```

#### FLAGS

-a, --assignee	login	Assign people by their login. Use "@me" to self-assign.
-b, --body	string	Supply a body. Will prompt for one otherwise.
-F, --body-file	file	Read body text from file (use "-" to read from standard input)
-e, --editor		Skip prompts and open the text editor to write the title and body in. The first
-l, --label	name	Add labels by name
-m, --milestone	name	Add the issue to a milestone by name
-p, --project	title	Add the issue to projects by title
--recover	string	Recover input from a failed run of create
-T, --template	file	Template file to use as starting body text
-t, --title	string	Supply a title. Will prompt for one otherwise.
-w, --web		Open the browser to create an issue

#### INHERITED FLAGS

--help	Show help for command
-R, --repo	[HOST/]OWNER/REPO Select another repository using the [HOST/]OWNER/REPO format

#### EXAMPLES

```
$ gh issue create --title "I found a bug" --body "Nothing works"  
$ gh issue create --label "bug,help wanted"  
$ gh issue create --label bug --label "help wanted"  
$ gh issue create --assignee monalisa,hubot  
  
$ gh issue create --project "Roadmap"  
$ gh issue create --template "bug_report.md"
```

#### LEARN MORE

```
Use `gh <command> <subcommand> --help` for more information about a command.  
Read the manual at https://cli.github.com/manual  
Learn about exit codes using `gh help exit-codes`
```

And these are the options only related to creating an issue.

There are a lot more options one can do with a gh issue:

```
gh issue --help
```

```
Work with GitHub issues.
```

#### USAGE

```
gh issue <command> [flags]
```

#### GENERAL COMMANDS

```
create:      Create a new issue
list:        List issues in a repository
status:      Show status of relevant issues
```

#### TARGETED COMMANDS

```
close:       Close issue
comment:     Add a comment to an issue
delete:      Delete issue
develop:     Manage linked branches for an issue
edit:        Edit issues
lock:        Lock issue conversation
pin:         Pin a issue
reopen:      Reopen issue
transfer:    Transfer issue to another repository
unlock:      Unlock issue conversation
unpin:       Unpin a issue
view:        View an issue
```

#### FLAGS

```
-R, --repo [HOST/]OWNER/REPO  Select another repository using the [HOST/]OWNER/REPO format
```

#### INHERITED FLAGS

```
--help  Show help for command
```

#### ARGUMENTS

An issue can be supplied as argument in any of the following formats:  
- by number, e.g. "123"; or  
- by URL, e.g. "https://github.com/OWNER/REPO/issues/123".

#### EXAMPLES

```
$ gh issue list
$ gh issue create --label bug
$ gh issue view 123 --web
```

#### LEARN MORE

Use 'gh <command> <subcommand> --help' for more information about a command.  
Read the manual at <https://cli.github.com/manual>

Navigate to your inclass repos

```
gh issue list
```

``consol no open issues in compsys-progtools/gh-inclass-AymanBx

```
+++{"lesson_part": "main"}
### Interactive Design
```

```
+++{"lesson_part": "main"}
We saw that if we just run the command without any options we get the interactive issue design
```

```
```bash
gh issue create
```

```
Creating issue in compsys-progtools/compsys-progtools-fa24-fall24-kwl-template
```

```
? Title test  
? Body <Received>  
? What's next? Submit  
https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/4
```

Notice that when you get to the body of the issue, if you're a windows user, gh might prompt you to hit `enter` to launch `notepad` or possibly `nano`. If you're a macos/linux user gh will most likely prompt you to hit `enter` to launch `nano`.

Type the content of the issue body (issue description). and either save and exit with `notepad` or `ctrl + o` -> `enter` -> `ctrl + x` with nano.

Let's see if we were successful

```
gh issue list
```

```
Showing 1 of 1 open issue in compsys-progtools/gh-inclass-AymanBx
```

```
#4 test less than a minute ago
```

```
gh issue #4 --web
```

```
Work with GitHub issues.
```

#### USAGE

```
gh issue <command> [flags]
```

#### GENERAL COMMANDS

```
create:      Create a new issue
list:        List issues in a repository
status:      Show status of relevant issues
```

#### TARGETED COMMANDS

```
close:       Close issue
comment:     Add a comment to an issue
delete:      Delete issue
develop:    Manage linked branches for an issue
edit:        Edit issues
lock:        Lock issue conversation
pin:         Pin a issue
reopen:     Reopen issue
transfer:   Transfer issue to another repository
unlock:     Unlock issue conversation
unpin:      Unpin a issue
view:       View an issue
```

#### FLAGS

```
-R, --repo [HOST/]OWNER/REPO  Select another repository using the [HOST/]OWNER/REPO format
```

#### INHERITED FLAGS

```
--help  Show help for command
```

#### ARGUMENTS

An issue can be supplied as argument in any of the following formats:  
- by number, e.g. "123"; or  
- by URL, e.g. "https://github.com/OWNER/REPO/issues/123".

#### EXAMPLES

```
$ gh issue list
$ gh issue create --label bug
$ gh issue view 123 --web
```

#### LEARN MORE

Use 'gh <command> <subcommand> --help' for more information about a command.  
Read the manual at <https://cli.github.com/manual>

I got the command wrong...

```
gh issue view #4 --web
```

```
accepts 1 arg(s), received 0
```

Also wrong, let's try without the `#`

```
gh issue view 4 --web
```

```
Opening github.com/compsys-progtools/gh-inclass-AymanBx/issues/4 in your browser.
```

This shows us the issue on GitHub through the browser

What else can we do with the `gh` tool?

```
gh issue -h
```

Work with GitHub issues.

#### USAGE

```
gh issue <command> [flags]
```

#### GENERAL COMMANDS

```
create:      Create a new issue
list:        List issues in a repository
status:      Show status of relevant issues
```

#### TARGETED COMMANDS

```
close:       Close issue
comment:     Add a comment to an issue
delete:      Delete issue
develop:    Manage linked branches for an issue
edit:        Edit issues
lock:        Lock issue conversation
pin:         Pin a issue
reopen:     Reopen issue
transfer:   Transfer issue to another repository
unlock:     Unlock issue conversation
unpin:      Unpin a issue
view:       View an issue
```

#### FLAGS

```
-R, --repo [HOST/]OWNER/REPO  Select another repository using the [HOST/]OWNER/REPO format
```

#### INHERITED FLAGS

```
--help  Show help for command
```

#### ARGUMENTS

An issue can be supplied as argument in any of the following formats:

- by number, e.g. "123"; or
- by URL, e.g. "<https://github.com/OWNER/REPO/issues/123>".

#### EXAMPLES

```
$ gh issue list
$ gh issue create --label bug
$ gh issue view 123 --web
```

#### LEARN MORE

Use 'gh <command> <subcommand> --help' for more information about a command.  
Read the manual at <https://cli.github.com/manual>

```
gh issue comment
```

accepts 1 arg(s), received 0

Forgot to give an issue number

```
gh issue comment 4
```

Once again, this opens a text editor whether it's nano, vi or notepad. Fill out, save and exit

? Submit? Yes

<https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/4#issuecomment-2688714021>

Let's view the issue's details

```
gh issue view 4
```

test #4  
Open • AymanBx opened about 3 minutes ago • 1 comment

No description provided

AymanBx (Member) • 0m • Newest comment

This is a new comment

View this issue on GitHub: <https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/4>

```
cspt getassignment --type practice --date 2025-02-25
```

- [ ] Read today's notes when they are posted. There are important tips and explanation to be sure you did  
- [ ] Most real projects partly adhere and at least partly deviate from any major design philosophy or pa

## 10.2. Now with the pipe

```
cspt getassignment --type practice --date 2025-02-25 | gh issue create --title "duplicate-practice" --body
```

Creating issue in compsys-progtools/gh-inclass-AymanBx

<https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/5>

```
gh issue list
```

Showing 2 of 2 open issues in compsys-progtools/gh-inclass-AymanBx

#5	duplicate-practice	less than a minute ago
#4	test	about 4 minutes ago

Let's view the issue's details

```
gh issue view 5
```

duplicate-practice #5  
Open • AymanBx opened less than a minute ago • 0 comments

[ ] Read today's notes when they are posted. There are important tips and explanation to be sure you did and ideas for explore/build badges.  
[ ] Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Add a ## Unix Philosophy section to your software.md about how that project adheres to and deviates from the unix philosophy. Be specific, using links to specific lines of code or specific sections in the documentation that support your claims. Provide at least one example of both adhering and deviating from the philosophy and three total examples (that is 2 examples for one side and one for the other). You can see what badge software.md was previously assigned in and the original instructions on the KWL file list <https://compsys-progtools.github.io/spring2025/genindex.html>.

View this issue on GitHub: <https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/5>

gh issue view 5 --web

Opening [github.com/compsys-progtools/gh-inclass-AymanBx/issues/5](https://github.com/compsys-progtools/gh-inclass-AymanBx/issues/5) in your browser.

gh issue close 5

✓ Closed issue #5 (duplicate-practice)

Since your repos are forks, they have two remotes, or upstream repositories, that you can pull from.

We can see with `git remote`

git remote

origin  
upstream

it has a `verbose` option with the `-v` flag that shows more detail

git remote -v

```
origin https://github.com/compsys-progtools/spring25-kwl-AymanBx.git (fetch)
origin https://github.com/compsys-progtools/spring25-kwl-AymanBx.git (push)
upstream https://github.com/compsys-progtools/compsys-progtools-sp25-spring25-kwl-kwl-template (f
upstream https://github.com/compsys-progtools/compsys-progtools-sp25-spring25-kwl-kwl-template (p
```

We can change which of those is a default

```
gh repo set-default origin
```

```
expected the "[HOST/]OWNER/REPO" format, got "origin"
```

Most students didn't find two remote repos.

Not sure if the reason for that is the way the repo was cloned into your local device. Or some configuration settings on your device

I forgot the format and tried to use the short name, `origin` when `gh` requires it to be in `owner/repo` format, but this error is informative, so we know what to do:

### ⚠️ Important

Do this step!! It will fix some of your errors

```
gh repo set-default compsys-progtools/spring25-kwl-AymanBx
```

```
✓ Set compsys-progtools/spring25-kwl-AymanBx as the default repository for the current directory
```

Now, let's do more work on the inclass repo

Navigate to inclass repo

```
cat README.md
```

```
# Practice

My name is Ayman
age = 27
|file | contents |

> | --| -- |

> | abstract_base_class.py | core abstract classes for the project |

> | helper_functions.py | utilty funtions that are called by many classes |

> | important_classes.py | classes that inherit from the abc |

> | alternative_classes.py | classes that inherit from the abc |

> | LICENSE.md | the info on how the code can be reused |

> | CONTRIBUTING.md | instructions for how people can contribute to the project |

> | setup.py | file with function with instructions for pip |

> | test_abc.py | tests for constructors and methods in abstract_base_class.py |

> | tests_helpers.py | tests for constructors and methods in helper_functions.py |

> | tests_imp.py | tests for constructors and methods in important_classes.py |

> | tests_alt.py | tests for constructors and methods in alternative_classes.py |

> | API.md | jupyterbook file to generate api documentation |

> | _config.yml | jupyterbook config for documentation |

> | _toc.yml | jupyter book toc file for documentation |

> | philosophy.md | overview of how the code is organized for docs |

> | example.md | myst notebook example of using the code |

> | scratch.ipynb | jupyter notebook from dev |
```

When I copied the content that I wanted to add to the README file the arrows from my terminal got copied as well, ruining the table in markdown syntax we were trying to create

Let's try to fix this with codespace

```
gh remote view --web
```

```
unknown command "remote" for "gh"

Usage: gh <command> <subcommand> [flags]

Available commands:
  alias
  api
  auth
  browse
  cache
  classroom
  co
  codespace
  completion
  config
  extension
  gist
  gpg-key
  issue
  label
  org
  pr
  project
  release
  repo
  ruleset
  run
  search
  secret
  ssh-key
  status
  variable
  workflow
```

The correct command is `repo` not `remote`

```
gh repo view --web
```

Opening [github.com/compsys-progtools/gh-inclass-AymanBx](https://github.com/compsys-progtools/gh-inclass-AymanBx) in your browser.

We don't see any of the changes we had made to this repo organizing the files

Let's try switching to the `Organization` branch

Ooh! There isn't an `Organization` branch!!

Let's check what's going on locally...

```
git status
```

```

On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    deleted: tests_alt.py
    deleted: tests_helpers.py
    deleted: tests_imp.py
    deleted: tsets_abc.py

Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    tests/

no changes added to commit (use "git add" and/or "git commit -a")

```

Hmmm, No mentions on whether this branch is up to date with an online branch, ahead or behind...

That's because we created this branch locally and we've been working locally all this time but we never pushed this branch or the changes made to it online

```
git push
```

```

fatal: The current branch test has no upstream branch.
To push the current branch and set the remote as upstream, use

  git push --set-upstream origin Organization

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.

```

This message is basically telling me, in order to push changes to GitHub, you must have a branch on GitHub (preferably with the same name) to track the changes that occur to your local branch. It gives us a hint on how to do so...

```
git push -u origin organization
```

```

Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 12 threads
Compressing objects: 100% (8/8), done.
Writing objects: 100% (9/9), 1.41 KiB | 1.41 MiB/s, done.
Total 9 (delta 3), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
remote:
remote: Create a pull request for 'organization' on GitHub by visiting:
remote:   https://github.com/compsys-progtools/gh-inclass-AymanBx/pull/new/organization
remote:
To https://github.com/compsys-progtools/gh-inclass-AymanBx
 * [new branch]      organization -> organization
branch 'organization' set up to track 'origin/organization'.

```

Now, let's work on GitHub Codespace

[Codespaces](#) are a virtual machine that you can use VSCode on in browser. You only have VSCode access to this system, but VSCode with the terminal is a lot of power.

If VSCode is new to you, use their documentation of the [VSCode interface](#) to get oriented to the different parts of the screen.

today we will also work in github codespaces.

1. Navigate to your github inclass repo on [Github.com](#)
2. Use the link in the README or the green code button to open a new codespace on main.

when your codespace is open, share its name (first part of the url 2 words)

a codespace is a virtual machine on a cloud platform, not cloud access to [github.com](#)

what might that mean about how we use it?

- you need to commit changes
- codespace is linux

#### **this is why we learn bash in this class**

as developers, we will all interact with linux/unix at times, so bash is the best shell to know if you only know one or do not want to switch between multiple

Multiple cursors are your friend

#### docs

Let's use multiple cursors to remove the unnecessary  and extra line in our READMEs

Remember to ,  and 

Codespace is a virtual machine, any work you do on it will need the same process that you would do on the personal device.

 **if** your push gets rejected, read the hints, it probably has the answer.

## 10.3. Hack the course

 build and explore ideas

Develop new command line tools, github actions, VS Code (or other IDE) extensions, visual aids, etc. that help future students in the course. This is a way for you to demonstrate your learning and contribute to open source repos that you can then link to on a portfolio website or resume

You can participate in this at different levels (or multiple):

- build: writing, documenting at the API level, and configuring major pieces (can be collaborative)
- explore: writing example/tutorial style docs that another student (or team) build (mostly solo; open to justification for collab)
- explore: adding summary/visuals to the notes
- community: (test option): testing other students contributions and making descriptive issues or writing short reviews (~3-5 sentences); must not be spammy
- community: (review option): reviewing PRs submitted by a classmate (can be within a collaborative build, but must be not your own code) a single PR can have multiple reviews if sensible and not duplicative; no spam

- community: (contribution) add a glossary term to this site

see the [bonus table](#) for more information

## 10.4. Recap

Why do I need a terminal

1. replication/automation
2. it's always there and doesn't change
3. it's faster when you know it

So, the shell is the feature that interacts **with** the operating system **and** then the terminal **is** the gui that

## 10.5. Why do we need this for computer systems?

### 10.5.1. Computer Systems are designed by people

Computer Science is not a natural science like biology or physics where we try to understand some aspect of the world that we live in. Computer Science as a discipline, like algorithms, mostly derives from Math.

So, when we study computer science, while parts of it are limited by physics, most of it is essentially an imaginary world that is made by people. Understanding how people think, both generally, and common patterns within the community of programmers understand how things work and why they are the way they are. The why can also make it easier to remember, or, it can help you know what things you can find alternatives for, or even where you might invent a whole new thing that is better in some way.

Of course, not *all* programmers think the same way, but when people spend time together and communicate, they start to share patterns in how they think. So, while you do **not** have to think the same way as these patterns, knowing what they are will help you reading code, and understanding things.

### 10.5.2. Context Matters

This context of how things were developed can influence how we understand it. We will also talk about the history of computing as we go through different topics in class so that we can build that context up.

### 10.5.3. Optimal is relative

The “best” way to do something is always relative to the context. “Best” is a vague term. It could be most computationally efficient theoretically, fastest to run on a particular type of hardware, or easiest for another programmer to read.

## 10.6. Prepare for Next Class

1. If on windows, you may need to reinstall gitbash or follow other steps from the [gh docs mintty page](#) for the following steps to work locally
2. install [jupyterbook](#) this is **different** from [jupyter lab](#) or [jupyter notebook](#) that 310 uses
3. Make sure that the [gh](#) CLI tool works by using it to create an issue called test on your kwl repo with [gh issue create](#).  
If on Windows try reinstalling with mintty
4. Find 3 examples of documentation for libraries, frameworks, or developer tools that you have used and make a post on the [class discussion board](#)

## 10.7. Experience Report Evidence

## 10.8. Badges

[Review](#)   [Practice](#)

TBD

## 10.9. Questions

# 11. How do programmers communicate about code?

## 11.1. Commit messages are essential

A git commit message must exist and is always for people, but can also be for machines.

The [conventional commits standard](#) is a format of commits

If you use this, then you can use automated tools to generate a full change log when you release code

[Tooling and examples of conventional commits](#)

That is a core example of the types of detailed communication we do in programming that is embedded into the work.

Where else have you seen how communication *about* the work shape the work

## 11.2. Why Documentation

Today we will talk about documentation, there are several reasons this is important:

- **using** official documentation is the best way to get better at the tools
- understanding how documentation is designed and built will help you use it better
- **writing and maintaining** documentation is really important part of working on a team

- documentation building tools are a type of developer tool (and these are generally good software design)

Design is best learned from examples. Some of the best examples of software *design* come from developer tools.

- [source \(js version\)](#)
- [source \(python version\)](#)

In particular documentation tools are really good examples of:

- pattern matching
- modularity and abstraction
- automation
- the build process beyond compiling

By the end of today's class you will be able to:

- describe different types of documentation
- find different information in a code repo
- generate documentation as html
- create a repo locally and push to GitHub

Plus we will reinforce things we have already seen:

- ignoring content from a repo
- paths
- good file naming

## 11.3. What is documentation

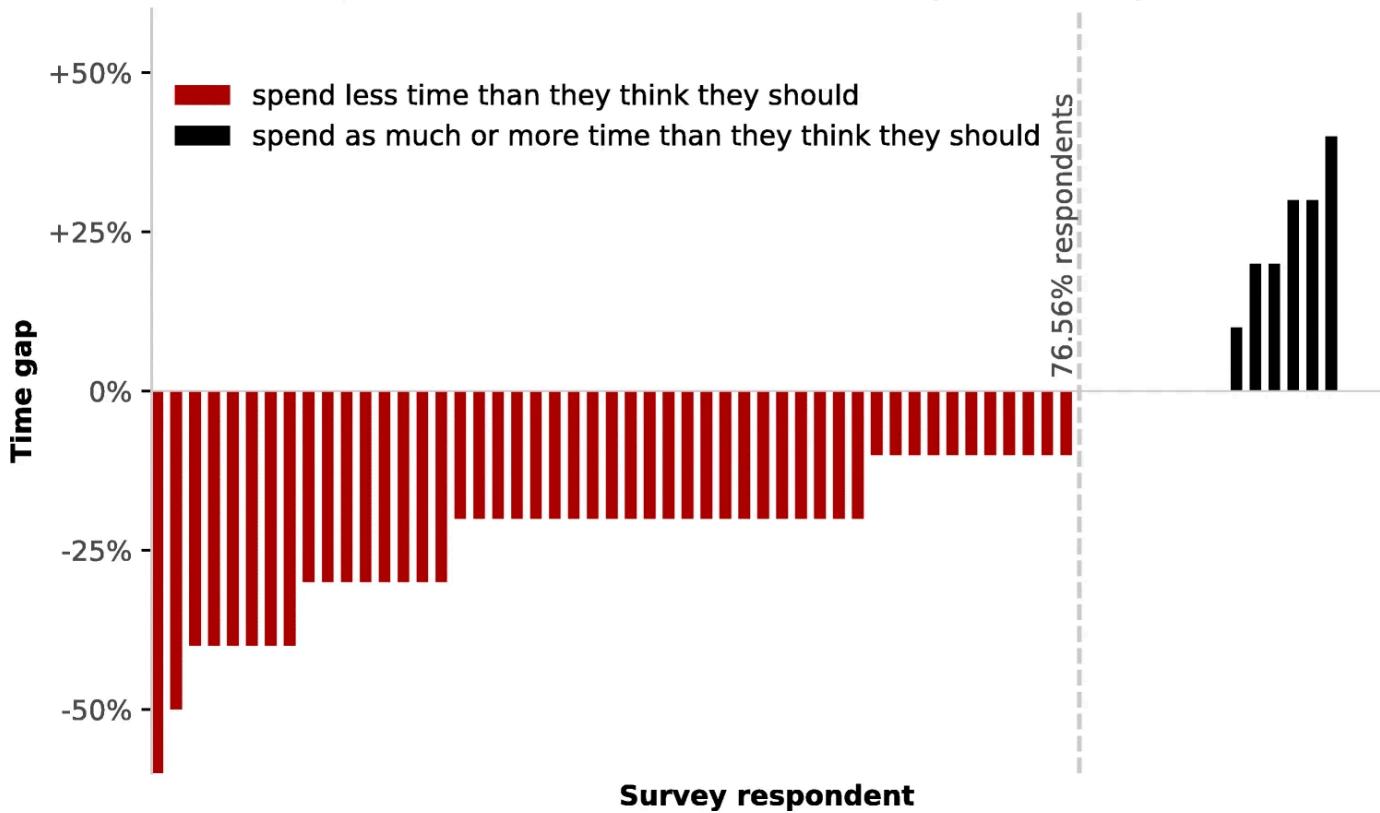
[documentation types table](#)

from [ethnography of documentation data science](#)

### 11.3.1. Why is documentation so important?

we should probably spend more time on it

**Less than 25% of respondents spend as much or more time writing documentation than what they think they should.**



[via source](#)

## 11.4. So, how do we do it?

Different types of documentation live in different places and we can use tools to maintain them.

As developers, we rely on code to do things that are easy for computers and hard for people.

[Documentation Tools](#)

[write the docs](#)

[linux kernel uses sphinx](#) and here is [why](#) and [how it works](#)

Why is there so much emphasis on documentation?

Back in the early days of programming, developers were primarily focused on writing code and achieving functional results. There were no established disciplines that enforced best practices for testing, modularity, or documentation.

As a result, programmers often wrote large, complex chunks of code with little to no explanation of how they worked. Variable names were frequently arbitrary, such as `1`, `2`, `a`, `b`, `z`, or `s` providing no meaningful context.

Do those names convey anything useful to you? Probably not. Now, imagine encountering them in a 500+ line program. The only way to understand their purpose would be to manually track every instance of their use—an exhausting and error-prone process.

This lack of clarity led to major challenges in code maintenance, collaboration, and debugging. As software development evolved, so did the need for structured documentation. Today, well-documented code is considered a best practice, ensuring that projects remain understandable, scalable, and maintainable over time.

## 11.5. Jupyterbook

[Jupyterbook](#) wraps sphinx and uses markdown instead of restructured text. The project authors [note in the documentation](#) that it “can be thought of as an *opinionated distribution of Sphinx*”. We’re going to use this on our kwl repos.

navigate to your folder for this course (mine is [Documents/systems](#))

```
cd Documents/systems/
```

We can confirm that [jupyter-book](#) is installed by checking the version.

```
jupyter-book --version
```

```
$ jupyter-book --version
Jupyter Book      : 1.0.3
External ToC       : 1.0.1
MyST-Parser        : 2.0.0
MyST-NB           : 1.2.0
Sphinx Book Theme : 1.1.4
Jupyter-Cache     : 1.0.1
NbClient          : 0.10.2
```

We will run a command to create a jupyterbook from a template, the command has 3 parts:

- [jupyter-book](#) is a program (the thing we installed)
- [create](#) is a subcommand (one action that program can do)
- [tiny-book](#) is an argument (a mandatory input to that action)

```
jupyter-book create tiny-book
```

```
=====
Your book template can be found at
  tiny-book/
=====
```

We see that it succeeds

You can make it with any name:

```
jupyter-book create examplew
```



This is

[add](#) .

- [g](#)
- [a](#)
- [.](#)

```
=====
Your book template can be found at
```

```
examplew\
```

because the name is an argument or input

Each one makes a directory, we can see by listing

```
ls
```

```
examplew/           spring25-kwl-AymanBx/
gh-inclass-AymanBx/  tiny-book/
```

And we can delete the second one since we do not actually want it.

```
rm examplew/
```

```
rm: examplew/: is a directory
```

we get an error because it is not well defined to delete a directory, and potentially risky, so `rm` is written to throw an error

Usually to remove a directory with the `rm` command we use the `-d` option

```
rm -d examplew/
```

```
rm: cannot remove 'examplew/': Directory not empty
```

Notice, we still got an error, but this time the error is different. It's telling us that the directory isn't empty. So this means the `-d` option is only for empty directories.

Instead, we have to tell it two additional things:

- to delete recursively `r`
- to force it to do something risky with `f` (not really necessary for this situation)

note we can stack single character options together with a single `-`. This means

```
rm -rf examplew/
```

We use the `-r` because we're not just deleting `examplew/`. We're deleting `examplew/_toc.yml` `examplew/_configure.yml` `examplew/requirements.txt` `examplew/...` `.` `.` `.`

And that's why we need the recursion.

```
cd tiny-book/  
ls
```

```
_config.yml  logo.png          notebooks.ipynb  
_toc.yml     markdown-notebooks.md  references.bib  
intro.md     markdown.md        requirements.txt
```

What happens if I used the `-a` option? We list all files, including hidden ones. What about `-l`? It actually lists all files in a list one item at a line with some more details about each file on the same line. Details related to write/read permissions on the file, owner of file and what not...

And because we can stack single letter options together we can do

```
ls -la
```

```
total 47  
drwxr-xr-x 1 ayman 197609 0 Feb 24 20:43 ./  
drwxr-xr-x 1 ayman 197609 0 Mar 4 13:03 ../  
-rw-r--r-- 1 ayman 197609 1000 Feb 24 20:43 _config.yml  
-rw-r--r-- 1 ayman 197609 180 Feb 24 20:43 _toc.yml  
-rw-r--r-- 1 ayman 197609 431 Feb 24 20:43 intro.md  
-rw-r--r-- 1 ayman 197609 9854 Feb 24 20:43 logo.png  
-rw-r--r-- 1 ayman 197609 1787 Feb 24 20:43 markdown-notebooks.md  
-rw-r--r-- 1 ayman 197609 1890 Feb 24 20:43 markdown.md  
-rw-r--r-- 1 ayman 197609 3378 Feb 24 20:43 notebooks.ipynb  
-rw-r--r-- 1 ayman 197609 5524 Feb 24 20:43 references.bib  
-rw-r--r-- 1 ayman 197609 30 Feb 24 20:43 requirements.txt
```

### 11.5.1. Structure of a Jupyter book

We will explore the output by looking at the files

```
ls
```

```
_config.yml      logo.png          notebooks.ipynb  
_toc.yml        markdown-notebooks.md  references.bib  
intro.md        markdown.md        requirements.txt
```

A jupyter book has two required files (`_config.yml` and `_toc.yml`), some for content, and some helpers that are common but not required.

- [config defaults](#)
- [toc file formatting rules](#)
- the `*.md` files are content
- the `.bib` file is bibliography information
- The other files are optional, but common. [Requirements.txt](#) is the format for pip to install python dependencies. There are different standards in other languages for how to specify requirements and organize files that make up a [package](#)

the extention (.yml) is [yaml](#), which stands for “YAML Ain’t Markup Language”. It consists of key, value pairs and is deigned to be a human-friendly way to encode data for use in any programming language.

#### Note

the extention (.yml) is [yaml](#), which stands for “YAML Ain’t Markup Language”. It consists of key, value pairs and is deigned to be a human-friendly way to encode data for use in any programming language.

### 11.5.2. Dev tools mean we do not have to write bibliographies manually

bibliographies are generated with [bibtex](#) which takes structured information from the references in a [bibtex file](#) with help from [sphinxcontrib-bibtex](#)

For general reference, reference managers like [zotero](#) and [mendeley](#) can track all of your sources and output the references in bibtex format that you can use anywhere or sync with tools like MS Word or Google Docs.

```
cat _config.yml
```

```
# Book settings
# Learn more at https://jupyterbook.org/customize/config.html

title: My sample book
author: The Jupyter Book Community
logo: logo.png

# Force re-execution of notebooks on each build.
# See https://jupyterbook.org/content/execute.html
execute:
    execute_notebooks: force

# Define the name of the latex output file for PDF builds
latex:
    latex_documents:
        targetname: book.tex

# Add a bibtex file so that we can create citations
bibtex_bibfiles:
    - references.bib

# Information about where the book exists on the web
repository:
    url: https://github.com/executablebooks/jupyter-book # Online location of your book
    path_to_book: docs # Optional path to your book, relative to the repository root
    branch: master # Which branch of the repository should be used when creating links (optional)

# Add GitHub buttons to your book
# See https://jupyterbook.org/customize/config.html#add-a-link-to-your-repository
html:
    use_issues_button: true
    use_repository_button: true
```

Compared to what we saw on the example configure file in the documentation of Jupyter-book posted in the link earlier. This looks pretty similar, missing the explaining comments at the end of every line. But here since this is YOUR book, you can edit the name, the author, copyright and whatnot...

The configuration file, tells jupyter-book basic iformation about the book, it provides all of the settings that jupyterbook and sphinx need to render the content as whatever output format we want.

The table of contents file describe how to put the other files in order.

```
cat _toc.yml
```

```
:linenos:  
# Table of contents  
# Learn more at https://jupyterbook.org/customize/toc.html
```

The first two lines are comments, the pound sign `#` is a comment in bash and YAML. Here, the developers chose, in the template to put information about how to set up this file right in the template file. This is developers helping their users!

```
:linenos:  
:lineno-start: 4  
format: jb-book  
root: intro
```

The next two lines are key value pairs that tell the high level settings

```
:linenos:  
:lineno-start: 6  
chapters:  
- file: markdown  
- file: notebooks  
- file: markdown-notebooks
```

the end of the file shows a list of files that will be treated as chapters. This is also the syntax for a list in YAML, the list is named `chapters` and each item, starting with a `-` has a single key, `file`

### 🔔 Where have we seen a YAML list?

You can create a community badge that uses as the location of the “contribution” a link to the snipped to a set of lines in your main spring2025 repo that is a YAML list. This is valid as long as it is created by Friday 03/07.

The one last file tells us what dependencies we have

```
cat requirements.txt
```

```
jupyter-book  
matplotlib  
numpy
```

If your book generates with error messages run `pip install -r requirements.txt`

## 11.6. Building Documentation

We can transform from raw source to an output by **building** the book

```
jupyter-book build .
```

```

Running Jupyter-Book v1.0.3
Source Folder: C:\Users\ayman\Documents\systems\tiny-book
Config Path: C:\Users\ayman\Documents\systems\tiny-book\_config.yml
Output Path: C:\Users\ayman\Documents\systems\tiny-book\_build\html
Running Sphinx v7.4.7
loading translations [en]... done
making output directory... done
[etoc] Changing master_doc to 'intro'
checking bibtex cache... out of date
parsing bibtex file C:\Users\ayman\Documents\systems\tiny-book\references.bib... parsed 5 entries
myst v2.0.0: MdParserConfig(commonmark_only=False, gfm_only=False, enable_extensions={'linkify', 'dollarn'}
myst_nb v1.2.0: NbParserConfig(custom_formats={}, metadata_key='mystnb', cell_metadata_key='mystnb', kerr
Using jupyter-cache at: C:\Users\ayman\Documents\systems\tiny-book\_build\.jupyter_cache
sphinx-multitoc-numbering v0.1.3: Loaded
building [mo]: targets for 0 po files that are out of date
writing output...
building [html]: targets for 4 source files that are out of date
updating environment: [new config] 4 added, 0 changed, 0 removed
C:\Users\ayman\Documents\systems\tiny-book\markdown-notebooks.md: Executing notebook using local CWD [mys
C:\Users\ayman\AppData\Roaming\Python\Python313\site-packages\zmq\_future.py:687: RuntimeWarning: Proacto
    self._get_loop()
Assertion failed: Connection reset by peer [10054] (C:\Users\runneradmin\AppData\Local\Temp\tmpas55v6tq\t
C:\Users\ayman\Documents\systems\tiny-book\markdown-notebooks.md: Executed notebook in 2.60 seconds [mys
C:\Users\ayman\Documents\systems\tiny-book\notebooks.ipynb: Executing notebook using local CWD [mystnb]
C:\Users\ayman\Documents\systems\tiny-book\notebooks.ipynb: Executed notebook in 8.73 seconds [mystnb]

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
copying assets...
copying static files... done
copying extra files... done
copying assets: done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
copying images... [100%] C:/Users/ayman/Documents/systems/tiny-book/_build/jupyter_execute/ee80d0a558444f
dumping search index in English (code: en)... done
dumping object inventory... done
[etoc] missing index.html written as redirect to 'intro.html'
build succeeded.

The HTML pages are in _build\html.

=====
Finished generating HTML for book.
Your book's HTML pages are here:
    _build\html\
You can look at your book by opening this file in a browser:
    _build\html\index.html
Or paste this line directly into your browser bar:
    file:///C:/Users/ayman/Documents/systems/tiny-book/_build\html\index.html
=====
```

### Try it yourself

Which files created by the template are not included in the rendered output? How could you tell?

Now we can look at what it did

```
ls
```

```
_build/      intro.md          markdown.md    requirements.txt
_config.yml  logo.png         notebooks.ipynb
_toc.yml     markdown-notebooks.md references.bib
```

we note that this made a new folder called `_build`. we can look inside there.

```
ls _build/
```

```
html/  jupyter_execute/
```

and in the html folder:

```
ls _build/html/
```

```
_images/           index.html        objects.inv
_sources/          intro.html       search.html
_sphinx_design_static/ markdown-notebooks.html  searchindex.js
_static/           markdown.html
genindex.html      notebooks.html
```

We find the `index.html` file which is the interactive page we created with `jupyter-book`

We can also copy the path to the file and open it in our browser

we can change the size of a browser window or use the screen size settings in inspect mode to see that this site is responsive.

We didn't have to write any html and we got a responsive site!

If we look at the content of the book we just created one of the pages is labeled `Notebooks with MyST Markdown` If we compare this with what we saw in the `_toc.yml` file we remember the last The last file listed is `markdown-notebooks.md`

Let's check the content of the file

```
cat markdown-notebooks.md
```

```

---
jupytext:
  formats: md:myst
  text_representation:
    extension: .md
    format_name: myst
    format_version: 0.13
    jupytext_version: 1.11.5
kernelspec:
  display_name: Python 3
  language: python
  name: python3
---

# Notebooks with MyST Markdown

Jupyter Book also lets you write text-based notebooks using MyST Markdown.
See [the Notebooks with MyST Markdown documentation](https://jupyterbook.org/file-types/myst-notebooks.html).
This page shows off a notebook written in MyST Markdown.

## An example cell

With MyST Markdown, you can define code cells with a directive like so:

```\{code-cell}
print(2 + 2)
```

When your book is built, the contents of any `code-cell` blocks will be
executed with your default Jupyter kernel, and their outputs will be displayed
in-line with the rest of your content.

```\{seealso}
Jupyter Book uses [Jupytext](https://jupytext.readthedocs.io/en/latest/) to convert text-based files to
```

## Create a notebook with MyST Markdown

MyST Markdown notebooks are defined by two things:

1. YAML metadata that is needed to understand if / how it should convert text files to notebooks (including
   See the YAML at the top of this page for example.
2. The presence of `code-cell` directives, which will be executed with your book.

That's all that is needed to get started!

## Quickly add YAML metadata for MyST Notebooks

If you have a markdown file and you'd like to quickly add YAML metadata to it, so that Jupyter Book will

```\{jupyter-book myst init path/to/markdownfile.md
```

```

cool, we can see that the content of that page is derived directly from this file

If you wanted to change the styling with sphinx you can use built in [themes](#) which tell sphinx to put different files in the [\\_static](#) folder when it builds your site, but you don't have to change any of your content! If you like working on front end things (which is great! it's just not always the goal) you can even build [your own theme](#) that can work with sphinx.

## 11.7. Starting a git repo locally

We made this folder, but we have not used any git operations on it yet, it is actually not a git repo, which we *could* tell from the output above, but let's use git to inspect and get another hint.

We can try `git status`

```
git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```

This tells us the `.git` directory is missing from the current path and all parent directories. This is because we simply created this folder `tiny-book` it's not linked to git whatsoever yet

To make it a git repo we use `git init` with the path we want to initialize, which currently is `.`

```
git init .
```

```
Initialized empty Git repository in C:/Users/ayman/Documents/systems/tiny-book/.git/
```

Let's see what changed

```
ls -a
```

```
./      _config.yml  markdown-notebooks.md  requirements.txt  
../      _toc.yml    markdown.md  
.git/   intro.md    notebooks.ipynb  
_build/  logo.png    references.bib
```

we got a `.git/` folder

Let's check if we succeeded

```
git status
```

```
On branch master
```

```
No commits yet
```

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
_build/  
_config.yml  
_toc.yml  
intro.md  
logo.png  
markdown-notebooks.md  
markdown.md  
notebooks.ipynb  
references.bib  
requirements.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

Here we are faced with a social aspect of computing that is *also* a good reminder about how git actually works

we'll change our default branch to main

```
git branch -m main
```

and check in with git now

```
git status
```

```
On branch main
No commits yet
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    _build/
      _config.yml
      _toc.yml
      intro.md
      logo.png
      markdown-notebooks.md
      markdown.md
      notebooks.ipynb
      references.bib
      requirements.txt
nothing added to commit but untracked files present (use "git
```

## Retiring racist language

Historically the default branch was called master.

- [derived from a master/slave analogy](#) which is not even how git works, but was adopted terminology from other projects
- [GitHub no longer does](#)
- [the broader community is changing as well](#)
- [git allows you to make your default not be master](#)
- [literally the person who chose the names "master" and "origin" regrets that choice](#) the name main is a more accurate and not harmful term and the current convention.

this time it works and we see a two important things:

- there are no previous commits
- all of the files are untracked

## 11.8. Handling Built files

The built site files are completely redundant, content wise, to the original markdown files.

We do not want to keep track of changes for the built files since they are generated from the source files. It's redundant and makes it less clear where someone should update content.

How can we tell it not to track the build folder?

Git helps us with this with the .gitignore

```
echo "_build" >> .gitignore
```

```
git status
```

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
    .gitignore
    _config.yml
    _toc.yml
    intro.md
    logo.png
    markdown-notebooks.md
    markdown.md
    notebooks.ipynb
    references.bib
    requirements.txt

nothing added to commit but untracked files present (use "git add" to track)
```

only the `.gitingore` file is listed! But it took away the `_build/` folder just as we want.

and we will commit the template.

```
git add .
```

```
warning: in the working copy of '.gitignore', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of '_config.yml', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of '_toc.yml', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'intro.md', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'markdown-notebooks.md', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'markdown.md', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'notebooks.ipynb', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'references.bib', LF will be replaced by CRLF the next time Git touches it
warning: in the working copy of 'requirements.txt', LF will be replaced by CRLF the next time Git touches it
```

We have to add separately because the files are **untracked** we cannot use the `-a` option on commit

This is GitBash telling you that git is helping. Windows uses two characters for a new line `CR` (carriage return) and `LF` (line feed). Classic Mac Operating system used the `CR` character. Unix-like systems (including MacOS X) use only the `LF` character. If you try to open a file on Windows that has only `LF` characters, Windows will think it's all one line. To help you, since git knows people collaborate across file systems, when you check out files from the git database (`.git/` directory) git replaces `LF` characters with `CRLF` before updating your working directory.

When working on Windows, when you make a file locally, each new line will have `CRLF` in it. If your collaborator (or server, eg GitHub) runs not a unix or linux based operating system (it almost certainly does) these extra characters will make a mess and make the system interpret your code wrong. To help you out, git will automatically, for Windows users, convert `CRLF` to `LF` when it adds your work to the index (staging area). Then when you push, it's the compatible version.

[git documentation of the feature](#) this is added to a [new Windows page under resources](#)

and then we will commit with a simple message

```
git commit -m 'jupyter book template'
```

```
[main (root-commit) 1da3fa4] jupyter book template
 10 files changed, 342 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 _config.yml
 create mode 100644 _toc.yml
 create mode 100644 intro.md
 create mode 100644 logo.png
 create mode 100644 markdown-notebooks.md
 create mode 100644 markdown.md
 create mode 100644 notebooks.ipynb
 create mode 100644 references.bib
 create mode 100644 requirements.txt
```

## 11.9. How do I push a repo that I made locally to GitHub?

Right now, we do not have any remotes, so if we try to push it will fail. Next we will see how to fix that.

First let's confirm

```
git push
```

```
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository using
  git remote add <name> <url>
and then push using the remote name
  git push <name>
```

and it tells us how to fix it. This is why inspection is so powerful in developer tools, that is where we developers give one another hints.

Right now, we do not have any remotes

```
git remote
```

For today, we will create an empty github repo shared with me, by accepting the assignment linked in prismia or ask a TA/instructor if you are making up class.

More generally, you can [create a repo](#)

That default page for an empty repo if you do not initiate it with any files will give you the instructions for what remote to add.

Now we add the remote

```
git remote add origin https://github.com/compsys-progools/tiny-book-aymanbx-1.git
```

We can see what it did

```
git remote
```

```
origin
```

```
git status
```

```
On branch main
nothing to commit, working tree clean
```

Remember, It doesn't mention anything about whether we're ahead or behind with origin/main That's because we haven't linked the two branches together yet.

```
git push
```

```
fatal: The current branch main has no upstream branch.
To push the current branch and set the remote as upstream, use

  git push --set-upstream origin main

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

git gives us advise on what to do

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

```
To https://github.com/compsys-progtools/tiny-book-AymanBx
! [rejected]          main -> main (fetch first)
error: failed to push some refs to 'https://github.com/compsys-progtools/tiny-book-AymanBx'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

This is telling us that there is a commit on the GitHub repo that doesn't exist locally so the two repos are out of sync.

(We haven't made any changes ourselves on GitHub but when we create a repo on GitHub we get an **initial commit**)

```
git pull
```

```

remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (5/5), 1.70 KiB | 102.00 KiB/s, done.
From https://github.com/compsys-progtools/tiny-book-AymanBx
 * [new branch] feedback -> origin/feedback
 * [new branch] main -> origin/main
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

git branch --set-upstream-to=origin/<branch> main

```

This is telling us to that since the previous pull command didn't execute successfully we still haven't linked the local main with the online main and we need to `git pull -u origin main`

## 11.10. Prepare for Next Class

1. review the notes on [what is a commit](#). In gitdef.md on the branch for this issue, try to describe git in the four ways we described a commit. **the point here is to think about what you know for git and practice remembering it, not “get the right answer”; this is prepare work, we only check that it is complete, not correct**
2. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file below. We will come back to these notes in class later, but it is best to record over a time period instead of trying to remember at that time. Store your notes in your fall24 repo in idethoughts.md on a dedicated `ide_prep` branch. **This is prep for after a few weeks from now, not for October 8; keep this branch open until it is specifically asked for**

## 11.11. Experience Report Evidence

## 11.12. Badges

- |               |                 |
|---------------|-----------------|
| <b>Review</b> | <b>Practice</b> |
|---------------|-----------------|
1. Review the notes, [jupyterbook docs](#), and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book, by manually adding those files. **do not add the whole template to your repo**, make the content you have already so it can *build* into html. Set it so that the `_build` directory is not under version control.
  2. Add `docs-review.md` to your KWL repo and explain the most important things to know about documentation in your own words using other programming concepts you have learned so far. Include in a markdown (same as HTML `<!-- comment -->`) comment the list of CSC courses you have taken for context while we give you feedback.

## 11.13. Questions

## 12. What is git?

Last class we created a local repo, then created an empty repo on GitHub. Linked both repos and attempted to push local changes on the “empty” repo but weren’t successful

The reason for that is the way the new GitHub repo was created. When we used GitHub Classroom to fork my template tiny-book repo as your new empty repo GitHub made two commits. Making it not an empty repo.

```
cd Documents/systems/tiny-book
```

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

```
To https://github.com/compsys-progtools/tiny-book-AymanBx
! [rejected]      main -> main (non-fast-forward)
error: failed to push some refs to 'https://github.com/compsys-progtools/tiny-book-AymanBx'
hint: Updates were rejected because the tip of your current branch is behind
hint: its remote counterpart. If you want to integrate the remote changes,
hint: use 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Let's check it out

```
gh repo view --web
```

Let's compare that to what we have locally

```
git log
```

```
commit 1da3fa4d2b1b14e3a92358455c2320697af43867 (HEAD -> main)
Author: AymanBx <ayman_sandouk@uri.edu>
Date:   Tue Mar 4 13:42:31 2025 -0500

    jupyter book template
```

When we attempted to `pull` we weren't fully successful because we failed to link the two mains because the `push` command failed.

```
git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (5/5), 1.70 KiB | 102.00 KiB/s, done.
From https://github.com/compsys-progtools/tiny-book-AymanBx
 * [new branch] feedback -> origin/feedback
 * [new branch] main -> origin/main
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.
```

```
git pull <remote> <branch>
```

If you wish to set tracking information for this branch you can do so with:

```
git branch --set-upstream-to=origin/<branch> main
```

We should follow git instructions

```
git branch --set-upstream-to=origin/main main
```

```
branch 'main' set up to track 'origin/main'.
```

Now we should be able to pull successfully

```
git pull
```

```
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false # merge
hint:   git config pull.rebase true # rebase
hint:   git config pull.ff only     # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Git recognized the different commit history as in the repos as a conflict

To resolve this type of conflict (main had unrelated changes on it that are unknown by my branch) We **rebase**

**rebase** is updating my current branch with new commits that occurred on main (or any branch I want to rebase with)

```
git pull --rebase
```

```
Successfully rebased and updated refs/heads/main.
```

What did that do?

```
git log
```

```
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Mar 4 13:42:31 2025 -0500

jupyter book template

commit d781535217b324d9cb2ce6cb45ae565b54ee786f (origin/main)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date: Tue Oct 8 16:54:12 2024 +0000

    Setting up GitHub Classroom Feedback

commit 72bcbb8cbd2769d21aad3c23c8fbe477d0260ced (origin/feedback)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date: Tue Oct 8 16:54:12 2024 +0000

GitHub Classroom Feedback
```

Now we should be able to push

```
git push
```

```
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (12/12), 16.33 KiB | 8.17 MiB/s, done.
Total 12 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/compsys-progtools/tiny-book-aymanbx.git
   d781535..60bd457  main -> main
```

A couple of weeks ago, we learned about what a commit is and then we took a break from how git works, to talk more about unix philosophy and how developers communicate about code

Today we will learn what git is more formally.

### study tip

We will go in and out of topics at times, in order to provide what is called *spaced repetition*, repeating material or key concepts with breaks in between.

Using git correctly is a really important goal of this course because git is an opportunity for you to demonstrate a wide range of both practical and conceptual understanding.

So, I have elected to interleave other topics with git to give core git ideas some time to simmer and give you time to practice them before we build on them with more depth at git.

Also, we are both learning git and *using* git as a motivating example of other key important topics.

## 12.1. Why so much git?

Today, we are going to learn *what* git is and later we will learn more details of how it is implemented.

Remember we are spending so much time with git for two reasons:

1. it is an important developer tool
2. it demonstrates important conceptual ideas that occur in other areas of CS

[git book](#) is the official reference on git.

this includes other spoken languages as well if that is helpful for you.

## 12.2. git definition

From here, we have the full definition of git

[git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it.](#)

We do not start from that point, because these documents were written for target audience of working developers who are familiar with other, old version control systems and learning an *additional* one.

Have you used another version control system before?

Most of you, however, have probably not used another version control system.

Let's break down the definition

## 12.3. Git is a File system

Content-addressable filesystem means a key-value data store.

What are some examples of key-value pairs that you have seen in computer science broadly, and in this course specifically, so far?

- python dictionaries
- pointers (address,content)
- parameter, passed values
- yaml files

some examples of key-value pairs that you have seen in computer science broadly, and in this course specifically

- python dictionaries
- pointers (address,content)
- parameter, passed values
- yaml files

What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

## 12.4. Git is a Version Control System

In the before times

# "FINAL".doc



FINAL.doc!



FINAL\_rev.2.doc



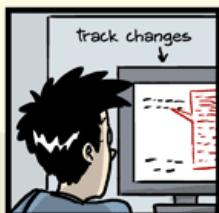
FINAL\_rev.6.COMMENTS.doc



FINAL\_rev.8.comments5.CORRECTIONS.doc



JORGE CHAM © 2012



FINAL\_rev.18.comments7.corrections9.MORE.30.doc



FINAL\_rev.22.comments49.corrections.10.#@\$%WHYDIDICOMETOGRAD SCHOOL????.doc

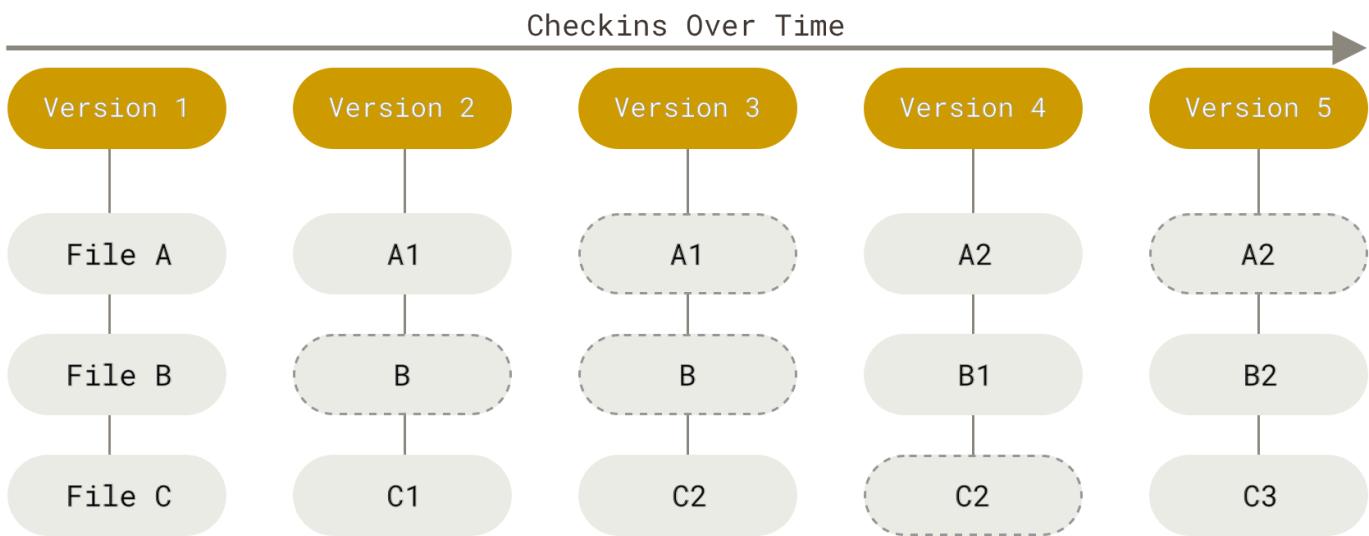


WWW.PHDCOMICS.COM

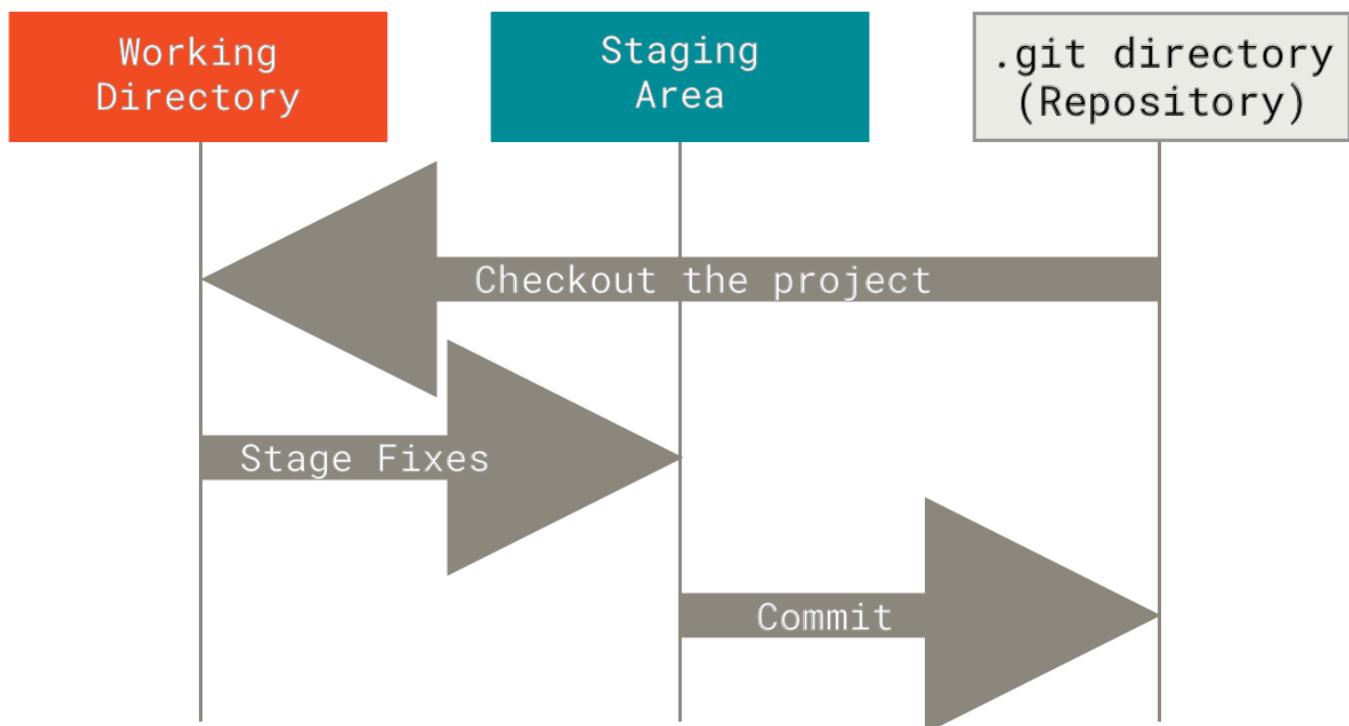
git stores **snapshots** of your work each time you commit.

What unit of git is how it represents a snapshot?

- [ ] branch
- [ ] blob
- [x] commit
- [ ] tag



it uses 3 stages:



These three stages are the in relation to your working directory, and potentially remotes.

So in broader context, the [git visual cheatsheet](#) is a more complete picture and has commands overlaid with the concept.

## 12.5. Git has two sets of commands

- Porcelain: the user friendly VCS
- Plumbing: the internal workings- a toolkit for a VCS

Which of the following commands are porcelain commands?

```
git commit  
git cat-file  
git add  
git status  
git hash-object
```

Which of the following commands are porcelain commands?

```
git commit  
git cat-file  
git add  
git status  
git hash-object
```

We have so far used git as a version control system. A version control system, in general, will have operations like commit, push, pull, clone. These may work differently under the hood or be called different things, but those are what something needs to have in order to keep track of different versions.

The plumbing commands reveal the way that git performs version control operations. This means, they implement the git file system operations for the git version control system.

You can think of the plumbing vs porcelain commands like public/private methods. As a user, you only need the public methods (porcelain commands) but those use the private ones to get things done (plumbing commands). We will use the plumbing commands over the next few classes to examine what git *really* does when we call the porcelain commands that we will typically use.

Example?

## 12.6. Git is distributed

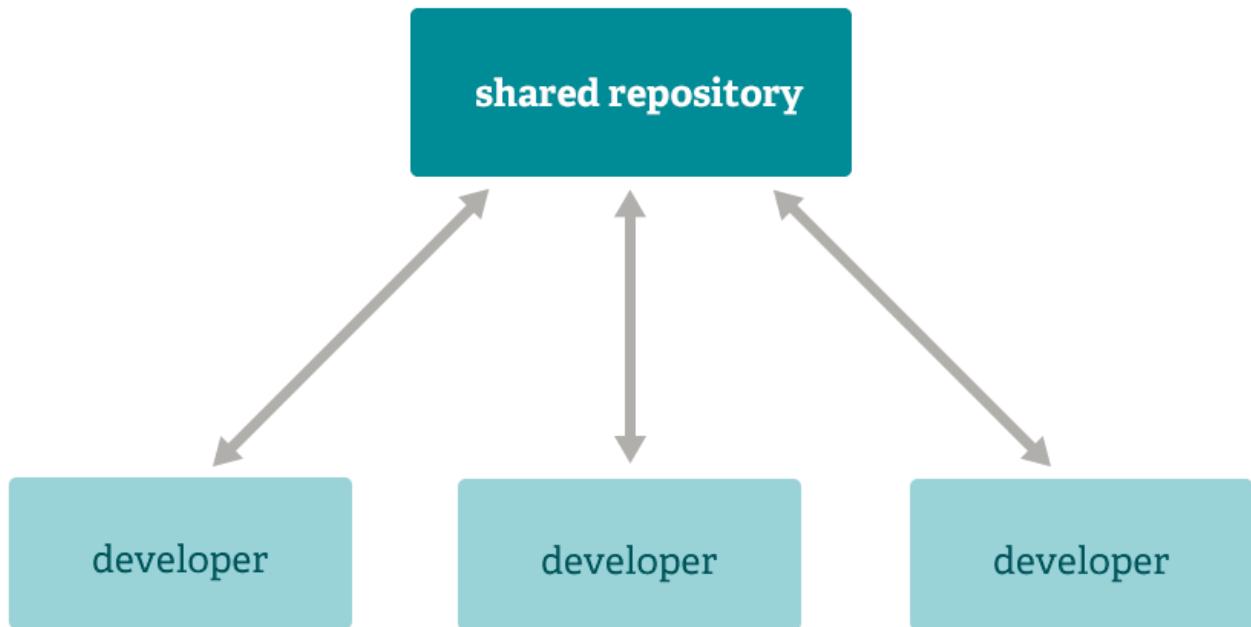
What does that mean?

Git runs locally. It can run in many places, and has commands to help sync across remotes, but git does not require one copy of the repository to be the “official” copy and the others to be subordinate. git just sees repositories.

For human reasons, we like to have one “official” copy and treat the others as other copies, but that is a social choice, not a technological requirement of git. Even though we will typically use it with an official copy and other copies, having a tool that does not care, makes the tool more flexible and allows us to create workflows, or networks of copies that have any relationship we want.

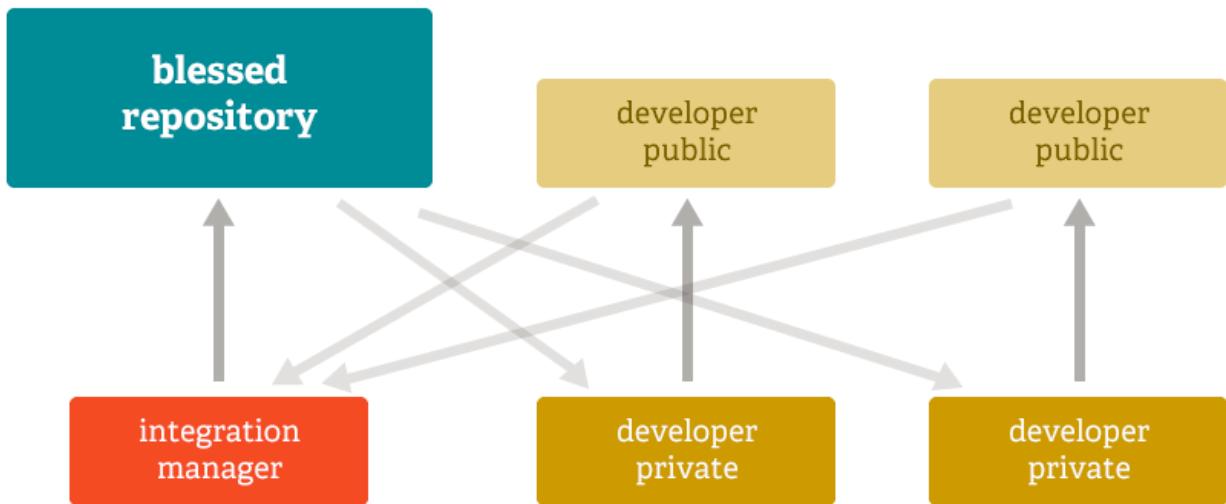
It's about the workflows, or the ways we socially *use* the tool.

### 12.6.1. Subversion Workflow

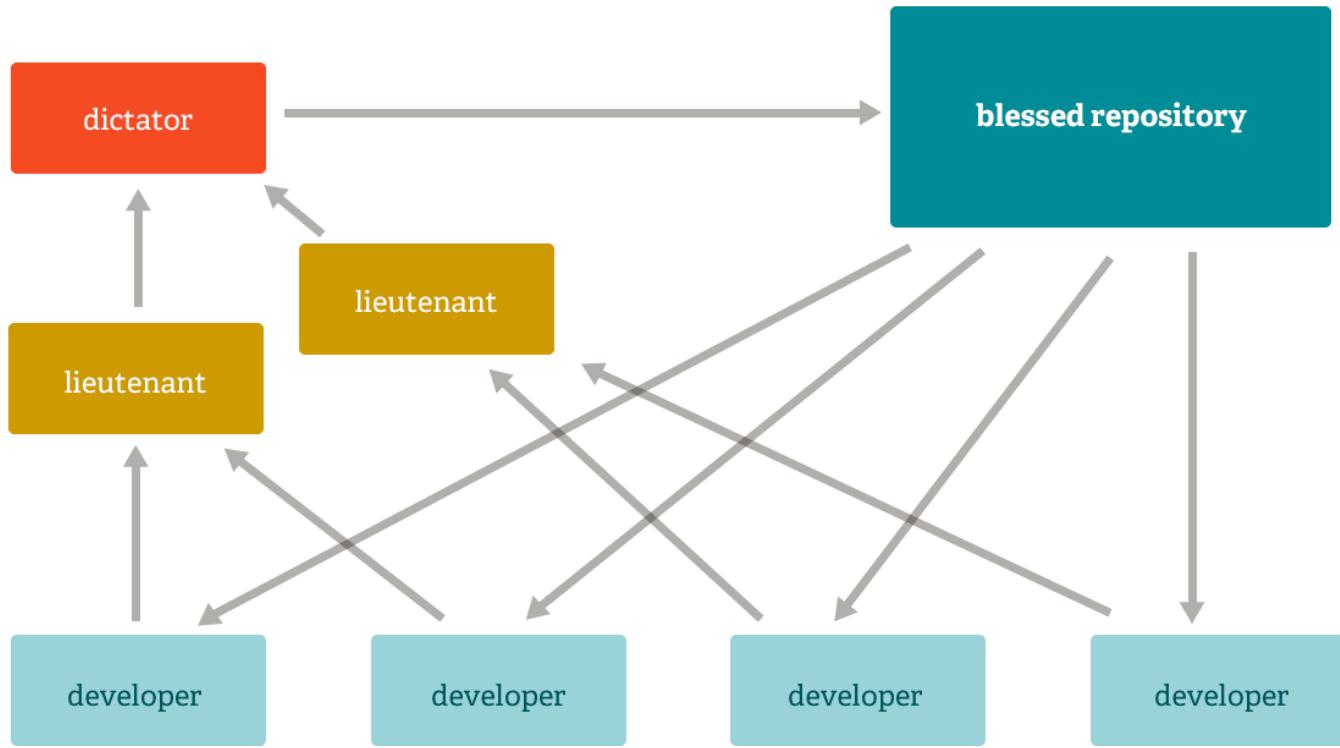


subversion is an older VCS

### 12.6.2. Centralized Manager



### 12.6.3. dictator and lieutenants



This is a variant of a multiple-repository workflow. It's generally used by huge projects with hundreds of collaborators; one famous example is the Linux kernel. Various integration managers are in charge of certain parts of the repository; they're called lieutenants. All the lieutenants have one integration manager known as the benevolent dictator. The benevolent dictator pushes from their directory to a reference repository from which all the collaborators need to pull.

## 12.7. How does git do all these things?

We can use the bash command `find` to search the file system. Note that this does not search the contents of the files, just the names.

```
find objects/ -type f
```

```
.git/objects/06/d56f40c838b64eb048a63e036125964a069a3a
.git/objects/0e/2e3b27f61b5908c4bb75a1ca680ee4053aa992
.git/objects/1d/a3fa4d2b1b14e3a92358455c2320697af43867
.git/objects/29/a422c19251aeaeb907175e9b3219a9bed6c616
.git/objects/2b/d9785b546aa1af7d6e41a48d33a0af811082dd
.git/objects/5f/534f8051f6a94d40e57e58242ef0113fae4fd1
.git/objects/6e/b15166db3ad944529be060af334deb2c022bbd
.git/objects/74/d5c7101ed8c8c1a6f87e31debd9445df1f0e71
.git/objects/78/3ec6aa5afe2f0a66087d01a112f543e1ed287e
.git/objects/7e/821e45db31376729c73f3616fb24db2b655a95
.git/objects/a0/57a320dc595f3f0e0d250c3af4a5653596914
.git/objects/d6/f9d92349c768da1863b412674f25cd27d23cfb
.git/objects/e3/5d8850c9688b1ce82711694692cc574a799396
.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects/f8/cdc73cb2be06824f521837366ec95b73d55ef8
.git/objects/fa/eea606145667f54d220a0c17ffe8d22db07146
.git/objects/fd/b7176c429a73d5335e127b27d530b8aaa07c7d
```

We searched for anything of the type `file` with the option `-type f`

This is a lot of files! It's more than we have in our working directory.

We can see that by looking at the working directory with `ls`

```
ls
```

```
_build/    _toc.yml  logo.png          markdown.md      references.bib  
_config.yml  intro.md  markdown-notebooks.md  notebooks.ipynb  requirements.txt
```

And remember, `build` is not being tracked. That means git knows nothing about it or about its content

This is a consequence of git taking snap shots and tracking both the actual contents of our working directory **and** our commit messages and other meta data about each commit.

## 12.8. Git Variables

the program `git` does not run continuously the entire time you are using it for a project. It runs quick commands each time you tell it to, it's goal is to manage files, so this makes sense. This also means that important information that `git` needs is also saved in files.

We can see the files that it has by listing the directory:

```
ls .git
```

```
COMMIT_EDITMSG  HEAD      description  index  logs/      refs/  
FETCH_HEAD      config    hooks/       info/  objects/
```

the files in all caps are like gits variables.

Lets look at the one called `HEAD` we have interacted with `HEAD` before when resolving merge conflicts.

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

`HEAD` is a pointer to the currently checked out branch. Do you remember where we see this pointer?

The other files with `HEAD` in their name are similarly pointers to other references, named corresponding to other things.

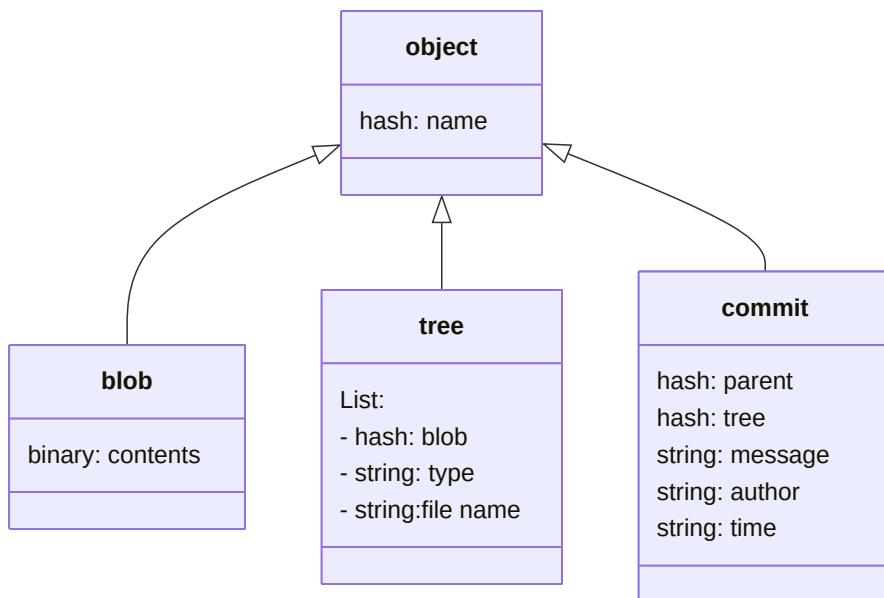
## 12.9. Git Objects

There are 3 types:

- blob objects: the content of your files (data)

- tree objects: stores file names and groups files together (organization)
- Commit Objects: stores information about the sha values of the snapshots

[mermaid.live](https://mermaid.live)



## 12.10. Examining git objects

Which of the following commands we have seen so far is a plumbing command?

- [ ] git commit
- [ ] git push
- [x] git cat-file
- [ ] git pull

```
git cat-file -t
```

Let's do so in the inclass repo

```
cd ../../gh-inclass-AymanBx/
```

Let's remember what has been done in this repo

```
git log
```

```
commit 99c73d4f2c0c87c3aef7a61366e56e3894a040a5 (HEAD -> organization, origin/organization)
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Feb 25 13:02:20 2025 -0500

    Organized all documentation files.

commit 93d4ae322683efa703dedbb9acb8431807a38af
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Tue Feb 25 12:53:36 2025 -0500

    Added overview of files in readme.md

commit 4ccfb5fe36073d15eaa4271d5f1355c0d9e1635b
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Thu Feb 20 13:13:59 2025 -0500

    added age to readme

commit 581ac5ef92ba5b4a647f30fc91a9d405e6900007 (origin/main, origin/HEAD, main)
Merge: 162a47b 7b56104
Author: Ayman Sandouk <111829133+AymanBx@users.noreply.github.com>
Date: Thu Feb 20 11:19:10 2025 -0500

    Merge pull request #3 from compsys-progtools/organizing_ac

Feb 20 in class activity

commit 7b561041f188ea7c2fa4d66d32957f639c574bd5 (origin/organizing_ac)
Author: AymanBx <ayman_sandouk@uri.edu>
Date: Thu Feb 20 11:01:56 2025 -0500

    add files for organizing activity
```

What branch are we on at the moment? We can tell from `git status` or from what we saw on `git log`. But all these commands are really getting their info from the HEAD pointer, and we now know how to do so ourselves:

```
cat .git/HEAD
```

```
ref: refs/heads/organization
```

Let's try to trace what we can find in that file

```
cat .git/refs/heads/organization
```

```
99c73d4f2c0c87c3aef7a61366e56e3894a040a5
```

We see a hash (made of 32 characters). A hash in git can be a representation of multiple things. One of its uses is that it is used as names of git objects that hold the different information of a git repository.

Let's confirm that we can find that file

```
ls .git/objects/
```

```
0f/ 23/ 39/ 4e/ 56/ 76/ 83/ 99/ a6/ ce/ e4/  
16/ 2d/ 3a/ 4f/ 58/ 7b/ 8b/ 9d/ a9/ d7/ f9/  
1e/ 33/ 3c/ 50/ 63/ 7f/ 92/ 9e/ ab/ d8/ info/  
1f/ 35/ 4c/ 54/ 65/ 80/ 93/ a2/ b0/ e0/ pack/
```

The first two characters of the hash can be found here as a directory `99/`

If we look at the contents of that folder

```
ls .git/objects/99
```

```
42c3c00dcde51d52a1c110413522043eca07cd  
c73d4f2c0c87c3aef7a61366e56e3894a040a5
```

We see two files, one of them is `c73d4f2c0c87c3aef7a61366e56e3894a040a5` which is the remainder of that hash

Can we read what's in that file?

```
cat .git/objects/99/c73d4f2c0c87c3aef7a61366e56e3894a040a5
```

```
x[mj1];xP[D]  
D h0[Ga`B] A("U> 2Ih!R:r8{'B'B')"E Sc[Z ={PPr}b[u|y=
```

Not very readable. But We've talked about how this is supposed to represent a status of the repository. It is supposed to represent a commit. We can confirm that by looking at the `git log` from earlier. The very first commit was represented by that same hash!

Let's see if there's another way to view the content of that file

```
git cat-file -t 99c73
```

```
commit
```

That's the type `-t`!

```
git cat-file -p 99c73
```

```
tree 7fc74e8f774bf2e6f25652bbab56ee45f21703c0  
parent 93d4ae322683efa703dedbb9acb8431807a38afd  
author AymanBx <ayman_sandouk@uri.edu> 1740506540 -0500  
committer AymanBx <ayman_sandouk@uri.edu> 1740506540 -0500
```

```
Organized all documentation files.
```

We see that the contents is detailed information about the last commit, we see some of this in the log. But here we find all the information. We had to use `git cat-file` specifically to view the content, because if we view it in its raw shape we get a

bunch of weird characters. That's because git compressed this information in binary to save disk space. Making the way git saves its information more space efficient.

Cool! Let's try one more thing. How can I find how the contents of a file is saved by git. Lets try to find a **blob** object. Let's check out the other file that was in the [99/](#) directory

```
git cat-file -t 9942c
```

```
commit
```

Ok, we found another commit. Let's view it

```
git cat-file -p 9942c
```

```
tree a2ea9badc19dc1d94af5957a8f6f128a3b316f49
parent 8bd4ea38fe31186b9e5d0c1e19c9aef748c6dce6
author Ayman Sandouk <111829133@users.github.com> 1739298243 -0500
committer GitHub <noreply@github.com> 1739298243 -0500
gpgsig -----BEGIN PGP SIGNATURE-----
wsFcBAABCAAQBQJnq5XECRC1aQ7uu5Uh1AAARUEQAB+Dnsi+XCvn+Vnb202qnpST
E8WlCMlKStERUEjHItTWHv+mY9YODKi66uBTDNvtJVZFr6DLJ2Xyr+Z5ee1T3nEE
FwA4xDjh3uegfJnleL2n/ebwlmQ+8xhtlfTvPg3rHHZw0B9kKQC1QbTiY0/FmMJ
yhu3KNS/7piXPjp7C8dYAI5sAyGUHBH/uGlVkvIkRmfuK8UgkLxQjlk3By5zHNLx
/4SLPB8NTf0qFMdag6DszlTn994FVKNPfbLikU433CeUw0yEUYUUZ0fil6f8nHnQ
4pjus9SIbhMWWUcVRVaiA85XW4EezqlaoMeZnBkgf/TekpzsDuPAIv7mRCa8Sbc
gk6NpHldfG9I0BYzFE7jSNa/JVT+em8IQq42pkXBB7Mu7ndKIovLAOSHLawquezd
thvw0f01TvaHkVhH9Cd7Noq9kjf/qKq9shJXRKUNNlhPM6FiNV9MwmaAGwgHvljg
HjrN/9ANxpakpw6tVGWkLCyCi+Ip9o22MSUY0YntI9cfKvEmeUYYv0zvMvJoIndZ
9krQr2zsfMHryG8U536+esEWQ17L9SFYZJljDakrRRY58lisxt4Un+D2zeMYNUIS
CY09UaZ5CIMHT0znKgWjaR+pQGKM4960QxhFN6s22B5+6InzQuWbetI0lcWXu/pa
fcqFgLJibxp0Zup3wABx
=tYyc
-----END PGP SIGNATURE-----
```

```
Update about.md
```

Looks like we updated the [about.md](#) file in this commit. Let's keep tracing. This commit is made of the tree

[a2ea9badc19dc1d94af5957a8f6f128a3b316f49](#). Let's check it out

```
git cat-file -p a2ea9
```

```
040000 tree 263fb9d22090e88edd2bf1847c24c3511de91b49    .github
100644 blob 920e41a47826309b0bca92645e1ce0c734830ed0  about.md
```

Cool, we found that blob that represents the [about.md](#) at that particular commit. Let's keep digging

```
git cat-file -t 920e4
```

```
blob
```

```
git cat-file -p 920e4
```

Second semster Masters

Expected graduation May 2026  
- I graduated highschool abroad

Hmm interesting. I remember [about.md](#) being different than that. wait,

```
cat about.md
```

Second semster Masters

Expected graduation May 2026  
- I got my BS in 2023  
- I graduated highschool abroad

Yeah, that's write, we had added another fun fact that time we got a merge conflict.

So that confirms it. We were just able to find an old version of [about.md](#) from an old commit. This is how git stores all the information that tracks all the changes that occurred.

One final thing, what happens if we try to view the contents of that hash without using `git cat-file`?

```
cat .git/objects/92/0e41a47826309b0bca92645e1ce0c734830ed0
```

```
xK[OR[ONM[KQ(N[-.I-R]MQ[\\"%])  
[E])%y@J##3.]0]8PEFfzFqrF~~[BbRQ~b  
[6]
```

Same as earlier, we just get a bunch of weird characters because of how git compresses the information into a binary.

## 12.11. Prepare for Next Class

1. Take a few minutes to think what you know about hashing and numbers. Create `hash_num_prep.md` with two sections:  
`## Hashing` with a few bullet points summarizing key points about hashing, and `## Numbers` with what types of number representations you know.
2. Review notes from [How do git branches work](#). Focus on resolving merge conflict in preparation to next lab.
3. Review notes from [What is a commit](#) & [What is git](#) (Both notes should be fixed by 3/10) to be fully prepared for this class after a nice break. Bring questions if you come up with any. You may qualify for a community badge if you post/contribute in a [discussion thread](#) related to the concepts mentioned in said classes if your ask meaningful questions that your classmates feel intrigued to discuss.

## 12.12. Experience Report Evidence

Append the contents of one of your trees or commits and one blob or tree inside of that first one to the bottom of your experience report.

## 12.13. Badges

Review

Practice

1. Read about different workflows in git and describe which one you prefer to work with and why in favorite\_git\_workflow.md in your kwl repo. Two good places to read from are [Git Book](#) and the [atlassian Docs](#)
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. Separate from what you added from the previous step. Add to your kwl table the following rows and fill them out. `git branches`, `merge conflicts`, `commits`. Try to include an older understanding about them in the `know` column and a newer understanding in the `learned` column
4. In commit\_contents.md, redirect the content of your most recent commit, its tree, and the contents of one blob. Edit the file or use `echo` to put markdown headings between the different objects. Add a title `# Complete Commit` to the file and at the bottom of the file add `## Reflection` subheading with some notes on how, if at all this excercise helps you understand what a commit is.

`git log` to see most recent commit `git cat-file` that hash `git cat-file` the tree has `git cat-file` hash of those files

## 12.14. Questions

# 13. How does git make a commit?

## 13.1. Important

If you missed a class and want to make up for the experience badge, you must complete the tasks found in "Experience Report Evidence" if found at the end of the notes

Today we will dig into how git really works. This will be a deep dive and provide a lot of details about how git creates a commit. It will reinforce important **concepts**, which is of practical use when fixing things give you some ideas about how you might fix things when things go wrong.

Later, we will build on this more on the practical side, but these **concepts** are very important for making sense of the more practical aspects of fixing things in git.

This deep dive in git is to help you build a correct, flexible understanding of git so that you can use it independently and efficiently. The plumbing commands do not need to be a part of your daily use of git, but they are the way that we can dig in and see what *actually* happens when git creates a commit.

**this is also to serve as an example method you could apply in understanding another complex system**

Inspecting a system's components is a really good way to understand it and correctly understanding it will impact your ability to ask good questions and even look up the right thing to do when you need to fix things.

Also, looking at the parts of git is a good way to reinforce specific design patterns that are common in CS in a practical way. This means that today we will also:

- review and practice with the bash commands we have seen so far
- see a practical example of hashing
- reinforce through examples what a pointer does

navigate to your github inclass repo

Recall: git stores important content in *files* that it uses like variables.

What is one example of a file in git that works like a pointer?

For example:

```
cat .git/HEAD
```

```
ref: refs/heads/organization
```

holds the current branch that git will compare the working directory with.

What do we find inside that file?

```
cat .git/refs/heads/organization
```

```
99c73d4f2c0c87c3aef7a61366e56e3894a040a5
```

We find a hash. If you remember this hash is a commit. We can confirm that by looking at the [git log](#)

```
cat .git/config
```

```
[core]
repositoryformatversion = 0
filemode = false
bare = false
logallrefupdates = true
symlinks = false
ignorecase = true
[remote "origin"]
url = https://github.com/compsys-progtools/gh-inclass-AymanBx
fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
remote = origin
merge = refs/heads/main
[branch "1-create-an-about-file"]
remote = origin
merge = refs/heads/1-create-an-about-file
[branch "organization"]
remote = origin
merge = refs/heads/organization
```

stores information about the different branches and remotes.

There are many:

```
ls .git
```

```
COMMIT_EDITMSG  ORIG_HEAD    hooks/  logs/   refs/
FETCH_HEAD      config       index    objects/ 
HEAD           description  info/    packed-refs
```

What file stores information for git and lives outside of the .git directory?

- [x] .gitignore
- [ ] HEAD
- [ ] config
- [ ] ORIG\_HEAD

.gitignore is a file in the working directory that contains a list of files and patterns to not track. We keep it in the working directory because we want it to be tracked by git like other files. So it can be pushed to GitHub also so files that are meant to be ignored can be ignored on all local copies of the repo

We can see it is a hidden file in the working directory with ls -a

```
ls -a
```

```
./          CONTRIBUTING.md    helper_functions.py
../          LICENSE.md        important_classes.py
.git/        README.md        my_secrets/
.github/     about.md         philosophy.md
.gitignore   abstract_base_class.py scratch.ipynb
.secrets     alternative_classes.py setup.py
API.md      docs/            tests/
```

and what it contains:

```
cat .gitignore
```

```
.secrets
my_secrets/*
```

## 13.2. Creating a repo from scratch

We will start in the top level course directory.

```
cd ..
ls
```

```
gh-inclass-AymanBx/ prep/ spring25-kwl-AymanBx/ tiny-book/
```

Yours should also have your kwl repo, gh inclass repo, course website clone, etc.

We can create an empty repo from scratch using `git init <path>`

Last time we used an existing directory like `git init .` because we were working in the directory that already existed

We can create an empty repo from scratch using `git init <path>`

Last time we used an existing directory like `git init .`, what does that mean about our path when we used the command?

- [ ] we had made a shortcut to the content we want to use for the repo already at `.`
- [x] the folder we wanted to use as the repo was the current working directory
- [ ] we wanted to make the repo name interactively after running the command

Today we will create a new directory called `test` and initialize it as a repo at the same time:

```
git init test
```

[we get this message again, see context from last week](#)

We can see what it did by first looking at the working directory

```
ls
```

```
gh-inclass-AymanBx/ prep/ spring25-kwl-AymanBx/
test/ tiny-book/
```

it made a new folder named as we said

and we can go into that directory

```
cd test/
```

and then rename the branch

```
git branch -m main
```

To clarify we will look at the status

```
git status
```

Notice that there are no commits, and no origin.

```
On branch main
No commits yet
nothing to commit (create/copy files and use "git add" to track)
```

```
ls .git
```

|        |             |         |      |
|--------|-------------|---------|------|
| HEAD   | description | info    | refs |
| config | hooks       | objects |      |

we can see the basic requirements of an empty repo here.

What does this tell us about the other files we saw in the gh inclass .git directory?

Additional variables that are created along the use of the repo

What command lets us search the file system, meaning the names of files for matches to a pattern?

- [ ] search
- [ ] grep
- [ ] awk
- [x] find
- [ ] wc
- [ ] fsearch

### 13.3. Searching the file system

We can use the bash command `find` to search the file system note that this does not search the **contents** of the files, just the names.

```
find .git/objects/
```

```
.git/objects/
.git/objects//pack
.git/objects//info
```

we have a few items in that directory and the directory itself.

We can limit by type, to only files with the `-type` option set to `f`

```
find .git/objects/ -type f
```

And we have no results. We have no objects yet. Because this is an empty repo

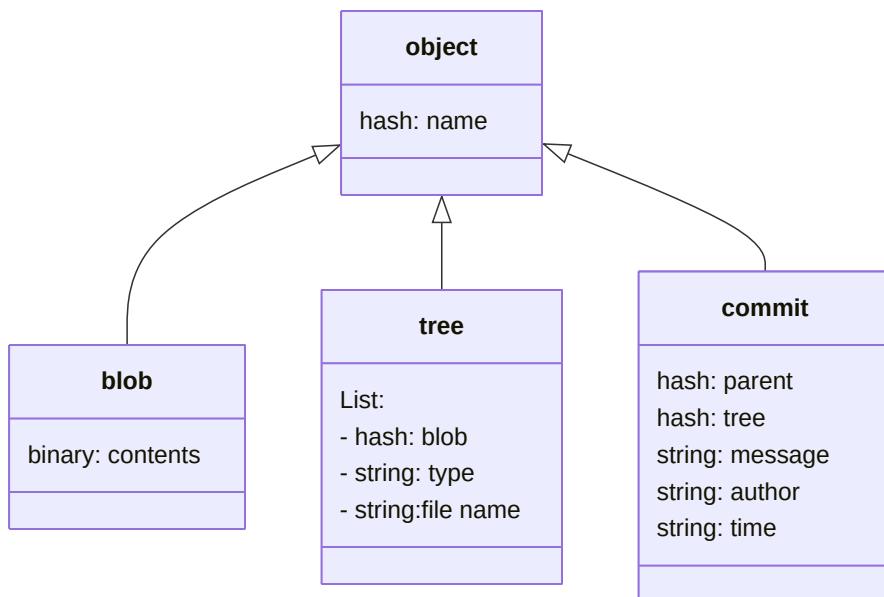
Which of the following is not a type of git `object` ?

- [ ] commit
- [x] branch
- [ ] tree
- [ ] blob

## 13.4. Git Objects

Remember our 3 types of objects

- blob objects: the content of your files (data)
- tree objects: stores file names and groups files together (organization)
- Commit Objects: stores information about the sha values of the snapshots



### 13.4.1. How to create an object

All git objects are files stored with the name that is the hash of the content in the file

Remember git is a content-addressable file system... so it uses key- value pairs.

Let's create our first git object. git uses hashes as the key. We give the hashing function some content, it applies the algorithm and returns us the hash as the reference to that object. We can also write to our .git directory with this.

The `git hash-object` command works on files, but we do not have any files yet. We can create a file, but we do not have to. Remememer, **everything** is a file.

What file do we always have access to to use for temporary use?

- [ ] [tmp.md](#)
- [x] stdout
- [ ] terminal
- [ ] temporary

What bash command sends its input to stdout?

- [ ] print
- [ ] show
- [x] echo

- [ ] tmpstore
- [ ] stdstore

### 13.4.2. Note

As a fun exercise, for the remainder of the class try to follow along with exactly the same text that I use in commands. Same capitalization and spacing. It isn't a big deal if you didn't

When we use things like `echo` it writes to the stdout file.

```
echo "test content"
```

```
test content
```

which shows on our terminal.

What allows us to chain commands together using output from one as input to the next?

- [ ] a link (&)
- [ ] a redirect (>>)
- [x] a pipe (|)

We can use a pipe to connect the stdout of one command to the stdin of the next.

Pipes are an important part of computer science too. We're seeing them in context of real uses, and we will keep seeing them. Pipes can connect the std out of one command to the std in of the next.

```
echo "test content" | git hash-object --stdin
```

We can break down this command:

- `git hash-object` would take the content you handed to it and merely return the unique key
- `--stdin` option tells `git hash-object` to get the content to be processed from `stdin` instead of a file
- the `|` is called a pipe (what we saw before was a redirect) it pipes a *process output* into the next command
- `echo` would write to `stdout`, with the pipe it passes that to `stdin` of the `git-hash`

We get back the hash:

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Notice how we all got the same hash. This is because the hash is generated by a hashing algorithm that uses the content we provided to generate the hash

Let's check if it wrote to the directory.

```
find .git/objects/ -type f
```

Now let's run it again with a slight modification. `-w` option tells the command to also write that object to the database

```
echo "test content" | git hash-object -w --stdin
```

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we can check if it wrote to the directory.

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we see a file that it was supposed to have!

### 13.4.3. Viewing git objects

We can try with `cat`

```
cat .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

Notice that the first two characters from the hash were used as a folder name

```
xK??OR04f(I-.QH??+I?+?K?
```

This is binary output that we cannot understand. Fortunately, git provides a utility. We can use `cat-file` to use the object by referencing at least 4 characters that are unique from the full hash, not the file name. (`7046` will not work, but `d670` will). Because the hash starts with at the name of the folder

`cat-file` requires an option `-t` is for type

```
git cat-file -t d670
```

```
blob
```

we see that it is a blob.

Then we can do it with the `-p` option for pretty print instead to see the content

```
git cat-file -p d670
```

```
test content
```

where we see the content we put in to the hashing function

This is the content that we put in, as expected.

#### 13.4.4. Hashing a file

What allows us to send the outputs of a command to a different file instead of stdout?

- [ ] a link (&)
- [x] a redirect (>>)
- [ ] a pipe ()

let's create a file.

```
echo "version 1" >test.txt
```

and store it, by hashing it

```
git hash-object -w test.txt
```

```
83baae61804e65cc73a7201a7252750c76066a30
```

we can look at what we have.

```
find .git/objects/ -type f
```

we see two objects as expected

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Now this is the status of our repo.

|                  |             |
|------------------|-------------|
| d67046           | 83baae      |
| + "test content" | + version 1 |
| +(blob)          | +(blob)     |

We can check the type of files with `git cat-file` and `-t`

```
git cat-file -t 83baa
```

```
git cat-file -t d670
```

```
blob
```

it is a blob object as expected

Notice, however, that we only have one file in the working directory.

```
ls
```

```
test.txt
```

it is the one test.txt, the first blob we made had no file in the working directory associated to it.

the workign directory and the git repo are not strictly the same thing, and can be different like this. Mostly they will stay in a closer relationship that we currently have unless we use plumbing commands, but this is good to build a solid understanding of how the `.git` directory relates to your working directory.

```
git status
```

```
On branch main
```

```
No commits yet
```

```
Untracked files:  
(use "git add <file>..." to include in what will be committed)  
  test.txt
```

```
nothing added to commit but untracked files present (use "git add" to track)
```

So far, even though we have hashed the object, git still thinks the file is untracked, because it is not in a tree and there are no commits that point to that part of the tree.

When working with porcelain commands, what do we do before we can make a commit?

- [ ] hash the content
- [ ] push to remote
- [x] stage the content (put it in the index)

What porcelain command puts content in the index?

- [ ] commit
- [ ] stash
- [x] add
- [ ] pull

This is still what our repo looks like

|                 |            |
|-----------------|------------|
| d67046          | 83baae     |
| +"test content" | +version 1 |
| +(blob)         | +(blob)    |

## 13.5. Updating the Index

Now, we can add our file as it is to the index.

```
git update-index --add --cacheinfo 100644 \
83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

the ↵ lets us wrap onto a second line.

- this the plumbing command `git update-index` updates or in this case creates an index which is the staging area of our repository
- the `--add` option is because the file doesn't yet exist in our staging area (we don't even have a staging area set up yet)
- `--cacheinfo` because the file we're adding isn't in your directory but is in the database.
- in this case, we're specifying a mode of 100644, which means it's a normal file.
- then the hash object we want to add to the index (the content) in our case, we want the hash of the first version of the file, not the most recent one.
- finally the file name of that content

Again, we check in with status

```
git status
```

```
On branch main
No commits yet

Changes to be committed:
(use "git rm --cached <file>..." to unstage)
  new file:   test.txt
```

We have the files staged as expected

Now the file is staged.

Let's edit it further.

```
echo "version 2" >> test.txt
```

Remember >> allows us to append

We can look at the content to ensure it as expected

```
cat test.txt
```

```
version 1  
version 2
```

So the file has two lines

Now check status again.

```
git status
```

```
On branch main  
No commits yet  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file:   test.txt  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
  modified:   test.txt
```

We added the first version of the file to the staging area, so that version is ready to commit but we have changed the version in our working directory relative to the version from the hash object that we put in the staging area so we *also* have changes not staged.

We can hash and store this version too.

```
git hash-object -w test.txt
```

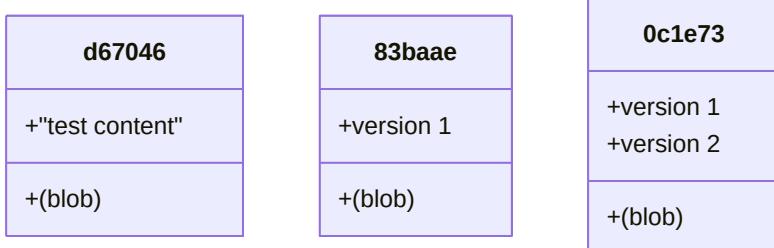
```
0c1e7391ca4e59584f8b773ecdbbb9467eba1547
```

We can then look again at our list of objects.

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547  
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

So now our repo has 3 items, all blobs:



```
git status
```

```
On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   test.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:  test.txt
```

hashing the object does not impact the index, which is what git status uses

### 13.5.1. Preparing to Commit

When we work with porcelain commands, we use add then commit. We have staged the file, which we know is what happens when we add. What else has to happen to make a commit.

We know that commits are comprised of:

- a message
- author and times stamp info
- a pointer to a tree
- a pointer to the parent (except the first commit)

We do not have any of these items yet.

Let's make a tree next.

Now we can write a tree from the index,

```
git write-tree
```

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

and we get a hash

Notice how we didn't have to pass any arguments to that command. It uses what is in the index to build the tree

Lets examine the tree, first check the type

```
git cat-file -t d832
```

```
tree
```

it is as expected

and now we can look at its contents

```
git cat-file -p d832
```

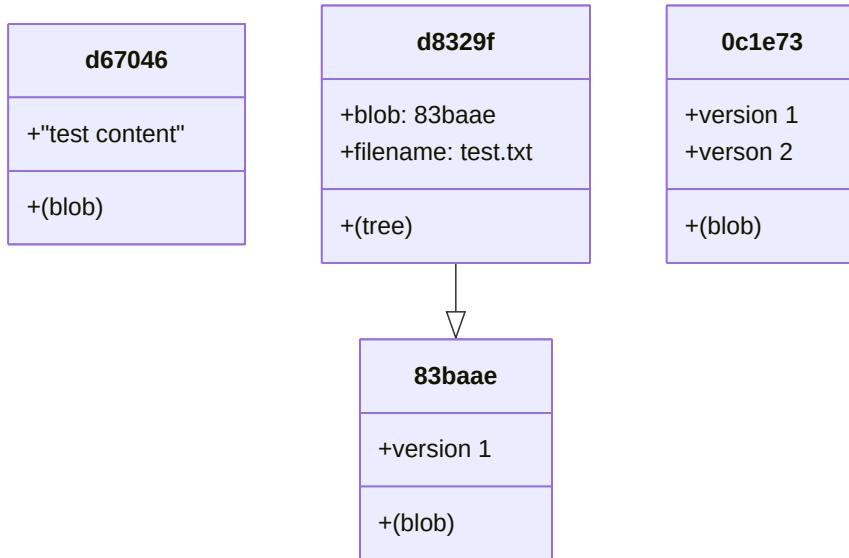
```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

Let's look at the objects now

```
find .git/objects/ -type f
```

```
.git/objects/0c/1e7391ca4e59584f8b773ecdbbb9467eba1547  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  
.git/objects/d8/329fc1cc938780ffd9f94e0d364e0ea74f579
```

Now this is the status of our repo:



Again, we will check in with git via `git status`

```
git status
```

```
On branch main  
No commits yet  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file: test.txt  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
  modified: test.txt
```

Nothing has changed, making the tree does not yet make the commit

This only keeps track of the objects, there are also still the HEAD that we have not dealt with and the index.

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

```
ls .git/refs/heads/
```

heads folder is still empty. That's an indicator that as far as git knows. No commits to any branch have been made yet.

### 13.5.2. Creating a commit manually

We can echo a commit message through a pipe into the commit-tree plumbing function to commit a particular hashed object.

```
echo "first commit"
```

```
first commit
```

the `git commit-tree` command requires a message via stdin and the tree hash. We will use stdin and a pipe for the message

```
echo "first commit" | git commit-tree d832
```

```
62467cd0c316a93201e98685a9d9f9beed8e6e27
```

and we get back a hash. But notice that this hash is unique for each of us. Because the commit has information about the time stamp and our user.

Now we check the final list of objects that we have for today

```
find .git/objects/ -type f
```

```
.git/objects/0c/1e7391ca4e59584f8b773ecdbbb9467eba1547  
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30  
.git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  
.git/objects/d8/329fc1cc938780ffdd9f94e0d364e0ea74f579  
.git/objects/62/467cd0c316a93201e98685a9d9f9beed8e6e27
```

### ! Important

Check that you also hav 5 objects and 4 of them should match mine, the one that would not match is the [188a75e](#) one it should be different because of the different timestamp and commit author.

We can also look at its type

```
git cat-file -t 62467
```

```
commit
```

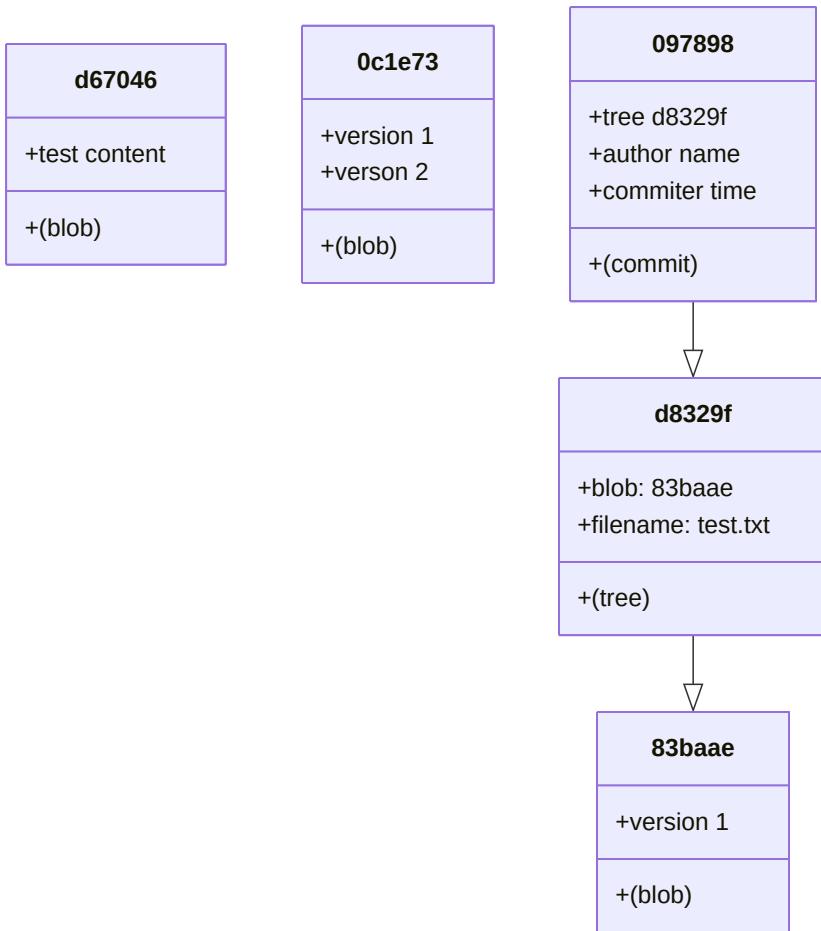
and we can look at the content

```
git cat-file -p 62467
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
author AymanBx <ayman_sandouk@uri.edu> 1742319374 -0400  
committer AymanBx <ayman_sandouk@uri.edu> 1742319374 -0400
```

```
first commit
```

Visually, this is what our repo looks like:



## 13.6. What does git status do?

*compares the working directory to the current state of the active branch*

- we can see the working directory with: `ls`
- we can see the active branch in the `HEAD` file
- what is its status?

```
git status
```

```

On branch main
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
    new file:   test.txt

Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test.txt

```

we see it is “on main” this is because we set the branch to main , but since we have not added the commit to main git still doesn’t know we made a commit. Notice that when we use the porcelain command for commit, it does this automatically; the porcelain commands do many things.

Notice, git says we have no commits yet even though we have written a commit.

In our case because we made the commit manually, we did not update the branch.

This is because the main branch does not *point* to any commit.

We can verify by looking at the `HEAD` file

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

and then viewing that file

```
cat .git/refs/heads/main
```

```
cat: .git/refs/heads/main: No such file or directory
```

which does not even exist!

```
ls .git/refs/heads
```

nothing exists there yet!

But we can see the `heads` folder exists

```
ls .git/refs
```

```
heads    tags
```

we can see the objects though:

```
find .git/objects/ -type f
```

```
git cat-file -t 62467
```

```
commit
```

```
git status
```

```
On branch main  
No commits yet  
Changes to be committed:  
(use "git rm --cached <file>..." to unstage)  
  new file: test.txt  
  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
  modified: test.txt
```

This is because git status works off the HEAD file and we have not updated that or set our branch to point to our commit yet.

## 13.7. Git References

We can make that file manually

```
echo 188a75ef66b6a85be0ab68d8575ec27808881dfc > .git/refs/heads/main
```

```
git status
```

```
On branch main  
Changes not staged for commit:  
(use "git add <file>..." to update what will be committed)  
(use "git restore <file>..." to discard changes in working directory)  
  modified: test.txt  
  
no changes added to commit (use "git add" and/or "git commit -a")
```

We can see that indeed we have one object that is a commit

```
git cat-file -p 62467
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579  
author AymanBx <ayman_sandouk@uri.edu> 1741922840 -0400  
committer AymanBx <ayman_sandouk@uri.edu> 1741922840 -0400  
  
first commit
```

```
git cat-file -p d8329
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

```
git cat-file -p 83baa
```

So we now have HEAD-> main and main -> our commit -> tree -> blob.

## 13.8. Experience Report Evidence

### Important

You need to have a test repo that matches this for class on tuesday.

Generate your evidence with the following in your test repo

```
find .git/objects/ -type f > testobj.md
```

then append the contents of your commit object to that file.

Move the `testobj.md` to your kwl repo in the experiences folder.

## 13.9. Prepare for Next Class

- [ ] Review the GitHub Action [file from last lab](#) and make note of what if any syntax in there is unfamiliar. (note that link will not work on the rendered website, but will work on badge issues)
- [ ] Use quote reply or edit to see how I made a relative path to a location within the repo in this issue. (to see another application of paths)
- [ ] Check out the [github action marketplace](#) to see other actions that are available and try to get a casual level of understanding of the types of things that people use actions for.

## 13.10. Experience Report Evidence

## 13.11. Badges

### Review

### Practice

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plumbing and porcelain. The table should have two columns, one for each type of command (plumbing and porcelain). Each row should have one git porcelain command and at least one of the corresponding git plumbing command(s). Include two rows: `add` and `commit`.

## 13.12. Questions

## 14. How can I automate things with bash?

Review the GitHub Action file that was used last lab as well as action workflow files in your kwl repo. Make note of what if any syntax in there is unfamiliar.

Today our goal is to learn more about how those things work.

So far we have used bash commands to navigate our file system as a way to learn about the file system itself. To do this we used commands like:

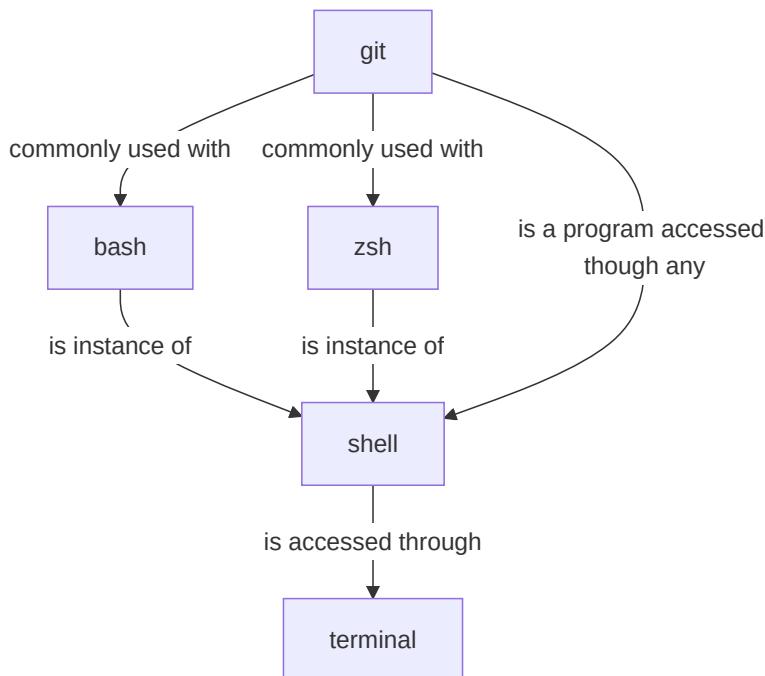
- `pwd`
- `cd`
- `ls`
- `mv`
- `cp`
- `rm`
- `find`

*do you remember what each of those does?, no need to respond, just think through it for yourself*

Bash is a unix shell for the GNU operating system and it has been adopted in other contexts as well. It is the default shell on Linux Ubuntu as well many others. This is why we learn about it here. GNU is a recursive acronym that stands for "GNU's Not Unix". It is a Unix-like completely free system

Use [mermaid](#) or the prismia drawing tool to visualize the relationship between the following terms:

- bash
- shell
- terminal
- git



A Unix shell is referring to both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined.

Read the official definition of `bash` and a shell in [the bash manual](#)

shell

programming language

command interpreter

## 14.1. Inspecting an example script

Today we will start by inspecting the github action file, first we'll navigate to the folder.

```
cd OneDrive\Desktop\URI\CSC311\collab-activity
```

Next we'll look at which files we have

```
ls .github/workflows/
```

```
generate_problem.yml
```

Let's look at the file

```
cat .github/workflows/generate_problem.yml
```

```

name: Generate Problem

on:
  workflow_dispatch:
inputs:
  problem_number:
    description: "Enter problem number (1, 2, or 3)"
    required: true
    type: choice
    options:
      - "1"
      - "2"
      - "3"
  programming_language:
    description: "Enter programming language (Python or C++)"
    required: true
    type: choice
    options:
      - "Python"
      - "Cpp"

jobs:
  setup_problem:
    runs-on: ubuntu-latest

steps:
  - name: Checkout repository
    uses: actions/checkout@v4

  - name: Debug - List files before renaming
    run: ls -lah

  - name: Reserve the old Readme file
    run: |
      mv README.md about.md
      ls -lah

  - name: Copy problem files to the main folder
    run: |
      PROBLEM_DIR=".github/Problems/Problem${{ github.event.inputs.problem_number }}/${{ github.event.inputs.problem_number }}"
      if [ -d "$PROBLEM_DIR" ]; then
        cp -r "$PROBLEM_DIR"/.* "."
        echo "Problem file copied successfully."
      else
        echo "Error: Problem file does not exist!" && exit 1
      fi

  - name: Commit and push changes
    run: |
      git config user.name "github-actions"
      git config user.email "github-actions@github.com"
      git add .
      git commit -m "Add new problem files"
      git push

  - name: Create issue for the problem
    env:
      GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
    run: |
      ISSUE_FILE=".github/Problems/Problem${{ github.event.inputs.problem_number }}/issue.md"
      gh issue create \
        --title "Problem ${{ github.event.inputs.problem_number }}" \
        --body-file "$ISSUE_FILE"

```

We notice some lines that we are very familiar with. We see lines that `mv` and `cp` files. We also see a set of actions that will `add`, `commit` and `push`. So we see how we automate commands that we have used normally before on a terminal within this script to make it execute all these commands with the action run.

What features do you expect if something is a programming language?

## 14.2. Variables in Bash

From the action files what do you think the syntax for a variable in bash is? Give an example by creating a variable called `MYVAR` with the value `my_val`

We can create variables

```
NAME='Ayman'
```

Do you see a variable declaration in the workflow script?

```
ISSUE_FILE=".github/Problems/Problem${{ github.event.inputs.problem_number }}/issue.md"
```

notice that there are **no spaces** around the `=`. spaces in bash separate commands and options, so they cannot be in variable declarations.

what is the syntax for using a variable? *try them out and find which is correct, or use code inspection on your actionfile*

- [ ] NAME>
- [x] \$NAME
- [ ] NAME\$
- [ ] NAME

echo `NAME` with a `$` before the variable name.

```
echo $NAME
```

Ayman

A common mistake is to put a space around the `=` sign, this is actually considered **good style** in many other languages.

```
NAME = 'Ayman'
```

```
-bash: NAME: command not found
```

In bash, however, this creates an error. When there is a space after `NAME`, `bash` tried to interpret `NAME` as a bash command, but then it does not find it, so it gives an error.

Removing the space works again:

```
NAME='Ayman Sandouk'
```

This variable is local, in memory, to the current terminal window, so if we open a separate window and try `echo $NAME` it will not work. We can also see that it does not create any file changes.

```
ls
```

```
README.md calculator.cpp calculator.py calculator.rs
```

We don't see anything referring to the `$NAME` variable

We can, however use the variable at different working directories in the same terminal. So if we move

```
cd ../Spring2025/  
echo $NAME
```

```
Ayman Sandouk
```

echo still works.

The `$` is essential syntax for recalling variables in bash. If we forget it, it treats it as a literal

```
echo NAME
```

```
NAME
```

so we get the variable **name** out instead of the variable **value**

### 14.3. Environment Variables

Environment variables are global among all terminal windows.

A common one is the `PATH`

```
echo $PATH
```

```
/c/Users/ayman/bin:/mingw64/bin:/usr/local/bin:/usr/bin:/bin:/mingw64/bin:/usr/bin:/c/Users/ayman/bin:/c/
```

mine shows multiple paths that will have packages installed in them, yours may be different

You can store environment variables to be set each time you start a terminal in your profile.

- On MacOS this file is: `~/.bash_profile`
- on linux it is `~/.bashrc`

- For windows users you should also have `~/.bashrc`
  - If you don't have it you can create it with `touch` and edit it to add any global variables you want

```
cat ~/.bashrc
```

```

# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# If not running interactively, don't do anything
case $- in
    *i*) ;;
    *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "##" used in a pathname expansion context will
# match all files and zero or more directories and subdirectories.
#shopt -s globstar

# make less more friendly for non-text input files, see lesspipe(1)
[ -x /usr/bin/lesspipe ] && eval "$(SHELL=/bin/sh lesspipe)"

# set variable identifying the chroot you work in (used in the prompt below)
if [ -z "${debian_chroot:-}" ] && [ -r /etc/debian_chroot ]; then
    debian_chroot=$(cat /etc/debian_chroot)
fi

# set a fancy prompt (non-color, unless we know we "want" color)
case "$TERM" in
    xterm-color|*-256color) color_prompt=yes;;
esac

# uncomment for a colored prompt, if the terminal has the capability; turned
# off by default to not distract the user: the focus in a terminal window
# should be on the output of commands, not on the prompt
#force_color_prompt=yes

if [ -n "$force_color_prompt" ]; then
    if [ -x /usr/bin/tput ] && tput setaf 1 >&/dev/null; then
        # We have color support; assume it's compliant with Ecma-48
        # (ISO/IEC-6429). (Lack of such support is extremely rare, and such
        # a case would tend to support setf rather than setaf.)
        color_prompt=yes
    else
        color_prompt=
    fi
fi

if [ "$color_prompt" = yes ]; then
    PS1='${debian_chroot:+($debian_chroot)}[\u033[01;32m]\u0@\h[\u033[00m]:[\u033[01;34m]\w[\u033[00m]'
else
    PS1='${debian_chroot:+($debian_chroot)}\u0@\h:\w$ '
fi
unset color_prompt force_color_prompt

# If this is an xterm set the title to user@host:dir
case "$TERM" in
xterm*|rxvt*)
    PS1="\[\e]0;${debian_chroot:+($debian_chroot)}\u@\h: \w\a\]$PS1"

```

```

        ;;
*)
    ;;
esac

# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
    test -r ~/.dircolors && eval "$(dircolors -b ~/.dircolors)" || eval "$(dircolors -b)"
    alias ls='ls --color=auto'
    #alias dir='dir --color=auto'
    #alias vdir='vdir --color=auto'

    alias grep='grep --color=auto'
    alias fgrep='fgrep --color=auto'
    alias egrep='egrep --color=auto'
fi

# colored GCC warnings and errors
#export GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:locus=01:quote=01'

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Add an "alert" alias for long running commands. Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$( [ $? = 0 ] && echo terminal || echo error)" "$(history|tail

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -q posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
        . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
        . /etc/bash_completion
    fi
fi
. "$HOME/.cargo/env"

```

A common one you might want to set is:

- **PS1** the primary prompt line. Content you can use is [documented in the gnu docs](#)
- **alias** lets you set another name for a program, for example I use **pip** to call **pip3**

```

nano ~/.bashrc
cat ~/.bashrc

```

```

# ~/.bashrc: executed by bash(1) for non-login shells.
# see /usr/share/doc/bash/examples/startup-files (in the package bash-doc)
# for examples

# Some aliases
alias pip=python3
alias python=python3
alias cl=clear

# If not running interactively, don't do anything
case $- in
    *i*) ;;
    *) return;;
esac

# don't put duplicate lines or lines starting with space in the history.
# See bash(1) for more options
HISTCONTROL=ignoreboth

# append to the history file, don't overwrite it
shopt -s histappend

# for setting history length see HISTSIZE and HISTFILESIZE in bash(1)
HISTSIZE=1000
HISTFILESIZE=2000

# check the window size after each command and, if necessary,
# update the values of LINES and COLUMNS.
shopt -s checkwinsize

# If set, the pattern "##" used in a pathname expansion context will
# match all files and zero or more directories and subdirectories.
#shopt -s globstar

# make less more friendly for non-text input files, see lesspipe(1)
[ -x /usr/bin/lesspipe ] && eval "$(SHELL=/bin/sh lesspipe)"

# set variable identifying the chroot you work in (used in the prompt below)
if [ -z "${debian_chroot:-}" ] && [ -r /etc/debian_chroot ]; then
    debian_chroot=$(cat /etc/debian_chroot)
fi

# set a fancy prompt (non-color, unless we know we "want" color)
case "$TERM" in
    xterm-color|*-256color) color_prompt=yes;;
esac

# uncomment for a colored prompt, if the terminal has the capability; turned
# off by default to not distract the user: the focus is on the output of commands, not on the prompt
#force_color_prompt=yes

if [ -n "$force_color_prompt" ]; then
    if [ -x /usr/bin/tput ] && tput setaf 1 >/dev/null; then
        # We have color support; assume it's compliant with Ecma-48
        # (ISO/IEC-6429). (Lack of such support is extremely rare, and such
        # a case would tend to support setf rather than setaf.)
        color_prompt=yes
    else
        color_prompt=
    fi
fi

if [ "$color_prompt" = yes ]; then
    PS1='${debian_chroot:+($debian_chroot)}[\033[01;32m]\u@\h[\033[00m]:[\033[01;34m]\w[\033[00m]'
else
    PS1='${debian_chroot:+($debian_chroot)}\u@\h:\w$ '
fi

```

```

unset color_prompt force_color_prompt

# If this is an xterm set the title to user@host:dir
case "$TERM" in
xterm*|rxvt*)
    PS1="\[\e[0;${debian_chroot:+($debian_chroot)}\u@\h: \w\]${PS1}"
    ;;
*)
    ;;
esac

# enable color support of ls and also add handy aliases
if [ -x /usr/bin/dircolors ]; then
    test -r ~/.dircolors && eval "$(dircolors -b ~/.dircolors)" || eval "$(dircolors -b)"
    alias ls='ls --color=auto'
    #alias dir='dir --color=auto'
    #alias vdir='vdir --color=auto'

    alias grep='grep --color=auto'
    alias fgrep='fgrep --color=auto'
    alias egrep='egrep --color=auto'
fi

# colored GCC warnings and errors
#export GCC_COLORS='error=01;31:warning=01;35:note=01;36:caret=01;32:locus=01:quote=01'

# some more ls aliases
alias ll='ls -alF'
alias la='ls -A'
alias l='ls -CF'

# Add an "alert" alias for long running commands.  Use like so:
#   sleep 10; alert
alias alert='notify-send --urgency=low -i "$( [ $? = 0 ] && echo terminal || echo error)" "$(history|tail

# Alias definitions.
# You may want to put all your additions into a separate file like
# ~/.bash_aliases, instead of adding them here directly.
# See /usr/share/doc/bash-doc/examples in the bash-doc package.

if [ -f ~/.bash_aliases ]; then
    . ~/.bash_aliases
fi

# enable programmable completion features (you don't need to enable
# this, if it's already enabled in /etc/bash.bashrc and /etc/profile
# sources /etc/bash.bashrc).
if ! shopt -oq posix; then
    if [ -f /usr/share/bash-completion/bash_completion ]; then
        . /usr/share/bash-completion/bash_completion
    elif [ -f /etc/bash_completion ]; then
        . /etc/bash_completion
    fi
fi

```

## 14.4. GH Actions as an example of scripts

As seen, we use scripts to run a collection of commands together with one call/ run of that action

Check out the [github action marketplace](#) to see other actions that are available and try to get a casual level of understanding of the types of things that people use actions for.

## 14.5. Bash Loops

We can also make loops like

```
for loopvar in list
do
# loop body
echo $loopvar
done
```

So, for example:

```
$ for name in Ayman madison greg sean
> echo $name
```

```
bash: syntax error near unexpected token `echo'
```

I'm got the "open loop" part wrong. Similar to how we use `{` in C & C++ or how we use `:` and indentation in Python. Bash has its own way of opening and closing a loop and that's `do` & `done`

```
for name in Ayman madison greg sean
> do
> echo $name
> done
```

```
Ayman
madison
greg
sean
```

Notes:

- The `>` is not typed, it is what happens since the interpreter knows that after we write the first line, the command is not complete.
- a list in bash is items with spaces, no brackets here (`Ayman madison greg sean`)

When we get the command back with the up arrow key, it puts it all on one line, because it was one command. The `;` (semicolon) separates the "lines"

```
for name in Ayman madison greg sean; do echo $name; done
```

What if we wanted to run the loop on a large list? We can make the list into a variable

```
NAMES="Ayman madison sean greg tyler trevor"
```

```
for name in $NAMES
> do
> echo $name
> done
```

```
Ayman
madison
sean
greg
tyler
trevor
```

## 14.6. Nesting commands

We can take the output of a command and put it in a variable or directly use it like we would use a variable

Can you find an example of a command being using nested in any of your kwl action files?

If we looked at `getassignment.yml` We see a bunch of lines that we've edited and worked on before.

```
# prepare badge lines
pretitle="prepare-$(cspt getbadgedate --prepare)
cspt getassignment --type prepare | gh issue create --title $pretitle --label prepare --body-file
```

We used a familiar command `cspt getbadgedate` to generate a variable

If we did this in our own terminals

```
cspt getbadgedate --prepare
```

```
2025-03-25
```

We get next class' date. Notice how in the script we combine it with `prepare-` Let's do the same thing in our terminal

```
pretitle="prepare-$(cspt getbadgedate --prepare)
```

Now, let's print the variable

```
echo $pretitle
```

```
prepare-2025-03-25
```

Let's see how many files we have in this directory

```
ls
```

```
LICENSE      _prepare/      faq/          notes/
README.md    _review/       files.sh       references.bib
_config.yml  _static/      genindex.md   requirements.txt
_data/        _toc.yml     img/          resources/
_lab/         _worksheets/ index.md      syllabus/
_practice/   activities/  logo.png     systools.png
```

Can we use a loop to print out each file at a time?

We can run a command to generate the list and then use that list as a variable by putting it inside `$( )` to run that command first.

Let's move to inclass repo

```
cd ~/Documents/systems/gh-inclass-AymanBx
```

```
for file in $(ls)
> do
> echo $file
> done
```

```
API.md
CONTRIBUTING.md
LICENSE.md
README.md
about.md
abstract_base_class.py
alternative_classes.py
docs/
helper_functions.py
important_classes.py
my_secrets/
philosophy.md
scratch.ipynb
setup.py
tests/
```

the `$( )` tells bash to run that command first and then hold its output as a variable for use elsewhere

Find a use of this in your experience report actions

We can modify what is in the `$( )`:

```
for file in $(ls -a); do echo $file; done
```

```
./  
../  
.git/  
.github/  
.gitignore  
.secrets  
API.md  
CONTRIBUTING.md  
LICENSE.md  
README.md  
about.md  
abstract_base_class.py  
alternative_classes.py  
docs/  
helper_functions.py  
important_classes.py  
my_secrets/  
philosophy.md  
scratch.ipynb  
setup.py  
tests/
```

## 14.7. Conditionals in bash

What are other programming language feature we would expect from a programming language like bash?

We can also do conditional statements

```
if test -f docs  
> then  
> echo "file"  
> fi
```

the key parts of this:

- `test` checks if a file or directory exists
- the `-f` option makes it check if the item is a *file*
- what to do if the condition is met goes after a `then` keyword
- the `fi` (backwards `if`) closes the if statement

Once again, similar to other programming languages. In bash to open and close an `if` statement we use `if`, `then` and `fi` to close the if statement

This outputs nothing because `docs` is a directory not a file.

If we switch it, we get output:

```
if test -f README.md; then echo "file"; fi
```

We can put the if inside of the loop.

Once we put `then` in, it works:

```
for file in $(ls)
> do
>   if test -f $file
>   then
>     echo $file
>   fi
> done
```

## 14.8. Script files

We can put our script into a file to automate more than one step all in one call

```
nano filecheck.sh
```

Let's get creative

```
# Introduction
echo "Hello and welcome to your first script"

echo "Hello world!"

# Now listing files only
for file in $(ls -a)
do
  if test -f $file
  then
    echo "$file" was found"
  fi
done

# Farewell
echo
echo "Goodbye!"
```

We can see that that file lives in out repo now

```
ls
```

|                        |                      |
|------------------------|----------------------|
| API.md                 | filecheck.sh         |
| CONTRIBUTING.md        | helper_functions.py  |
| LICENSE.md             | important_classes.py |
| README.md              | my_secrets/          |
| about.md               | philosophy.md        |
| abstract_base_class.py | scratch.ipynb        |
| alternative_classes.py | setup.py             |
| docs/                  | tests/               |

and run it with `bash <filename>`

```
bash filecheck.sh
```

```
Hello and welcome to your first script
Hello world!
.gitignore was found
.secrets was found
API.md was found
CONTRIBUTING.md was found
LICENSE.md was found
README.md was found
about.md was found
abstract_base_class.py was found
alternative_classes.py was found
filecheck.sh was found
helper_functions.py was found
important_classes.py was found
philosophy.md was found
scratch.ipynb was found
setup.py was found

Goodbye!
```

```
cat filecheck.sh
```

```
# Introduction
echo "Hello and welcome to your first script"

echo "Hello world!"

# Now listing files only
for file in $(ls -a)
do
    if test -f $file
    then
        echo $file" was found"
    fi
done

# Farewell
echo
echo "Goodbye!"
```

Remember, extensions from a computer's perspective don't matter. We use them only for us humans to be able to tell what a file represents

```
mv filecheck.sh filecheck
bash filecheck
```

```
Hello and welcome to your first script
Hello world!
.gitignore was found
.secrets was found
API.md was found
CONTRIBUTING.md was found
LICENSE.md was found
README.md was found
about.md was found
abstract_base_class.py was found
alternative_classes.py was found
filecheck was found
helper_functions.py was found
important_classes.py was found
philosophy.md was found
scratch.ipynb was found
setup.py was found

Goodbye!
```

It ran no problem!

## 14.9. CLI operations

When you are working sometimes it is helpful to be able to manipulate (or create) issues, pull requests or even releases from the command line.

This is how I post announcements. I work on the notes (a markdown file) in vs code and then I use the vscode terminal to commit, push, create a tag, and create a release. I can post the notes and notify you all that they are posted without leaving VScode at all; this makes it much simpler/faster than it would be using Brightspace

We can also search and filter them by piping the output to `grep` which searches the **contents** of a file (including stdin). We previously searched the file **names** with `find`. So `find` searches the paths that exist and `grep` actually reads the contents of the files, it does so faster than many other languages would be.

```
cd ~/Documents/systems/spring25-kwl-AymanBx
gh issue list -s all
```

```
Showing 30 of 57 issues in compsys-progtools/spring25-kwl-AymanBx that match your search
```

|     |                   |                  |                  |
|-----|-------------------|------------------|------------------|
| #62 | review-2025-02-20 | review           | about 8 hours... |
| #61 | practice-2025-... | practice         | about 8 hours... |
| #60 | prepare-2025-0... | prepare          | about 8 hours... |
| #59 | prepare-2025-0... | prepare          | about 2 days ago |
| #58 | Lab 6             |                  | about 2 days ago |
| #57 | Lab 6             |                  | about 2 days ago |
| #56 | practice-2025-... | practice, due    | about 6 days ago |
| #55 | review-2025-03-06 | review, due      | about 6 days ago |
| #54 | prepare-2025-0... | prepare, due     | about 1 day ago  |
| #53 | practice-2025-... | practice, exp... | about 1 day ago  |
| #52 | review-2025-03-04 | review, expired  | about 1 day ago  |
| #51 | prepare-2025-0... | prepare, due     | about 6 days ago |
| #50 | prepare-2025-0... | prepare, expired | about 1 day ago  |
| #49 | Lab 5             |                  | about 16 days... |
| #48 | practice-2025-... | practice, exp... | about 8 days ago |
| #47 | review-2025-02-25 | review, expired  | about 8 days ago |
| #46 | prepare-2025-0... | prepare, expired | about 6 days ago |
| #45 | test              |                  | about 22 days... |
| #44 | Lab 4             |                  | about 23 days... |
| #43 | review-2025-02-20 | review, expired  | about 13 days... |
| #42 | practice-2025-... | practice, exp... | about 13 days... |
| #41 | prepare-2025-0... | prepare, expired | about 8 days ago |
| #40 | practice-2025-... | practice, exp... | about 13 days... |
| #39 | review-2025-02-20 | review, expired  | about 13 days... |
| #38 | prepare-2025-0... | prepare, expired | about 8 days ago |
| #37 | review-2025-02-13 | review, expired  | about 20 days... |
| #36 | practice-2025-... | practice, exp... | about 20 days... |
| #35 | Lab 3             |                  | about 28 days... |
| #34 | practice-2025-... | practice, exp... | about 15 days... |
| #33 | review-2025-02-18 | review, expired  | about 15 days... |

`grep` can be used with pattern matching as well

```
gh issue list -s all | grep "Lab"
```

|    |      |       |                               |
|----|------|-------|-------------------------------|
| 58 | OPEN | Lab 6 | 2025-03-17 19:26:28 +0000 UTC |
| 57 | OPEN | Lab 6 | 2025-03-17 18:50:52 +0000 UTC |
| 49 | OPEN | Lab 5 | 2025-03-03 19:21:55 +0000 UTC |
| 44 | OPEN | Lab 4 | 2025-02-24 19:48:33 +0000 UTC |
| 35 | OPEN | Lab 3 | 2025-02-19 19:53:10 +0000 UTC |

Learning more about `grep` is a good explore badge topic

## 14.10. Badge Hints

Here are the options:

- See the options for `gh pr list`.
- Use two strategies for what you are the author and when you are the reviewer
- Try the json option on `gh pr list` and see how it can help you
- I use bash in the your experience badge action to make a file with a date in the file name
- You can run a file from different locations

## 14.11. Experience Report Evidence

## 14.12. Prepare for Next Class

1. Read the notes from March 18th. We will build on these directly in the future. You need to have the `test` repo with the same status as me
2. Add to your IDE idethoughts.md file if you have not in a few days on a dedicated `ide_prep` branch. This is prep for after a few weeks from now, not for 03/25; keep this branch open until it is specifically asked for
  - I have not posted an outline for idethoughts.md. Start with sections like `preference`, `debugging methods`, `organizations`, `tools/extentions` Add any other thoughts on using different IDEs you can think of. These sections are not limiting nor are they mandated

## 14.13. Badges

**Review**    **Practice**

1. Update your KWL Chart learned column with what you've learned
2. write a bash script to make it so that `cat`ing the files in your `gh_inclass` repo does not make the prompt move

## 14.14. Questions

# 15. Why did we learn the plumbing commands?

You will not typically use them on a day to day basis, but they are a good way to see what happens at the interim steps and make sure that you have the right understanding of what git does.

A **correct** understanding is essential for using more advanced features

While there is of course some content that we want you to know after this course, my goal is also to teach you process, by modeling it.

No one will ever know all of the things but you can be fast or slow at finding answers.

And you can find correct answers, incorrect answers, or looks-okay-but-you-will-regret-this-later answers.

you do not want to become the colleague that everyone regrets working with

My goal is that you get good at *quickly* finding **correct** answers.

Large language models will not do that for you.

## 15.1. Today's Questions

- What are references?
- How can I release and share my code?

- How else can git help me?

When does the contents of a file get hashed?

- [ ] every time you edit a file, git hashes it automatically
- [x] when a file is staged for commit
- [ ] any time a staged file is edited
- [ ] when the commit is created

What type of git object stores the name of a file?

- [x] tree
- [ ] blob
- [ ] commit
- [ ] branch

## 15.2. What does git status do?

*compares the working directory to the current state of the active branch and the index*

- we can see the working directory with: `ls`
- we can see the active branch in the `HEAD` file
- what is its status?

Recall that we:

- created a blob objct directly
- created a file
- hashed the file, to create its blob object
- added the file to the index
- wrote a tree from files in the index
- created a commit object with a tree and a commit message
- copied the commit hash and placed it inside a new file `.git/refs/heads/main` using `echo` and `>`

Let's use inspection to review where our repo is left off from last week.

```
git status
```

```
On branch main
```

```
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test.txt
```

## 15.3. Branches are references

branches are not git objects, they are references.

We can see that by where they are stored.

How can we inspect to see where references are stored?

using ls and/or find

We can see our list of object with `find`

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547  
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4  
.git/objects//d8/329fc1cc938780ffd9f94e0d364e0ea74f579  
.git/objects//e3/ba10cb02de504d4f48b9af4934ddcc4d0be3df  
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Since we made most of these objects as not commits, the hashes are mostly shared, but one of the hashes is unique so we worked together to identify that one.

Then for that unique hash, we confirmed it was a commit using `git cat-file` to view its type

```
git cat-file -t e3ba
```

```
commit
```

as expected

Now let's look at what the `HEAD` pointer says to try to understand why it does not see that commit, since we know that git status works from `HEAD`

```
cat .git/HEAD
```

```
.git/refs/heads/main
```

### ! Important

git updates the file for the branch each time you add a commit with `git commit`, the first time `git commit` is run it also creates the file

If we look in the folder, we can see what is in there

```
ls .git/refs/heads/
```

```
main
```

```
cat .git/refs/heads/main
```

```
e3ba10cb02de504d4f48b9af4934ddcc4d0be3df
```

### 15.3.1. Updating a branch manually

The branch file is named as the branch (here `main`) and stored in the `.git/refs/heads/` folder and contains the full hash of the commit that branch is pointing to.

Since we made the commit manually, we need to move the branch manually too. We will use echo, by copying the full hash from above (I copied the part after the last `/` in the list of objects above and then typed the `e3` before pasting).

Mine looks like:

```
echo e3ba10cb02de504d4f48b9af4934ddcc4d0be3df > .git/refs/heads/main
```

but your commit hash will be different than mine.

Now we check with git again:

```
git status
```

```
On branch main
Changes not staged for commit:
  (use "git add <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
    modified:   test.txt

no changes added to commit (use "git add" and/or "git commit -a")
```

and now we can see that it no longer sees the staged file and *does* see our commit.

```
git cat-file -p 188a
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Ayman Sandouk <Ayman_sandouk@uri.edu> 1677177139 -0500
committer Ayman Sandouk <Ayman_sandouk@uri.edu> 1677177139 -0500
first commit
```

```
git log
```

So we now have HEAD-> main and main -> our commit -> tree -> blob.

Branches in git are references to specific commits.

## 15.4. Git Tags

To practice this, let's go to the github inclass repo

```
cd gh-
```

(this is not the full name of the directory, but starting with this and then pressing tab will get you there)

The other type of git reference is called a tag. There are two types of tags:

- a lightweight tag is like a branch that does not move
- an annotated tag is a git object, like a commit almost.

We can see where they are stored in the `refs` folder:

```
ls .git/refs/
```

```
heads    remotes    tags
```

Remember:

- heads are branches (like `main`)
- remotes are servers that we can push to (eg [https://github.com/...](https://github.com/))

Before we make a tag, let's make sure we are on the most up to date main. First we check the branch and working directory:

```
git status
```

```
On branch main
Your branch is up to date with 'origin/main'.

nothing to commit, working tree clean
```

main, clean working tree is good.

Next we pull to update from github

```
git pull
```

```
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 955 bytes | 136.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-sp24-Ayman_sandouk
 * [new branch]      1-create-a-add-a-classmate -> origin/1-create-a-add-a-classmate
Already up to date.
```

and we get the updates

Now, to make a tag we use `git tag <name of tag>` to create a lightweight tag

```
git tag v3.0
```

This command has no output, but we will inspect

```
ls .git/refs/tags/
```

```
v3.0
```

we see this created a file for the tag,

we will look at this too

```
cat .git/refs/tags/v3.0
```

```
0d6ef1d3db0729088d515b35b588f39af5b770fd
```

the lightweight tag is just like a branch, another pointer to the commit.

The difference is that this does not move when we create new commits, so this is like a shortcut to a specific commit.

We can see the tag relative to branches in the log as well:

```
git log
```

```
commit 0d6ef1d3db0729088d515b35b588f39af5b770fd (HEAD -> main, tag: v3.0, origin/main, origin/HEAD)
Merge: 9f39946 e3b192a
Author: AymanBx <Ayman_sandouk@uri.edu>
Date:   Thu Feb 8 13:43:19 2024 -0500

    Merge pull request #6 from compsys-progtools/organization

        first pass organizing

commit e3b192aa0cd490226e8adcd81d3d0b95adb5676b (origin/organization, organization)
Author: Ayman Sandouk <Ayman_sandouk@uri.edu>
Date:   Tue Feb 6 13:41:53 2024 -0500

        organizng

commit 260c9c309922970f80bfa2c93cc23bcfb962740
Author: Ayman Sandouk <Ayman_sandouk@uri.edu>
Date:   Tue Feb 6 13:06:20 2024 -0500

        describe the files

commit 29ffc88519103085ed3a2ab01cffb3c99d70fc6a
Author: Ayman Sandouk <Ayman_sandouk@uri.edu>
Date:   Tue Feb 6 12:59:20 2024 -0500
```

If we push:

```
git push
```

```
Everything up-to-date
```

we see nothing happens initially.

Tags have to be pushed explicitly

```
git push --tags
```

We can look at how [tags are represented on github](#). After opening the repo up in browser:

```
gh repo view --web
```

Tags enable [releases](#), where are [defined on github](#) as:

Releases are deployable software iterations you can package and make available for a wider audience to download and use.

We can [create a release for the tag we made](#)

#### 15.4.1. We can checkout any commit, not just branches

```
git checkout 188a
```

Note: switching to '188a'.

You are `in 'detached HEAD' state`. You can look around, make experimental changes `and` commit them, `and` you can discard `any` commits you make `in` this state without impacting `any` branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may do so (now `or` later) by using `-c with` the switch command. Example:

```
git switch -c <new-branch-name>
```

Or undo this operation `with`:

```
git switch -
```

Turn off this advice by setting config variable `advice.detachedHead` to `false`

```
HEAD is now at 188a75e first commit
```

we can follow from this warning.

```
cat .git/HEAD
```

```
188a75ef66b6a85be0ab68d8575ec27808881dfc
```

This changes the head pointer directly to the commit.

```
ls
```

```
test.txt
```

and checkout also updates the working directory

```
cat test.txt
```

```
version 1
```

```
git checkout test
```

```
Switched to branch 'test'
```

```
ls
```

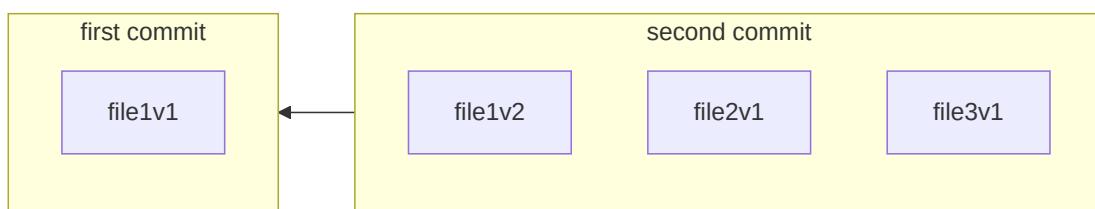
```
test.txt
```

```
cat test.txt
```

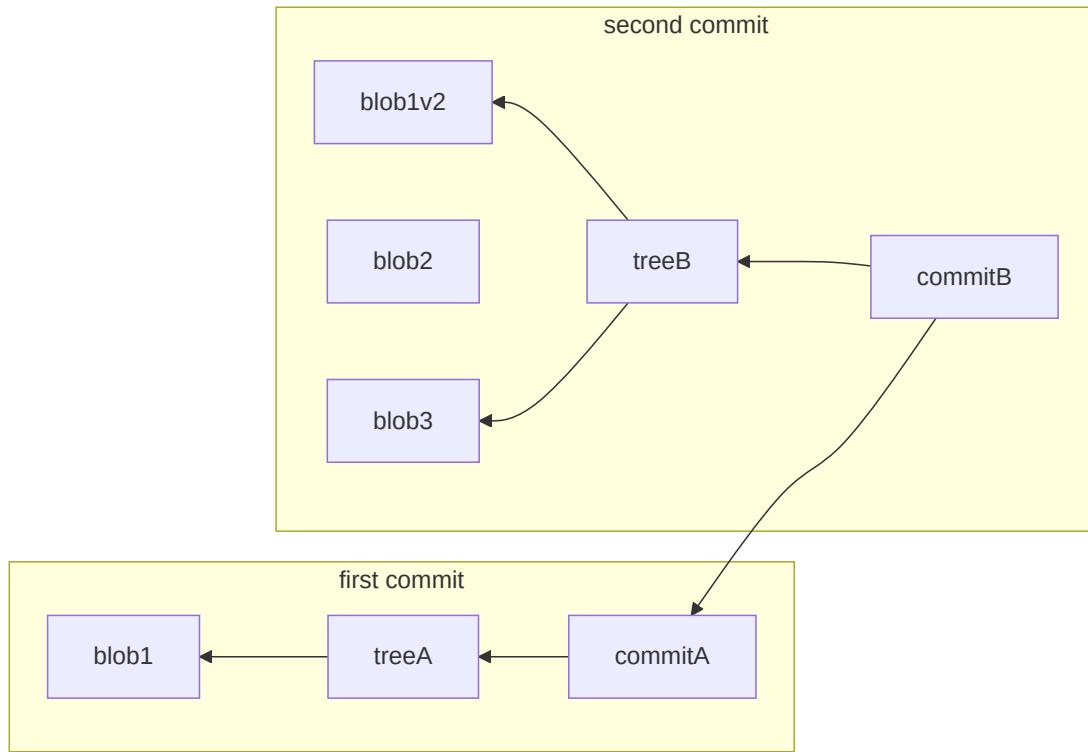
```
version 1
```

## 15.5. What does this mean?

We tend to think of commits like this:



In reality



## 15.6. We can move pointers around freely

```
flowchart BT
blob1
blob2
blob3
subgraph A[first commit]
    direction TB
    commitA
    %% file1v1
    treeA
    commitA-->treeA

end
subgraph B[second commit]
    direction TB
    %% file1v2
    %% file2v1
    %% file3v1
    commitB
    treeB
    commitB-->treeB
end
%% subgraph C[third commit]
%% direction TB
%% %% file1v2
%% %% file2v1
%% %% file3v1
%% commitC
%% commitC-->treeC
%% end
B--> A

%% treeC-->blob1
treeC>blob2
treeB-->blob3
end
%% B--> A
commitB -->commitA
```

## 15.7. Experience Report Evidence

write your git status and object list to a file.

## 15.8. What is a hash?

a hash is:

- a fixed size value that can be used to represent data of arbitrary sizes
- the *output* of a hashing function
- often fixed to a hash table

Common examples of hashing are lookup tables and encryption with a cryptographic hash.

A hashing function could be really simple, to read off a hash table, or it can be more complex.

For example:

|      |         |
|------|---------|
| Hash | content |
| 0    | Success |
| 1    | Failure |

If we want to represent the status of a program running it has two possible outcomes: success or failure. We can use the following hash table and a function that takes in the content and returns the corresponding hash. Then we could pass around the 0 and 1 as a single bit of information that corresponds to the outcomes.

This lookup table hash works here.

In a more complex scenario, imagine trying to hash all of the new terms you learn in class.

A table would be hard for this, because until you have seen them all, you do not know how many there will be. A more effective way to hash this, is to derive a *hashing function* that is a general strategy.

A *cryptographic* hash is additionally:

- unique
- not reversible
- similar inputs hash to very different values so they appear uncorrelated

Now lets go through each of these properties

### 15.8.1. Cryptographic Hashes are unique

This means that two different values we put in should give different results.

For this property alone, a simple function could work:

```
def basic_unique(input):
    return input
```

but this is not a hash because its length would not be constant and not a cryptographic has because it is easily reversible.

### 15.8.2. Cryptographic Hashes are not reversible

This means that given the hash (output), we cannot compute the message(input).

Any function that gives the same output for two (or more) values meets this criteria.

for example modulus:

13%3

1

10%3

1

It can be any function that gives the same output for two (or more) values.

but this is not a cryptographic hash

We can use the git hashing algorithm without writing to the repo too:

Then we get the hash back. Try changing the input just a little and running the hashing algorithm again.

do similar inputs have similar hashes?

### 15.8.3. Similar inputs lead to seemingly uncorrelated outputs

We can see this using git:

```
echo "it's almost spring" | git hash-object --stdin
```

```
1327354a0d47111e361ec52acc46b9509bed69a0
```

and we get a hash

and then we change it just a little bit:

```
echo "it's almsot spring" | git hash-object --stdin
```

```
f798473089b4a00684cd0f13a6f2d1d7a0e033ba
```

and now the hash is completely different

### 15.8.4. Hashes are fixed length

So, no matter the size of the input, we get back the same length.

This is good for memory allocation reasons.

We could again write a function that only does this simply:

```
def fixed_len(input):
    """
    pad or trim
    """
    len_target=100
    str_in = str(input)
    if len(str_in)< len_target:
        return str_in.ljust(len_target, '-')
    else:
        return str_in[:len_target]
```

Back to it being not a cryptographic hash

Try

```
fixed_len("lkjhlkjhlkjhlkjlkjhkjhkljhjlkhkjhkjhkljhkhjlhkjlhhhhjlhlfgfdfgfgfkgfkgfkhkjhlkjhljkhnbh123")
```

Try

```
fixed_len("lkjhlkjhlkjhlkjlkjhkjhkljhjlkhkjhkjhkhjlhkjlhhhhjlhlfgfdfgfgfkgfkgfkhkjhlkjhljkhnbh258")
```

What are some ways a hash could be used?

## 15.9. How can hashes be used?

Hashes can then be used for a lot of purposes:

- message integrity (when sending a message, the unhashed message and its hash are both sent; the message is real if the sent message can be hashed to produce the same hash)
- password verification (password selected by the user is hashed and the hash is stored; when attempting to login, the input is hashed and the hashes are compared)
- file or data identifier (eg in git)

### 15.9.1. Hashing in passwords

Passwords can be encrypted and the encrypted information is stored, then when you submit a candidate password it can compare the hash of the submitted password to the hash that was stored. Since the hashing function is nonreversible, they cannot see the password.

[Password Breach blog post](#)

Some sites are negligent and store passwords unencrypted, if your browser warns you about such a site, proceed with caution and definitely do not reuse a password you ever use. (you *should never* reuse passwords, but especially do not if there is a warning)

An attacker who gets one of those databases, cannot actually read the passwords, but they could build a lookup table. For example, "password" is a bad password because it has been hashed in basically every algorithm and then the value of it can be reversed. Choosing an uncommon password makes it less likely that your password exists in a lookup table.

For example, in SHA-1 the hashing algorithm that git uses

```
echo "password" | git hash-object --stdin
```

f3097ab13082b70f67202aab7dd9d1b35b7ceac2

## 15.10. Hashing in Git

In git we hash the content directly to store it in both the database (.git) directory and the commit information.

Recall, when we were working in our test repo we created an empty repository and then added content directly, we all got the same hash, but when we used git commit our commits had different hashes because we have different names and made the commits at different seconds.

We also saw that *two* entries were created in the `.git` directory for the commit.

Git was originally designed to use SHA-1.

Then the [SHA-1 collision attack](#) was discovered

Git switched to hardened HSA-1 in response to a collision.

In that case it adjusts the SHA-1 computation to result in a safe hash. This means that it will compute the regular SHA-1 hash for files without a collision attack, but produce a special hash for files with a collision attack, where both files will have a different unpredictable hash. [from](#).

[they will change again soon](#)

GitHub uses git, it is not an alternative implementation or a fork, so yes it will switch too. The developers at GitHub and other git hosts are among the most impacted by the change since they write code that directly interacts with git objects.

git uses the SHA hash primarily for uniqueness, not privacy

It does provide some *security* assurances, because we can check the content against the hash to make sure it is what it matches.

This is a Secure Hashing Algorithm that is derived from cryptography. Because it is secure, no set of mathematical options can directly decrypt an SHA-1 hash. It is designed so that any possible content that we put in it returns a unique key. It uses a combination of bit level operations on the content to produce the unique values.

The SHA-1 Algorithm hashes content into a fixed length of 160 bits.

how many different possible hashes can it produce?

- [ ]  $160 \times 160$
- [x]  $2^{160}$
- [ ]  $160^2$
- [ ]  $160^2$

This means it can produce  $2^{160}$  different hashes. Which makes the probability of a collision very low.

The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about  $2^{80}$  (the formula for determining collision probability is  $p = (n(n - 1)/2) * (1/2^{160})$ ).  $2^{80}$  is  $1.2 \times 10^{24}$  or 1 million billion billion. That's 1,200

times the number of grains of sand on the earth.

#### —[A SHORT NOTE ABOUT SHA-1 in the Git Documentation](#)

The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about  $2^{80}$  (the formula for determining collision probability is  $p = n(n-1)/2 * (1/2^{160})$ ).  $2^{80}$  is  $1.2 \times 10^{24}$  or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.

#### —[A SHORT NOTE ABOUT SHA-1 in the Git Documentation](#)

## 15.11. Prepare for Next Class

- [ ] Think about what you know about networking
- [ ] Get the big ideas of hpc, by reading this [IBM intro page](#) and [some hypothetical people](#) who would attend an HPC carpentry workshop. Make a list of key terms as an issue comment
- [ ] Look over build/explore ideas if you plan to do any on discussion repo. [Build ideas](#). [Explore ideas](#)

## 15.12. Badges

Review

Practice

- [ ] Read about the Learn more about the [SHA-1 collision attack](#)
- [ ] Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object (the minimum is 4. That's usually enough for us to use something like `cat-file` command, `git tag` or even `git checkout`) and include that number in gitcounts.md with a title `# Git counts`. How many files would have to exist to reach that number of objects assuming every file was edited in each of two commits? *If you get stuck, outline what you know and then request a review.*
- [ ] Create tagtypeexplore.md with the template below. Determine how many of the tags in the course website are annotated vs lightweight using. (You may need to use `git pull --tags` in your clone of the course website)

```
# Tags
```

```
<!-- short defintion/description in your own words of what a tag is and what it is for -->
```

```
## Inspecting tags
```

```
Course website tags by type:
```

- annotated:
- lightweight:

## 15.13. Questions

1. think about what you know about networking
2. make sure you have putty if using windows

3. get the big ideas of hpc, by reading this [IBM intro page](#) and [some hypothetical people](#) who would attend an HPC carpentry workshop. Make a list of key terms as an issue comment

2. Look over build/explore ideas if you plan to do any.

- [discussion repo](#)
- [website repo](#)
- [courseutils](#)

{"lesson\_part": "activity", "ac\_type": "prepare"}

1. Read the 3 bulleted examples of [why use a cluster](#) from HPC carpentry.

2. Read [this discussion of why using a remote server](#)

## 16. How can I work on a remote server?

### 16.1. How can I authenticate more securely from a terminal?

### 16.2. What are remote servers and HPC systems?

Today we will connect to a remote server and learn new bash commands for working with the *content* of files.

### 16.3. Connecting to Seawulf

We connect with secure shell or `ssh` from our terminal (GitBash or Putty on windows) to URI's teaching High Performance Computing (HPC) Cluster [Seawulf](#).

This cluster is for course related purposes at URI, if you want to use a HPC system of some sort for a side project, consider Amazon Web Services, Google Cloud, or Microsoft Azul services, you can get some allocation for free a a student.

If you are doing research supervised by a URI professor, there are other servers on campus and URI participates in a regional HPC resource as well.

Our login is the part of your uri e-mail address before the @ and I will tell you how to find your default password if you missed class (do not want to post it publicly). Comment on your experience report PR to ask for this information.

```
ssh -l ayman_sandouk seawulf.uri.edu
```

When it logs in it looks like this and requires you to change your password. They configure it with a default and with it past expired.

```
The authenticity of host 'seawulf.uri.edu (131.128.217.210)' can't be established.  
ECDSA key fingerprint is SHA256:RwhTUyjWLqwohXiRw+tYlTiJEbqX2n/drCpkIwQVCro.  
Are you sure you want to continue connecting (yes/no/[fingerprint])? y  
Please type 'yes', 'no' or the fingerprint: yes  
Warning: Permanently added 'seawulf.uri.edu,131.128.217.210' (ECDSA) to the list of known hosts.  
ayman_sandouk@seawulf.uri.edu's password:  
You are required to change your password immediately (root enforced)  
WARNING: Your password has expired.  
You must change your password now and login again!  
Changing password for user ayman_sandouk.  
Changing password for ayman_sandouk.  
(current) UNIX password:  
New password:  
Retype new password:  
passwd: all authentication tokens updated successfully.  
Connection to seawulf.uri.edu closed.
```

You use the default password when prompted for your username's password. Then again when it asks for the **(current)** **UNIX password:**. Then you must type the same, new password twice.

### Choose a new password you will remember, we will come back to this server

after you give it a new password, then it logs you out and you have to log back in.

We have logged into our home directory which is empty

```
ls
```

it is bash, so we can use regular bash commands

```
pwd  
/home/ayman_sandouk
```

```
whoami
```

```
ayman_sandouk
```

Notice that the prompt says **uriusername@seawulf** to indicate that you are logged into the server, not working locally.

## 16.4. Downloading files

**wget** allows you to get files from the web.

we are going to download some genetics data that is compressed as a practice tool

```
wget http://www.hpc-carpentry.org/hpc-shell/files/bash-lesson.tar.gz
```

```
ls
```

```
bash-lesson.tar.gz
```

then we can unzip it:

```
tar -xvf bash-lesson.tar.gz
```

we can use the `man` command to learn about a command from the “man(ual) page”

## 16.5. Working with large files

One of these files, contains the entire genome for the common fruitfly, let's take a look at it: \

```
cat dmel-all-r6.19.gtf
```

We see that this actually takes a long time to output and is way tooo much information to actually read. In fact, in order to make the website work, I had to cut that content using command line tools, my text editor couldn't open the file and GitHub was unhappy when I pushed it.

For a file like this, we don't really want to read the whole file but we do need to know what it's strucutred like in order to design programs to work with it.

`head` lets us look at the first 10 lines.

```
head dmel-all-r6.19.gtf
```

We can use the `-n` parameter to change the number.

And, `tail` shows the last few.

```
tail dmel-all-r6.19.gtf
```

which in this case looks mostly the same

```
2L FlyBase exon 782124 782181 . + . gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase exon 782238 782441 . + . gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase exon 782495 782885 . + . gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase start_codon 781297 781299 . + 0 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase CDS 781297 782048 . + 0 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase CDS 782124 782181 . + 1 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase CDS 782238 782441 . + 0 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase CDS 782495 782821 . + 0 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase stop_codon 782822 782824 . + 0 gene_id "FBgn0041250"; gene_symbol "Gr21a
2L FlyBase 3UTR 782825 782885 . + . gene_id "FBgn0041250"; gene_symbol "Gr21a
```

We can also see how much content is in the file `wc` give a word count and with its `-l` parameter gives us the number of lines.

```
wc -l dmel-all-r6.19.gtf
```

```
542048 dmel-all-r6.19.gtf
```

Over five hundred forty thousand lines is a lot.

How can we get the number of lines in each of the `.fastq` files?

*remember other times we have used patterns*

```
wc -l *.fastq
```

when it does work, we also get the total.

We can use redirects as before to save these to a file:

```
wc -l *.fastq > linecounts.txt
```

```
cat linecounts.txt
20000 SRR307023_1.fastq
20000 SRR307023_2.fastq
20000 SRR307024_1.fastq
20000 SRR307024_2.fastq
20000 SRR307025_1.fastq
20000 SRR307025_2.fastq
20000 SRR307026_1.fastq
20000 SRR307026_2.fastq
20000 SRR307027_1.fastq
20000 SRR307027_2.fastq
20000 SRR307028_1.fastq
20000 SRR307028_2.fastq
20000 SRR307029_1.fastq
20000 SRR307029_2.fastq
20000 SRR307030_1.fastq
20000 SRR307030_2.fastq
320000 total
```

We can also search files, without loading them all into memory or displaying them, with `grep`:

```
grep Act5c dmel-all-r6.19.gtf
```

```
grep mRNA dmel-all-r6.19.gtf
```

this output a lot, so the output is truncated here

```
X      FlyBase mRNA    19961689    19968479 . . + . gene_id "FBgn0031081"; ge
2L      FlyBase mRNA    781276  782885 . . + . gene_id "FBgn0041250"; gene_symbol "Gr21a
```

and we can combine `grep` with `wc` to count occurrences.

```
grep mRNA dmel-all-r6.19.gtf | wc -l
34025
```

## 16.6. File permissions

Let's make a small script, recalling what we have learned so far:

```
echo "echo 'script works'" >> demo.sh
```

We can confirm that the script looks like a we expected

```
cat demo.sh
echo 'script works'
```

One thing we could do is to run the script using `./`

```
./demo.sh
```

but we get a permission denied error

```
-bash: ./demo.sh: Permission denied
```

By default, files have different types of permissions: read, write, and execute for different users that can access them. To view the permissions, we can use the `-l` option of `ls`.

```
ls -l
total 138452
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 12534006 Apr 18 2021 bash-lesson.tar.gz
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 20 Mar 8 13:12 demo.sh
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 77426528 Jan 16 2018 dmel-all-r6.19.gtf
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 721242 Jan 25 2016 dmel_unique_protein_isoforms_fb_2016
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 25056938 Jan 25 2016 gene_association.fb
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 447 Mar 8 13:07 linecounts.txt
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_2.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_1.fastq
-rw-r--r-- 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_2.fastq
```

For each file we get 10 characters in the first column that describe the permissions. The 3rd column is the username of the owner, the fourth is the group, then size date revised and the file name.

We are most interested in the 10 character permissions. The fist column indicates if any are directories with a `d` or a `-` for files. We have no directories, but we can create one to see this.

```
mkdir results
```

```
ls -l
total 138452
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12534006 Apr 18 2021 bash-lesson.tar.gz
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 20 Mar 8 13:12 demo.sh
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 77426528 Jan 16 2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 721242 Jan 25 2016 dmel_unique_protein_isoforms_fb_2016
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 25056938 Jan 25 2016 gene_association.fb
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 447 Mar 8 13:07 linecounts.txt
**drwxr-xr-x. 2 ayman_sandouk spring2022-csc392 10 Mar 8 13:16 results**
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_2.fastq
```

We can see in the bold line, that the first character is a d.

The next nine characters indicate permission to **read**, **w**rite****, and **e**xecute**** a file. With either the letter or a **-** for permissions not granted, they appear in three groups of three, three characters each for owner, group, anyone with access.

If we want to run the file, we *can* instead use **bash** directly, but this is limited relative to calling our script in other ways.

```
bash demo.sh
script works
```

Instead, to add execute permission, we can use **chmod**

```
chmod +x demo.sh
```

```
ls -l
total 138452
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12534006 Apr 18 2021 bash-lesson.tar.gz
-rwxr-xr-x. 1 ayman_sandouk spring2022-csc392 20 Mar 8 13:12 demo.sh
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 77426528 Jan 16 2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 721242 Jan 25 2016 dmel_unique_protein_isoforms_fb_2016
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 25056938 Jan 25 2016 gene_association.fb
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 447 Mar 8 13:07 linecounts.txt
drwxr-xr-x. 2 ayman_sandouk spring2022-csc392 10 Mar 8 13:16 results
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_2.fastq
```

```
./demo.sh
script works
```

We can add a bit more to our script to make it more interesting

```
nano demo.sh
```

**note here nano is all we have without a lot of extra work**

```
for VAR in *.gz
do
    echo $VAR
done

echo 'script works'
```

and note that that does not change the permission.

```
ls -l
```

That output is long, how could we make it shorter and more focused on what we want?

use grep!

```
ls
```

```
example
github-in-class-ayman_sandouk-1
kwl
seawulf
test
testobj.md
tiny-book
```

```
ssh -l ayman_sandouk seawulf.uri.edu
```

```
ayman_sandouk@seawulf.uri.edu's password:
```

```
Last failed login: Thu Mar 30 12:53:35 EDT 2023 from pool-96-238-44-82.prvdri.fios.verizon.net on ssh:not
There was 1 failed login attempt since the last successful login.
Last login: Thu Mar 30 11:54:40 2023 from pool-72-87-118-171.prvdri.fios.verizon.net
```

```
[ayman_sandouk@seawulf ~]$ pwd
```

```
/home/ayman_sandouk
```

```
[ayman_sandouk@seawulf ~]$ ls
```

|   |                   |
|---|-------------------|
| bash-lesson.tar.gz                          | SRR307024_2.fastq |
| bash-lesson.tar.gz.1                        | SRR307025_1.fastq |
| demo.sh                                     | SRR307025_2.fastq |
| dmel-all-r6.19.gtf                          | SRR307026_1.fastq |
| dmel_unique_protein_isoforms_fb_2016_01.tsv | SRR307026_2.fastq |
| gene_association.fb                         | SRR307027_1.fastq |
| linecounts.txt                              | SRR307027_2.fastq |
| my_job.sh                                   | SRR307028_1.fastq |
| results                                     | SRR307028_2.fastq |
| slurm-23950.out                             | SRR307029_1.fastq |
| SRR307023_1.fastq                           | SRR307029_2.fastq |
| SRR307023_2.fastq                           | SRR307030_1.fastq |
| SRR307024_1.fastq                           | SRR307030_2.fastq |

```
[ayman_sandouk@seawulf ~]$ mkdir example
```

```
[ayman_sandouk@seawulf ~]$ cd example/
```

```
[ayman_sandouk@seawulf example]$ pwd
```

```
/home/ayman_sandouk/example
```

```
[ayman_sandouk@seawulf example]$ mkdir ex2
```

```
[ayman_sandouk@seawulf example]$ cd ex2/
```

```
[ayman_sandouk@seawulf ex2]$ pwd
```

```
/home/ayman_sandouk/example/ex2
```

```
[ayman_sandouk@seawulf ex2]$ cd
```

```
[ayman_sandouk@seawulf ~]$ ls -l
```

```
total 150704
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12534006 Apr 18 2021 bash-lesson.tar.gz
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12534006 Apr 18 2021 bash-lesson.tar.gz.1
-rwxr-xr-x. 1 ayman_sandouk spring2022-csc392 20 Oct 26 17:11 demo.sh
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 77426528 Jan 16 2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 721242 Jan 25 2016 dmel_unique_protein_isoforms_fb_2016
drwxr-xr-x. 3 ayman_sandouk spring2022-csc392 24 Mar 30 12:59 example
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 25056938 Jan 25 2016 gene_association.fb
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 447 Mar 8 2022 linecounts.txt
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 84 Mar 8 2022 my_job.sh
drwxr-xr-x. 2 ayman_sandouk spring2022-csc392 10 Mar 8 2022 results
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 89 Mar 8 2022 slurm-23950.out
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625262 Jan 25 2016 SRR307023_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625376 Jan 25 2016 SRR307024_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625286 Jan 25 2016 SRR307025_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625302 Jan 25 2016 SRR307026_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625312 Jan 25 2016 SRR307027_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625338 Jan 25 2016 SRR307028_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625390 Jan 25 2016 SRR307029_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1625318 Jan 25 2016 SRR307030_2.fastq
```

```
[ayman_sandouk@seawulf ~]$ ls -l --block-size=M
```

```
total 148M
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12M Apr 18 2021 bash-lesson.tar.gz
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 12M Apr 18 2021 bash-lesson.tar.gz.1
-rwxr-xr-x. 1 ayman_sandouk spring2022-csc392 1M Oct 26 17:11 demo.sh
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 74M Jan 16 2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1M Jan 25 2016 dmel_unique_protein_isoforms_fb_2016_01.ts
drwxr-xr-x. 3 ayman_sandouk spring2022-csc392 1M Mar 30 12:59 example
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 24M Jan 25 2016 gene_association.fb
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1M Mar 8 2022 linecounts.txt
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1M Mar 8 2022 my_job.sh
drwxr-xr-x. 2 ayman_sandouk spring2022-csc392 1M Mar 8 2022 results
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 1M Mar 8 2022 slurm-23950.out
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307023_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307023_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307024_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307024_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307025_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307025_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307026_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307026_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307027_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307027_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307028_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307028_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307029_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307029_2.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307030_1.fastq
-rw-r--r--. 1 ayman_sandouk spring2022-csc392 2M Jan 25 2016 SRR307030_2.fastq
```

```
[ayman_sandouk@seawulf ~]$ exit
```

Remember to logout to be courteous to others who are sharing the resources with you. We don't want to be reserving resources that we aren't using.

```
logout
```

```
Connection to seawulf.uri.edu closed.
```

## 16.7. Experience Report Evidence

## 16.8. Prepare for Next Class

- [] Ensure you can log into seawulf

## 16.9. Review today's class

### Important

Today's badges are [integrative](#) 2x badges.

## 16.10. Badges

Review

Practice

### ! Important

This is an [integrative](#) 2x badge.

1. File permissions are represented numerically often in octal, by transforming the permissions for each level (owner, group, all) as binary into a number. Add octal.md to your KWL repo and answer the following. Try to think through it on your own, but you may look up the answers, as long as you link to (or ideally cite using jupyterbook citations) a high quality source.

1. Describe how to transform the permissions [`r--`, `rw-`, `rwx`] to octal, by treating each set
1. Transform the permission we changed our script to `rwxr-xr-x` to octal.
1. Which of the following would prevent a file from being edited by other people 777 or 755 and wh

2. Read through this [rsa encryption demo site](#) to review how it works. Find 2 other encryption algorithms that could be used with ssh (hint: try [man ssh](#) or [read it online](#)) and compare them in encryption\_compare.md. Your comparison should be enough to advise someone which is best and why, but need not get into the details.

## 16.11. Questions

1. install gcc locally
2. ensure you can log into seawulf

## 17. What happens when I build code in C?

### 17.1. What is *buiding* code?

Building is transforming code from some input format (some text for example) into the final desired format.

This can mean different things in different contexts. For example:

- the course website is built from markdown files into html output using jupyter-book
- a C program is built from C source code to executable that the machine can understand as the output

We sometimes say that compiling takes code from source to executable, but this process is actually multiple stages and compiling is *one* of those steps.

We will focus on *what* has to happen more than *how* it all happens.

CSC301, 402, 501, 502 go into greater detail on how languages work.

Our goal is to:

- (where applicable) give you a preview
- get enough understanding of what happens to know where to look when debugging

# 18. How can I authenticate more securely from a terminal?

Previous notes:

- [sp22 server](#)
- [sp22 ssh keys](#)
- [f22 server](#)

## 18.1. remember seawulf?

```
ssh -l ayman_sandouk seawulf.uri.edu
```

Or

```
ssh -l ayman_sandouk@seawulf.uri.edu
```

```
ayman_sandouk@seawulf.uri.edu's password:
```

```
[ayman_sandouk@seawulf ~]$ pwd
```

```
logout
```

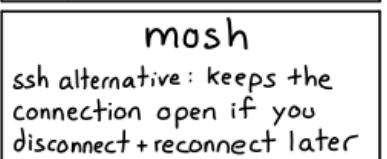
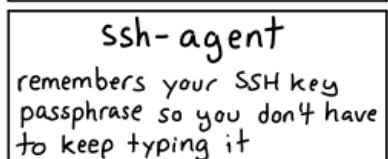
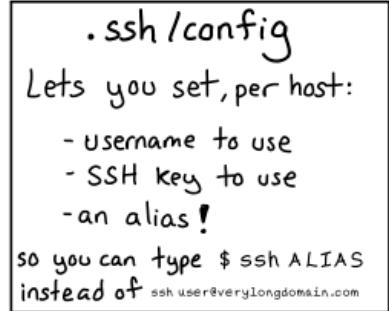
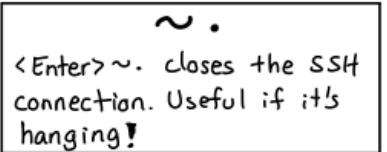
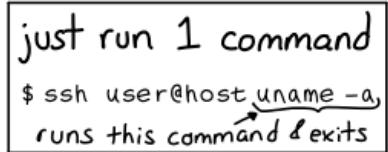
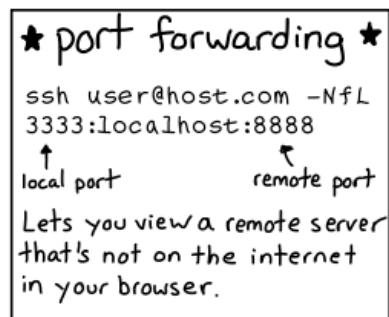
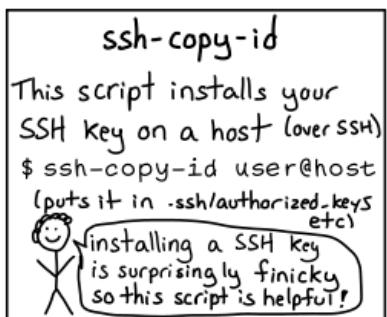
```
Connection to seawulf.uri.edu closed.
```

## 18.2. Creating SSH Keys

### 18.3. Using ssh keys to authenticated

1. generate a key pair
2. store the public key on the server
3. Request to login, tell server your public key, get back a session ID from the server
4. if it has that public key, then it generates a random string, encrypts it with your public key and sends it back to your computer
5. On your computer, it decrypts the message + the session ID with your private key then hashes the message and sends it back
6. the server then hashes its copy of the message and session ID and if the hash received and calculated match, then you are logged in

[a toy example](#)



[from wizardzines](#)

Lots more networking details in the [full zine](#) available for purchase or I have a copy if you want to borrow it.

this free zine describes networks at a lower level [full zine](#) but does not include ssh

## 18.4. Generating a Key Pair

We can use `ssh-keygen` to create a keys.

- `-f` option allows us to specify the file name of the keys.
- `-t` option allows us to specify the algorithm
- `-b` option allows us to specify the size

```
ssh-keygen -f ~/.ssh/seawulf -t rsa -b 1024
```

## 18.5. Sending the public key to a server

again `-i` to specify the file name

```
ssh-copy-id -i ~/.ssh/seawulf ayman_sandouk@seawulf.uri.edu
```

## 18.6. Logging in

To login without usng a password you have to tell ssh which key to use:

```
ssh -i ~/.ssh/seawulf ayman_sandouk@seawulf.uri.edu
```

Or you can add the following to your `~/.ssh/config` file

```
Host seawulf
  Hostname seawulf.uri.edu
  Username ayman_sandouk
  IdentityFile ~/.ssh/seawulf
```

and then use

```
ssh seawulf
```

## 18.7. Using SSH Keys

We are going to work on seawulf so that we all have the same compiler.

To use it we use he `-i` option and then the path to the private key file

```
ssh -i ~/seawulf ayman_sandouk@seawulf.uri.edu
```

```
Last login: Tue Apr  4 11:53:06 2023 from 172.20.207.131
```

## 18.8. Using an interactive session

Last class we worked on the login node, but that is not best practice.

Today we will use an interactive session using the `interactive` program.

The login node has limited resources, and is only usually used to connect to the server and possibly run batch jobs from there. The login node resources are shared among many users, so, out of courtesy, we try not to occupy these resources too much or for too long

First we'll look at its help

```
Usage: interactive [-c] [-p] [-J] [-w]

Optional arguments:
  -c: number of CPU cores to request (default: 1)
  -p: partition to run job in (default: general)
  -J: job name (default: interactive)
  -w: node name
```

NB: interactive jobs have a time limit of 8 hours.

Written by: Alan Orth <a.orth@cgiar.org>

Then we will start one with all default settings

```
[ayman_sandouk@seawulf ~]$ interactive
```

```
salloc: Granted job allocation 27371
salloc: Waiting for resource configuration
salloc: Nodes n005 are ready for job
```

it tells us what it is doing while it gets ready.

We see that the prompt changes, now we are `@n005` instead of `@seawulf`, lets check out working directory:

```
[ayman_sandouk@n005 ~]$ ls
```

|   |                   |
|---|-------------------|
| bash-lesson.tar.gz                          | SRR307026_1.fastq |
| dmel-all-r6.19.gtf                          | SRR307026_2.fastq |
| dmel_unique_protein_isoforms_fb_2016_01.tsv | SRR307027_1.fastq |
| gene_association.fb                         | SRR307027_2.fastq |
| SRR307023_1.fastq                           | SRR307028_1.fastq |
| SRR307023_2.fastq                           | SRR307028_2.fastq |
| SRR307024_1.fastq                           | SRR307029_1.fastq |
| SRR307024_2.fastq                           | SRR307029_2.fastq |
| SRR307025_1.fastq                           | SRR307030_1.fastq |
| SRR307025_2.fastq                           | SRR307030_2.fastq |

our files are still there. The *compute* node changed, but not our disk space location.

And our working directory is the same:

```
[ayman_sandouk@n005 ~]$ pwd
```

```
/home/ayman_sandouk
```

still our home directory

We will make an empty directory to work in for today.

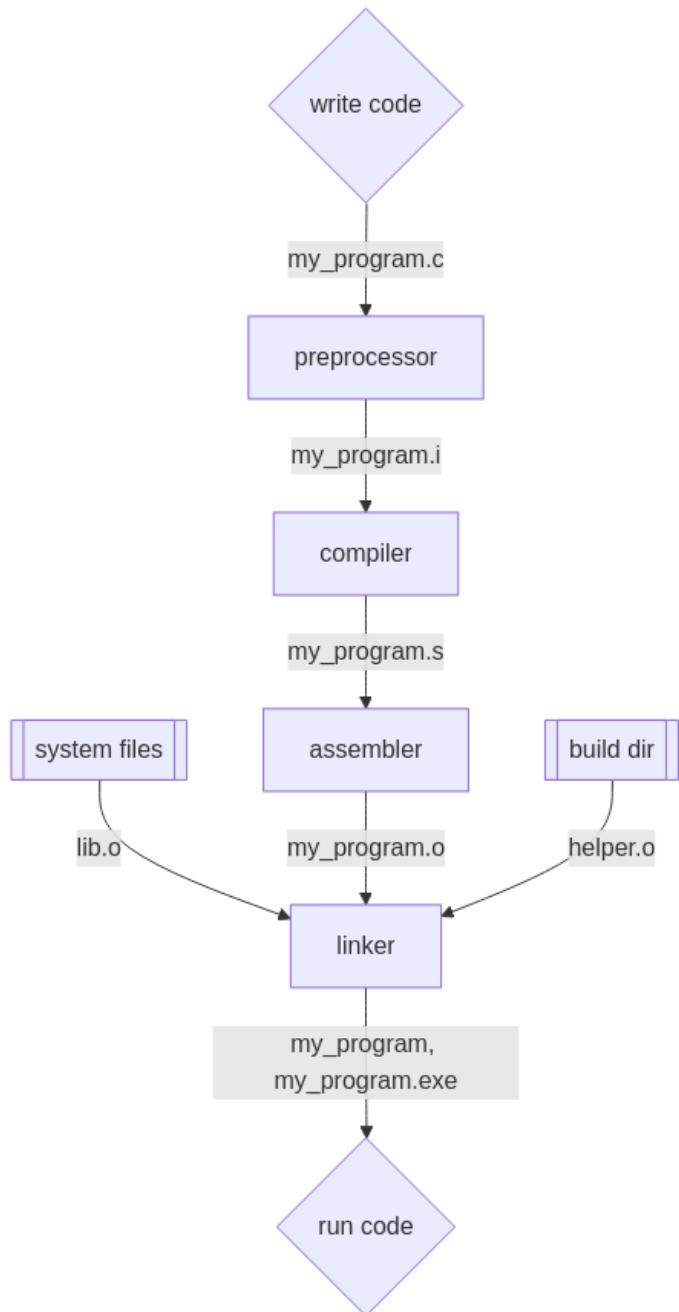
```
[ayman_sandouk@n005 ~]$ mkdir compilec
```

and go into it to work

```
[ayman_sandouk@n005 ~]$ cd compilec/
```

```
ls
```

## 18.9. An overview



## 18.10. An overview

## 18.11. A simple program

```
nano hello.c
```

and we will put in a simple hello world program

```
#include <stdio.h>
void main () {
    printf("Hello world\n");
}
```

We will see this is the only file in the folder

```
ls
```

```
hello.c
```

## 18.12. Preprocessing with gcc

First we handle the preprocessing which pulls in headers that are included in our source code. We will use the compiler [gcc](#)

We will use [gcc](#) for many steps. We will use different options that it offers to do subsets of the complete compile task:

- [-E](#) stops after preprocessing
- [-o](#) makes it write the .i file and passes the file name for it

```
gcc -E hello.c -o hello.i
```

If it succeeds, we see no output, but we can check the folder

```
ls
```

now we have a new file

```
hello.c hello.i
```

If we think that the [.i](#) file might be big, what can we use to compare the two to see the impact of preprocessing?

We can inspect what it does using [wc](#)

```
wc -l hello.c
```

```
6 hello.c
```

we started with just 6 lines of code

and we can compare that to the preprocessed file:

```
wc -l hello.i
```

```
842 hello.i
```

and see we get a lot more

```
#include <stdio.h>
void main () {
    printf("Hello world\n");
}
```

Since it is long, we will first look at the top

```
head hello.i
```

```
# 1 "hello.c"
# 1 "<built-in>"
# 1 "<command-line>"
# 1 "/usr/include/stdc-predef.h" 1 3 4
# 1 "<command-line>" 2
# 1 "hello.c"
# 1 "/usr/include/stdio.h" 1 3 4
# 27 "/usr/include/stdio.h" 3 4
# 1 "/usr/include/features.h" 1 3 4
# 375 "/usr/include/features.h" 3 4
```

and the end

```
tail hello.i
```

```
extern void funlockfile (FILE *__stream) __attribute__ ((__nothrow__ , __leaf__));
# 943 "/usr/include/stdio.h" 3 4
# 2 "hello.c" 2
void main () {
    printf("Hello world\n");
}
```

we see that our original program, is at the end of the file, and the beginning is where the include line has been expanded.

## 18.13. Compiling

Next we take our preprocessed file and compile it to get assembly code.

Again, we use `gcc`:

- `-S` tells it to produce assembly
- we will use the preprocessed file as input

```
gcc -S hello.i
```

but we can see what it output:

```
ls
```

```
hello.c hello.i hello.s
```

we have a new file as well with the `.s` extension. Notice how we didn't need to use the `-o` option this time

Again, lets inspect

```
wc -l hello.s
```

```
25 hello.s
```

this is longer than the source, but not as long as the header. The header contains lots of information that we *might* need, but the assembly is only what we *do*.

And it's manageable, so we inspect it directly:

```
cat hello.s
```

Try it

Look at  
the bas  
include  
preproc

Note

I have s  
highlight  
termina

```

.file    "hello.c"
.section .rodata
.LC0:
.string "Hello world"
.text
.globl main
.type   main, @function
main:
.LFB0:
.cfi_startproc
pushq %rbp
.cfi_def_cfa_offset 16
.cfi_offset 6, -16
movq %rsp, %rbp
.cfi_def_cfa_register 6
movl $.LC0, %edi
call puts
popq %rbp
.cfi_def_cfa 7, 8
ret
.cfi_endproc
.LFE0:
.size   main, .-main
.ident  "GCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)"
.section .note.GNU-stack,"",@progbits

```

There are many more steps. These are lower level operations, but they are still text stored in the file.

### 💡 Hint

Learning more about assembly languages is a good explore badge topic

## 18.14. Assembling

Assembling is to take the assembly code and get object code. Assembly is relatively broad and there are families of assembly code, it is also still written *for humans* to understand it readily. It's more complex than source code because it is closer to the hardware. The object code however, is specific instructions to your machine and not human readable.

Again, with `gcc`:

- `-c` tells it to stop at the object file
- `-o` again gives it the name of the file to write

```
gcc -c hello.s -o hello.o
```

Again, check what it does by looking at files

```
ls
```

```
hello.c hello.i hello.o hello.s
```

now we see a new file, the `.o`

and again check its length

```
wc -l hello.o
```

```
5 hello.o
```

this is even shorter,

we can check how many characters and words

```
wc hello.o
```

```
5 17 1496 hello.o
```

it is not even too many characters

let's look at it

```
cat hello.o
```

```
ELF>?@  
UH??]?Hello worldGCC: (GNU) 4.8.5 20150623 (Red Hat 4.8.5-44)zRx  
K A?  
?? hello.cmainputs  
  
???????? .symtab.strtab.shstrtab.rela.text.data.bss.rodata.comment.note.GNU-stack.rela.eh_frame @?0  
&PP1P  
90\.B?W?R@  
?  
?0a
```

This is not human readable, though:

## 18.15. Linking

Now we can link it all together; in this program there are not a lot of other dependencies, but this fills in anything from libraries and outputs an executable

once again with `gcc`:

- `-o` flag specifies the name for output
- `-lm` tells it to link from the .o file.

```
gcc -o hello hello.o -lm
```

```
ls
```

```
hello hello.c hello.i hello.o hello.s
```

Finally we can run our program

```
./hello
```

```
Hello world
```

## 18.16. Putting it all together

We can repeat with a different name and work directly from source to executable:

```
gcc -o demohello hello.c -lm
```

and run again.

```
./demohello
```

it runs the same

```
Hello world
```

if we link the object file

```
gcc -o hello hello.o -lm
```

and run

```
./hello
```

we see no change

```
Hello world
```

Now we can build completely from source that we edited and run

```
gcc -o hello hello.c -lm  
./hello
```

```
gcc --help
```

```
gcc -Wall -g -o hello hello.c -lm
```

 : When used, GCC includes extra data in the compiled output, such as symbol names, types, source file names, and line numbers. This information is crucial for debuggers like GDB, allowing developers to step through code, inspect variables, and understand the program's execution flow.

## 18.17. Working with multiple files

This all looks a bit different if we have our code split across files.

we will make a new file `main.c`

```
nano main.c
```

with content as follows

```
/* Used to illustrate separate compilation.  
Created: Joe Zachary, October 22, 1992  
Modified:  
*/  
  
#include <stdio.h>  
  
void main () {  
    int n;  
    printf("Please enter a small positive integer: ");  
    scanf("%d", &n);  
    printf("The sum of the first n integers is %d\n", sum(n));  
    printf("The product of the first n integers is %d\n", product(n));  
}
```

Then `help.c`

```
nano help.c
```

```

/* Used to illustrate separate compilation

Created: Joe Zachary, October 22, 1992
Modified:

*/

/* Requires that "n" be positive. Returns the sum of the
   first "n" integers. */

int sum (int n) {
    int i;
    int total = 0;
    for (i = 1; i <= n; i++)
        total += i;
    return(total);
}

/* Requires that "n" be positive. Returns the product of the
   first "n" integers. */

int product (int n) {
    int i;
    int total = 1;
    for (i = 1; i <= n; i++)
        total *= i;
    return(total);
}

```

Now we try again

```
gcc -Wall -g -c main.c
```

```

main.c:10:6: warning: return type of 'main' is not 'int' [-Wmain]
void main () {
^
main.c: In function 'main':
main.c:12:2: warning: implicit declaration of function 'sum' [-Wimplicit-function-declaration]
printf("The sum of the first n integers is %d\n", sum(n));
^
main.c:13:2: warning: implicit declaration of function 'product' [-Wimplicit-function-declaration]
printf("The product of the first n integers is %d\n", product(n));
^

```

we will leave only the one, which we will leave

Now we preprocess, compile and assemble the helper code:

```
gcc -Wall -g -c help.c
```

and look at what was created

```
ls
```

```

demohello  hello.c  hello.o  help.c  main.c
hello      hello.i  hello.s  help.o  main.o

```

## Tip

One reason we split code is to make it readable, but another reason is what we just did. We can compile each file separately, when your code is large and compiling takes a long time, splitting it will mean you only have to recompile the file(s) you have recently changed and relink, instead of recompiling everything.

and finally we link them.

```
gcc -o demo main.o help.o -lm
```

and then we can run

```
./demo
```

```
Please enter a small positive integer: 5
The sum of the first n integers is 15
The product of the first n integers is 120
```

```
ls
```

```
demo hello hello.c help.c help.o main.c main.o
```

```
Please enter a small positive integer: 6
The sum of the first n integers is 21
The product of the first n integers is 720
```

```
exit
```

Now we can modify the code

```
int main(int argc, char *argv[]) {
    int n = atoi(argv[1]);

    printf("The sum of the first %d integers is %d\n", n, sum(n));
    printf("The product of the first %d integers is %d\n", n, product(n));

    return 0;
}
```

Notice that now that we've fixed main.c we only need to recompile that. There is no need to recompile help.c since nothing changed in its code. We really just need the machine code of the new main and then we need to link both help.o with the new main.o. Let's call it main2.o

```
gcc -o main2.o -c main.c
gcc -o demo main2.o help.o -lm
```

and then we can run

```
./demo
```

```
Segmentation fault
```

We made the new code accept an integer argument directly from the terminal. So if we don't pass one, it tries to grab an unavailable value and fails like it did here

```
./demo 5
```

```
The sum of the first n integers is 15  
The product of the first n integers is 120
```

## 18.18. batch jobs

```
#!/bin/bash  
#SBATCH -t 1:00:00  
#SBATCH --nodes=1 --ntasks-per-node=1  
./single_job
```

## 18.19. Experience Report Evidence

## 18.20. Prepare for Next Class

TBD

## 18.21. Badges

**Review**

**Practice**

1. Create some variations of the `hello.c` we made in class. Make `hello2.c` print twice with 2 print commands. Make `hello5.c` print 5 times with a for loop and `hello7.c` print 7 times with a for loop. Build them all on the command line and make sure they run correctly.
2. Write a bash script, `assembly.sh` to compile each program to assembly and print the number of lines in each file.
3. use `scp` to download your modified main, script files, and output to your local computer and include them in your kwl repo.

## 18.22. Questions

Install [nand2tetris](#)

and make sure you can run the CPU Emulator

On Mac/Linux:

```
bash CPUEmulator.sh
```

On Windows: Double click on the CPUEmulator.bat file

## 19. What happens when we run code?

How are you today?

Are the explore options on the IDE practice badge helpful?

*pick the most appropriate first, but you can also free-response to expand*

- [ ] Yes, already submitted a proposal
- [ ] Yes, about to submit a proposal
- [ ] No, I'm still confused
- [ ] Kinda, those are not interesting to me, but useful examples
- [ ] You gave examples?
- [ ] N/A I do not plan explore badges

```
cd Documents/systems  
ls
```

```
example          seawulf  
github-in-class-brownaymanm-1 test  
kwl             testobj.md  
nand2tetris     tiny-book
```

### 19.1. on Mac/Linux

```
cd nand2tetris/tools
```

and then

```
bash CPUEmulator.sh
```

or in file explorer, Double click on the `CPUEmulator.bat` file

### 19.2. On Windows:

Double click on the CPUEmulator.bat file

**you can submit an issue to the coursewebsite if you figure out a CLI way to start it on windows**

## 19.3. To work on the online supported version

Follow this link for CPU emulator <https://nand2tetris.github.io/web-ide/cpu>

## 19.4. Starting the CPU Emulator

Linux/MacOS

Windows

In your terminal:

```
cd nand2tetris/tools
```

and then

```
bash CPUEmulator.sh
```

## 19.5. Loading a program

We're not going to *do* project 5, which is to **build** a CPU, but instead to use the test programs designed to check if a student's CPU works with the CPU Emulator they provide to *understand* a simple canonical CPU.

For more on how the emulator works see the [CPU Emulator Tutorial](#).

For much more detail about how this all works [chapter 4](#) of the related text book has description of the machine code and assembly language.

We're going to use the test cases from the book's project 5:

1. Load Program
2. Navigate to [nand2tetris/projects/05](#)

## 19.6. How does the computer actually add constants?

We'll use [add.hack](#) first.

This program adds constants, 2+3.

It is a program, assembly code that we are loading to the simulator's ROM, which is memory that gets read only by the CPU.

Run the simulator and watch what each line of the program does.

Notice the following:

- to compute with a constant, that number only exists in ROM in the instructions
- to write a value to memory the address register first has to be pointed to where in the memory the value will go, then the value can be sent there

The simulator has a few key parts:

- address register
- program counter

If you prefer to read, see [section 5.2.1- 5.2.6 of nan2tetris book](#)

- This program the first instruction puts 2 in the address register from the instructions in ROM.
- Then it moves the 2 through the ALU to the data register (D)
- then it puts 3 on the address register
- then it adds the numbers at D and A
- then it puts 0 on the address register
- then it write the output from the ALU (D) to memory (at the location indicated by the A register)

## 19.7. The max between two numbers

To compute the maximum value `max.hack`:

- subtracts the two numbers
- compares the difference to 0
- jumps to different lines based on if it is >0 or not
- assigns the address register a value accordingly
- copies one of the two values to the new location

## 19.8. Explore ideas

Create some experiments in the simulator and write new programs. Here are some ideas, but you can try anything you are curious about.

- What line do you change to change where it outputs the data?
- How could you add a third number?
- How could you add two pairs, saving the intermediate numbers?
- How could you do  $(4+4)*(3+2)$ ?

## 19.9. Why is the object file not human readable?

[https://learning.oreilly.com/library/view/beautiful-code/9780596510046/ch17.html#from\\_code\\_to\\_pointers](https://learning.oreilly.com/library/view/beautiful-code/9780596510046/ch17.html#from_code_to_pointers)

The object file displayed as random weird characters because it is written to disk in a different format than our terminal reads it in. It is the specific instructions (from the assembly) and memory addresses written to a file in binary. Our terminal reads it one byte (8bits) at a time and interprets those as characters.

Today, we will see how characters are represented as integers, then binary, and read and write files in binary of strings we know to see how it what happens and mimic something like how that object file happened by writing as binary and then reading it as characters.

## 19.10. Prelude: REPL

One way we can use many interpreted languages, including Python is to treat the interpreter like an application and then interact with it in the terminal.

This is called the Read, Execute, Print Loop or REPL.

We can execute simple expressions like mathematical expressions:

```
4+5
```

and it will print the result

```
9
```

We can create variables

```
name = 'ayman'
```

but assignment does not return anything, so nothing is printed.

We can see the value of a variable, by typing its name

```
name
```

and then it prints the value

```
'ayman'
```

### 19.10.1. Characters to Integers

Python provides a builtin function `ord` to find the Unicode key for a character.

This structure, the `[]` is called a list comprehension. It executes a `for` loop in order to build a `list`. In the REPL, it will then print out the list.

```
[ord(char) for char in name]
```

We get the integer representation for each one.

Note that it's important to try multiple things if you aren't sure the format you are getting back or to check the official documents, as Denno found by testing this with his name in lowercase.

```
[ord(char) for char in 'denno']
```



```
with open("name.txt", 'rb') as f:  
    print(f.read())
```

This is binary, but the python interpreter also *prints* it as a string, the **b** indicates the object is actually binary type.

```
b'ayman'
```

If we read it not as binary, it returns as the string

```
with open("name.txt", 'r') as f:  
    print(f.read())
```

```
ayman
```

We can also confirm the type

```
with open("name.txt", 'rb') as f:  
    content = f.read()
```

```
type(content)  
<class 'bytes'>
```

The with keyword is alternative to:

```
f = open("name.txt", 'rb')  
f.read()  
f.close()
```

Practice exercise

```
msg = [65, 121, 109, 97, 110, 32, 83, 97, 110, 100, 111, 117, 107, 10]
```

```
bytes(bytarray(msg))  
b'Ayman Sandouk\n'
```

## 19.11. Experience Report Evidence

Add in your experience report the modifications to the code as above.

## 19.12. Questions

1. Review your [idethoughts.md](#) from a few weeks ago and add some summary notes.
2. Think about what features or extensions in your favorite IDE you like the most and that you think others may not know about be prepared to share.

## 20. What is an IDE?

[quantum available for URI researchers](#)

### 20.1. Review your notes

#### Important

Do this before proceeding to the next section

Either discuss with peers in class or on the GitHub [discusstion](#) (asynch) discuss commonalities in your IDE notes.

#### 20.1.1. In person

1. What different tasks did all of you use an IDE for?
2. What features of an IDE did you all use?
3. Which features were used but not very much?
4. Share the most helpful IDE feature you use?

Update your individual IDE notes with 1-2 things you learned from your peers.

#### 20.1.2. Asynchronous

There are questions on the [GitHub Discussion](#). Update your individual IDE notes in your KWL repo with links to your post and replies.

## 20.2. Learn more

- [What is an IDE?](#)
- [compare IDEs](#)
- Most popular in [2021](#)
- Most popular in [2022](#)

#### 20.2.1. In person

In class with your peers you can divide these up and read one and then share key points with others.

With your group

- build a large list of IDE attributes or features that would be important. For example, share your favorite extensions and configurations
- make a table of how would you evaluate attribute? Which ones would you evaluate by just if it exists or not? Which ones would you evaluate in different degrees, what attributes of them would you evaluate?
- Discuss with your group how you would rank them. You do not all have to agree on a final ranking, but notice the differences.

#### vs code is open source

Summarize what you all discussed [on GitHub](#) for your classmates. Note the ranking, with any disagreements.

### 20.2.2. Asynchronous

After reading the above, also read at least 3 different articles about the “best IDE” for your favorite language or for multiple languages.

Notice what IDE attributes or features the authors think is important, and how they evaluate each criterion. Which ones are evaluated as present/missing? Which ones are evaluated in more detail.

Join the discussion on [GitHub](#) summarizing what you found the most important criteria to be and if you personally agree or not.

### 20.3. Experience Reports

For today, whether you are in class or asynchronous use the experience report (makeup) action. Use ISO date format: YYYY-MM-DD for the workflow input.

### 20.4. How do I choose a Programming Language for a project?

Today we'll explore how programming languages are categorized. Along the way, this will expose what core, generic features of a programming language are.

The key takeaway that I want for you is to have intuition for how to choose a language for a project, not only your favorite, but what is the best for different types of projects.

#### Tip

You should do either the [in groups](#) (if in class) or the [on your own](#) sections below for the x.x.x. For the x.x sections do all of them.

#### Important

Use the makeup workflow with an [ISO formatted date](#) (YYYY-MM-DD) if you work asynchronously. If in class you can do a regular experience badge.

## 20.5. Comparing languages you know

Tak a few minutes on your own to fill in the follwing table for two languages of your choice. Replace the <language> with two languages that you are familiar with. Add two additional rows. You can do this by memory, or by looking up/discussing. If you look things up, be sure to use reputable sources and include links to them.

In class you can work on this on a whiteboard or on your computer.

| feature           | <language1> | <language2> |
|-------------------|-------------|-------------|
| use of whitespace | Text        | Text        |
| list/array types  |             |             |
| variable typing   |             |             |
| memory usage      |             |             |

### 20.5.1. In groups

Share your table with some classmates and then discuss how they are similar and different.

Together, produce a list of questions for what other things you would want to know about how programming languages compare. All of you should include the collaboratively developed list in your experience report.

### 20.5.2. On your own

Think of a few questions about how you might compare programming languages.

Try to find high quality answers for them, from reputable sources. You can use an AI to find preliminary answers, but you need to find high quality sources to fact check and make sure you do not confuse yourself with incorrect information from a chatbot

#### Hint

Use a [Google Sheets](#) file (one per group) to collaboratively share the list of questions all to the same place. See one example below.

## 20.6. Learn more

Read the following

- What is the study of programming languages? [intro to PL](#)
- [why so many languages blog post](#)
- [feminism in programming language design](#)

### 20.6.1. In groups

You can divde the sources up and then discuss.

Discuss the readings, in particular:

- what you found most interesting
- was anything new to you?
- what do you want to remember most or learn more about?

## 20.6.2. On your own

Add notes on broader patterns below your table in your experience report based on reading the sources above.

Answer each of the following questions with specific references to the readings:

- what you found most interesting
- was anything new to you?
- what do you want to remember most
- what do you want to learn more about?

## 20.7. PL in Developer Survey

Resources:

- the most recent [developer survey// languages section](#)
- Additional reading about the languages from their official references if needed to answer the following questions.

### 20.7.1. With classmates

First discuss your prepare work then go back to the survey results to find more information and answer the following.

Discuss the findings to answer the following questions. This discussion should be at least 10-15 minutes including looking up information about different languages. As a group, you may divide and conquer this research.

- what is surprising?
- what did you expect?
- what do the popular languages have in common?
- what do the dreaded languages have in common?
- How are popular vs dreaded languages different? What features might be the cause for making a language dreaded?
- How do used languages differ from less commonly used languages? What features might be the cause for making a language popular?
- How have things changed since 2011?

Include (shared) notes from the discussion in your experience report. Reflect (individually) on a few key points (2-3 bullet points) from the discussion in your experience report for today.

### 20.7.2. On your own

Include answers to the following questions in your experience report for today.

- what is surprising?
- what did you expect?
- what do the popular languages have in common?
- what do the dreaded languages have in common?
- How are popular vs dreaded languages different? What features might be the cause for making a language dreaded?

- How to used languages differ from less commonly used languages? What features might be the cause for making a language popular?
- How have things changed since 2011?

Reflect on how you might use this information when deciding what language to learn next based on this information.

## 20.8. Experience Report Evidence

## 20.9. Prepare for Next Class

1. In fractionalbinary.md use 8 bits to represent the following numbers by using 4 bits as usual (8,2,4,1) and the other 4 bits are 1/2, 1/4, 1/8, 1/16th:

- 3.75
- 7.5
- 11.625
- 5.1875

1. Add to your file some notes about the limitations of representing non integer values this way. How much would using more bits help with, what limitations are not resolved by adding more bits. For example, how could you represent .1?

## 20.10. Review Today's Badges

### Important

Today's badges are [integrative](#) 2x badges.

## 20.11. Badges

[Review](#) [Practice](#)

### Important

This is an [integrative](#) 2x badge.

1. Try a new IDE and review it in newide.md. Your review should be 3 sections: Summary, Evaluation, and Reflection. Summary should be 1-3 sentences of your conclusions. Evaluation should be a detailed evaluation according to your group's criteria and one other group's criteria. In Reflection, reflect on your experience: What is easy? hard? What could you apply from the ones you already use? Were there features you had trouble finding?
2. Preview the [Stack Overflow Developer Survey](#) Technology section parts that are about tools. Create dev\_insights.md with 3-5 bullet points for discussion. These can be facts you found most interesting or questions you have based on the results (it can be clarifying or deeper questions)
3. Choose two languages from the [desired admired list](#). This could be a highly admired and least desired; it could be one with a small gap and one with a large gap. Read a few posts about each language and try to figure out why it is/not

admired or desired. Summarize your findings. Include links to all of the posts you read in a section titled `## Sources` in your markdown file. For each source, make a bulleted list with some notes about the author's background and any limitations that might put on the scope of their opinions. (for example, a data scientist's opinion on languages is very valuable for data science, but less for app development) Add this to your kwl repo in `language_love_dread.md`.

## 20.12. Questions

### KWL Chart

#### Working with your KWL Repo

##### Important

The `main` branch should only contain material that has been reviewed and approved by the instructors.

1. Work on a specific branch for each activity you work on
2. when it is ready for review, create a PR from the item-specific branch to `main`.
3. when it is approved, merge into main.

##### Tip

You ma  
on your

#### Minimum Rows

##### Warning

To be updated

#### Required Files

This lists the files for reference, but mostly you can keep track by badge issue checklists.

| date       | file   | type       |
|------------|--|------------|
| 2025-01-28 | brain.md   | /_practice |
| 2025-01-30 | gitoffline.md  | /_practice |
| 2025-01-30 | gitoffline.md  | /_review   |
| 2025-02-11 | branches-forks.md  | /_practice |
| 2025-02-11 | branches.md  | /_review   |
| 2025-02-13 | command-help.md  | /_practice |
| 2025-02-18 | commit-def.md  | /_review   |
| 2025-02-20 | terminal_organization_adv.md   | /_practice |
| 2025-02-20 | terminalpractice.md  | /_practice |
| 2025-02-20 | software.md  | /_prepare  |
| 2025-02-20 | terminal_review.md   | /_review   |
| 2025-02-25 | software.md` about how that project adheres to and deviates from the unix philosophy. Be specific, using links to specific lines of code or specific sections in the documentation that support your claims. |            |
| 2025-02-25 | Provide at least one example of both adhering and deviating from the philosophy and three total examples (that is 2 examples for one side and one for the other). You can see what badge `software.md        |            |
| 2025-03-06 | gitdef.md  | /_prepare  |
| 2025-03-06 | idethoughts.md   | /_prepare  |
| 2025-03-06 | favorite_git_workflow.md   | /_review   |
| 2025-03-06 | workflows.md   | /_practice |
| 2025-03-06 | workflows.md   | /_practice |
| 2025-03-18 | gitplumbingreview.md   | /_review   |
| 2025-03-18 | gitplumbingdetail.md   | /_practice |
| 2025-03-18 | gitislike.md   | /_practice |
| 2025-03-18 | hash_num_prep.md   | /_prepare  |
| 2025-03-25 | idethoughts.md   | /_prepare  |

| date       | file                   | type       |
|------------|------------------------|------------|
| 2025-03-25 | tagtypeexplore.md      | /_review   |
| 2025-03-25 | gitcounts.md           | /_review   |
| 2025-03-25 | tagtypes.md            | /_practice |
| 2025-03-25 | gitcounts_scenarios.md | /_practice |
| 2025-03-27 | octal.md               | /_practice |
| 2025-03-27 | octal.md               | /_review   |
| 2025-04-08 | newlanguage.md         | /_practice |
| 2025-04-08 | dev_insights.md        | /_review   |
| 2025-04-08 | settingssync.md        | /_practice |
| 2025-04-10 | fractionalbinary.md    | /_prepare  |

## Team Repo

### ⚠ Warning

We will not use this in spring 2024

## Contributions

Your team repo is a place to build up a glossary of key terms and a “cookbook” of “recipes” of common things you might want to do on the shell, bash commands, git commands and others.

For the glossary, follow the [jupyterbook](#) syntax.

For the cookbook, use standard markdown.

to denote code inline `use single backticks`

`to denote code inline `use single backticks``

to make a code block use 3 back ticks

````  
to make a code block use 3 back ticks  
````

To nest blocks use increasing numbers of back ticks.

To make a link, [\[show the text in squarebrackets\]\(url/in/parenthesis\)](#)

## Collaboration

You will be in a “team” that is your built in collaboration group to practice using Git Collaboratively.

There will be assignments that are to be completed in that repo as well. These activities will be marked accordingly. You will take turns and each of you is required to do the initialization step on a recurring basis.

This is also where you can ask questions and draft definitions to things.

## Peer Review

If there are minor errors/typos, suggest corrections inline.

In your summary comments answer the following:

- Is the contribution clear and concise? Identify any aspect of the writing that tripped you up as a reader.
- Are the statements in the contribution verifiable (either testable or cited source)? If so, how do you know they are correct?
- Does the contribution offer complete information? That is, does it rely on specific outside knowledge or could another CS student not taking our class understand it?
- Identify one strength in the contribution, and identify one aspect that could be strengthened further.

Choose an action:

- If the suggestions necessary before merging, select **request changes**.
- If it is good enough to merge, mark it **approved** and open a new issue for the broader suggestions.
- If you are unsure, post as a **comment** and invite other group members to join the discussion.

## Review Badges

### Review After Class

After each class, you will need to review the day’s material. This includes reviewing prismia chat to see any questions you got wrong and reading the notes. Review activities will help you to reinforce what we do in class and guide you to practice with the most essential skills of this class, they represent the minimum bar for C level work.

### Prepare for the next class

These tasks are usually not based on material that we have already seen in class. Mostly they are to have you start thinking about the topic that we are *about* to cover before we do so. Often this will include reviewing related concepts that you should have learned in a previous course (like pointers from 211) Getting whatever you know about the topic fresh in your mind in advance of class helps your brain get ready to learn the new material more easily; brains learn by making connections.

Other times prepare tasks are to have you install things so that you can engage in the class.

The correct answer is not as important for these activities as it is to do them **before class**. We will build on these ideas in class. These are evaluated on completion only<sup>[1]</sup>, but we may ask you questions or leave comments if appropriate, in that event you should reply and then we will approve.

[1] you will get full credit as long as all of the things are *done in good faith* even if not correct. However if it looks like you tried to outsource (eg to LLM) or plagiarize a solution, you will not earn credit for that.

## Practice Badges

### Note

these are listed by the date they were posted

Practice badges are a chance to first review the basics and then try new dimensions of the concepts that we cover in class. After each class, you will need to review the day's material. This includes reviewing prismia chat to see any questions you got wrong and reading the notes. The practice badge will also ask you to apply the day's material in a similar, but distinct way. They represent the minimum bar for B-level understanding.

## KWL File List

## Explore Badges

### Warning

Explore Badges are not required, but an option for higher grades. The logistics of this could be streamlined or the instructions may become more detailed during the penalty free zone.

Explore Badges can take different forms so the sections below outline some options. This page is not a cumulative list of requirements or an exhaustive list of options.

### Tip

You might get a lot of suggestions for improvement on your first one, but if you apply that advice to future ones, they will get approved faster.

## How do I propose?

Create an issue on your kwl repo, label it explore, and "assign" @AymanBx.

In your issue, describe the question you want to answer or topic to explore and the format you want to use. There is no real template for this, it can be as short as one sentence, but there may be follow up questions.

If you propose something too big, you might be advised to consider a build badge instead. If you propose something too small, you will get ideas as options for how to expand it and you pick which ones.

## Where to put the work?

- If you extend a more practice exercise, you can add to the markdown file that the exercise instructs you to create.
- If its a question of your own, add a new file to your KWL repo.
- If you do the work elsewhere, log it like a community badge but in a file called `external_explore_badges.md`

### Important

Either way, there must be a separate issue for this work that is also linked to your PR

## What should the work look like?

It should look like a blog post, written tutorial, graphic novel, or visual aid with caption. It will likely contain some code excerpts the way the class notes do. Style-wise it can be casual, like how you may talk through a concept with a friend or a more formal, academic tone. What is important is that it clearly demonstrates that you understand the material.

The exact length can vary, but these must go beyond what we do in class in scope

## Explore Badge Ideas:

- Extend a more practice:
  - for a more practice that asks you to describe potential uses for a tool, try it out, find or write code excerpts and examine them
  - for a more practice that asks you to try something, try some other options and compare and contrast them. eg “try git in your favorite IDE” -> “try git in three different IDEs, compare and contrast, and make recommendations for novice developers”
- For a topic that left you still a little confused or their was one part that you wanted to know more about. Details your journey from confusion or shallow understanding to a full understanding. This file would include the sources that you used to gather a deeper understanding. eg:
  - Describe how cryptography evolved and what caused it to evolve (i.e. SHA-1 being decrypted)
  - Learn a lot more about a specific number system
  - compare another git host
  - try a different type of version control
- Create a visual aid/memory aid to help remember a topic. Draw inspiration from [Wizard Zines](#)
- Review a reference or resource for a topic
- write a code tour that orients a new contributor to a past project or an open source tool you like.

Examples from past students:

- Scripts/story boards for tiktoks that break down course topics
- Visual aid drawings to help remember key facts

For special formatting, use [jupyter book's documentation](#).

 Note

## Build Badges

Build may be individual or in pairs.

These  
submit  
grading  
approx  
explore

## Proposal Template

If you have selected to do a project, please use the following template to propose a build

```
## < Project Title >

<!-- insert a 1 sentence summary -->

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will solve -->

### Method

<!-- describe what you will do , will it be research, write & present? will there be something you build -->

### Deliverables

<!-- list what your project will produce with target deadlines for each-->

### Milestones
```

The deliverables will depend on what your method is, which depend on your goals. It must be approved and the final submitted project will have to meet what is approved. Some guidance:

- any code or text should be managed with git (can be GitHub or elsewhere)
- if you write any code it should have documentation
- if you do experiments the results should be summarized
- if you are researching something, a report should be 2-4 pages, plus unlimited references in the 2 column [ACM format](#).

This guidance is generative, not limiting, it is to give ideas, but not restrict what you *can* do.

## Updates and work in Progress

These can be whatever form is appropriate to your specific project. Your proposal should indicate what form those will take.

## Summary Report

This summary report will be added to your kwl repo as a new file `build_report_title.md` where `title` is the (title or a shortened version) from the proposal. Use the template below for the summary report.

```

# <your project title> Summary Report

## Abstract
<!-- a one paragraph "abstract" type overview of what your project consists of. This should be written

## Reflection
<!-- a one paragraph reflection that summarizes challenges faced and what you learned doing your project

## Artifacts

<!-- links to other materials required for assessing the project. This can be a public facing web resour

```

## Collaborative Build rules/procedures

- Each student must submit a proposal PR for tracking purposes. The proposal may be shared text for most sections but the deliverables should indicate what each student will do (or be unique in each proposal).
- the proposal must indicate that it is a pair project, if iteration is required, I will put those comments on both repos but the students should discuss and reply/edit in collaboration
- the project must include code reviews as a part of the workflow links to the PRs on the project repo where the code reviews were completed should be included in the reflection
- each student must complete their own reflection. The abstract can be written together and shared, but the reflection must be unique.

## Build Ideas

### Your Profile (CV/Resume) Website

Use a static site generator, like one of the below.

#### Astro

- <https://astro.build>
- <https://docs.astro.build/en/getting-started/>
- [requires npm installation](#)

### General ideas to write a proposal for

- make a [vs code extension](#) for this class or another URI CS course
- port the courseutils to rust. [crate clap](#) is like the python click package I used to develop the course utils
- build a polished documentation website for your CSC212 project with [sphinx](#) or another static site generator
- use version control, including releases on any open source side-project and add good contributor guidelines, README, etc

### Auto-approved proposals

For these build options, you can copy-paste the template below to create your proposal issue and assign it to [@AymanBx](#).

## Add docs to another project

You can add documentation website to another project

```
## Project Docs

Add documentation website for <code proejct>.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will
This project will provide information for a user to use <the project> The information will live in the re

### Method

<!-- describe what you will do , will it be research, write & present? will there be something you build
1. ensure there is API level documentation in the code files
1. build a documentation website using [jupyterbook/ sphinx/doxygen/] that includes setup instructions ar
1. configure the repo to automatically build the documentation website each time the main branch is updat

### Deliverables

- link to repo with the contents listed in method in the reflection file

### Milestones

<!-- give a target timeline -->
```

## Developer onboarding

You can add documents that provide a developer onboarding experience to other code you have written

```

## Developer onboarding

Add developer onboarding information to <insert project title here>

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will solve. This project will provide information for a potential contributor to add new features to a code base. The goal is to make it easy for them to understand the context and requirements.

### Method

<!-- describe what you will do , will it be research, write & present? will there be something you build or test

1. ensure there is API level documentation in the code files
1. add a license, readme, and contributor file
1. add [code tours](https://marketplace.visualstudio.com/items?itemName=vscode-contrib.codetour) that help explain the code
1. set up a PR template
1. set up 2 issue templates: 1 for feature request and 1 for bug reporting

### Deliverables

- link to repo with the contents listed in method in the reflection file

### Milestones

<!-- give a target timeline -->

```

## Project Examples

- One type of project would be to do a research project on a topic we cover in class and author a report with your findings that demonstrates your knowledge of the topic. You must use developer-centric authoring tools, for example latex (eg with overleaf) or mystmd with github . The report would include an **Abstract**, **Body**, **Reflection** including what you did and what you learned from it, and a **Bibliography**. Potential research topics include:
  - Motherboards
  - CPUs: Their History, Evolution, and How They Work
  - GPUs: A Graphics Card That Revolutionized Machine Learning
  - The Differences Between Operating Systems: MacOS vs Windows VS Linux
  - Abstraction For Dummies: Explaining Abstract Concepts to the Layman
- Another type of project could be to create a program using the tools taught in class to maintain the program. What would be included in this would be a .md reporting your findings that demonstrates an understanding of the tools used and a link to the repository hosting the program including **documentation** written for the program.

## Syllabus and Grading FAQ

How much does activity x weigh in my grade?

How do I keep track of my earned badges?

Also, when are each badge due, time wise?

Who should I request to review my work?

Will everything done in the penalty free zone be approved even if there are mistakes?

Once we make revisions on a pull request, how do we notify you that we have done them?

What should work for an explore badge look like and where do I put it?

## Git and GitHub

I can't push to my repository, I get an error that updates were rejected

My command line says I cannot use a password

Help! I accidentally merged the Badge Pull Request before my assignment was graded









For an Assignment, should we make a new branch for every assignment or do everything in one branch?

Doing each new assignment [in](#) its own branch [is](#) best practice. In a typical software development flow once

## Other Course Software/tools

### Courseutils

This is how your badge issues are created. It also has some other utilities for the course. It is open source and questions/issues should be posted to its [issue tracker](#)

# Jupyterbook

## Changing paths on windows

To edit a path on windows, go to the search bar and type 'edit environment variables', click the environment variable button, click on 'path' then new, then insert the new path

## Avoiding windows security block

The closest thing to work around the security block is to exclude files, to exclude a file, take note of the file and know where to find it, go to windows security, virus protection and threat protection, scroll down to exclusions, add or exclude folders, then add the specific folder that is getting blocked

## Glossary

### Tip

Contributing glossary terms or linking to uses of glossary terms to this page is eligible for community badges

#### **absolute path**

the path defined from the root of the system

#### **add (new files in a repository)**

the step that stages/prepares files to be committed to a repository from a local branch

#### **argument**

input to a command line program

#### **bash**

bash or the bourne-again shell is the primary interface in UNIX based systems

#### **bitwise operator**

an operation that happens on a bit string (sequence of 1s and 0s). They are typically faster than operations on whole integers.

#### **branch**

a copy of the main branch (typically) where developmental changes occur. The changes do not affect other branches because it is isolated from other branches.

#### **Compiled Code**

code that is put through a compiler to turn it into lower level assembly language before it is executed. must be compiled and re-executed everytime you make a change.

#### **detached head**

a state of a git repo where the head pointer is set to a commit without a branch also pointing to the commit

## **directory**

a collection of files typically created for organizational purposes

## **divergent**

git branches that have diverged means that there are different commits that have same parent; there are multipe ways that git could fix this, so you have to tell it what strategy to use

## **fixed point number**

the concept that the decimal point does not move in the number. Cannot represent as wide of a range of values as a floating point number.

## **floating point number**

the concept that the decimal can move within the number (ex. scientific notation; you move the decimal based on the exponent on the 10). can represent more numbers than a fixed point number.

## **git**

a version control tool; it's a fully open source and always free tool, that can be hosted by anyone or used without a host, locally only.

## **GitHub**

a hosting service for git repositories

## **.gitignore**

a file in a git repo that will not add the files that are included in this .gitignore file. Used to prevent files from being unnecessarily committed.

## **git objects**

FIXME something (a file, directory) that is used in git; has a hash associated with it

## **git Plumbing commands**

low level git commands that allow the user to access the inner workings of git.

## **git Workflow**

a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner

## **HEAD**

a file in the .git directory that indicates what is currently checked out (think of the current branch)

## **merge**

putting two branches together so that you can access files in another branch that are not available in yours

## **merge conflict**

when two branches to be merged edit the same lines and git cannot automatically merge the changes

## **mermaid**

mermaid syntax allows user to create precise, detailed diagrams in markdown files.

## **hash function**

the actual function that does the hashing of the input (a key, an object, etc.)

## hashing

transforming an input of arbitrary length to a unique fixed length output (the output is called a hash; used in hash tables and when git hashes commits).

## integrated development environment

also known as an IDE, puts together all of the tools a developer would need to produce code (source code editor, debugger, ability to run code) into one application so that everything can be done in one place. can also have extra features such as showing your file tree and connecting to git and/or github.

## interpreted code

code that is directly executed from a high level language. more expensive computationally because it cannot be optimized and therefore can be slower.

## issue

provides the ability to easily track ideas, feedback, tasks, or bugs. branches can be created for specific issues. an issue is open when it is created. pull requests have the ability to close issues. see more in the [docs](#)

## Linker

a program that links together the object files and libraries to output an executable file.

## option

also known as a flag, a parameter to a command line program that change its behavior, different from an argument

## path

the “location” of a file or folder(directory) in a computer

## pointer

a variable that stores the address of another variable

## pull (changes from a repository)

download changes from a remote repository and update the local repository with these changes.

## [pull request](#)

allow other users to review and request changes on branches. after a pull request receives approval you can merge the changed content to the main branch.

## PR

short for [pull request](#)

## push (changes to a repository)

to put whatever you were working on from your local machine onto a remote copy of the repository in a version control system.

## relative path

the path defined **relative** to another file or the current working directory; may start with a name, includes a single file name or may start with `./`

**release**

a distribution of your code, related to a git tag

**remote**

a copy of the repository hosted on a server

**repository**

a project folder with tracking information in it in the form of a .git directory in it

**ROM (Read-Only Memory)**

Memory that only gets read by the CPU and is used for instructions

**SHA 1**

the hashing function that git uses to hash its functions (found to have very serious collisions (two different inputs have same hashes), so a lot of software is switching to SHA 256)

**sh**

abbr. see shell

**shell**

a command line interface; allows for access to an operating system

**ssh**

allows computers to safely connect to networks (such as when we used an ssh key to clone our github repos)

**templating**

templating is the idea of changing the input or output of a system. For instance, the Jupyter book, instead of outputting the markdown files as markdown files, displays them as HTML pages (with the contents of the markdown file).

**terminal**

a program that makes shell visible for us and allows for interactions with it

**tree objects**

type of git object in git that helps store multiple files with their hashes (similar to directories in a file system)

**yml**

see YAML

**YAML**

a file specification that stores key-value pairs. It is commonly used for configurations and settings.

**zsh**

zsh or z shell is built on top of the bash shell and contains new features

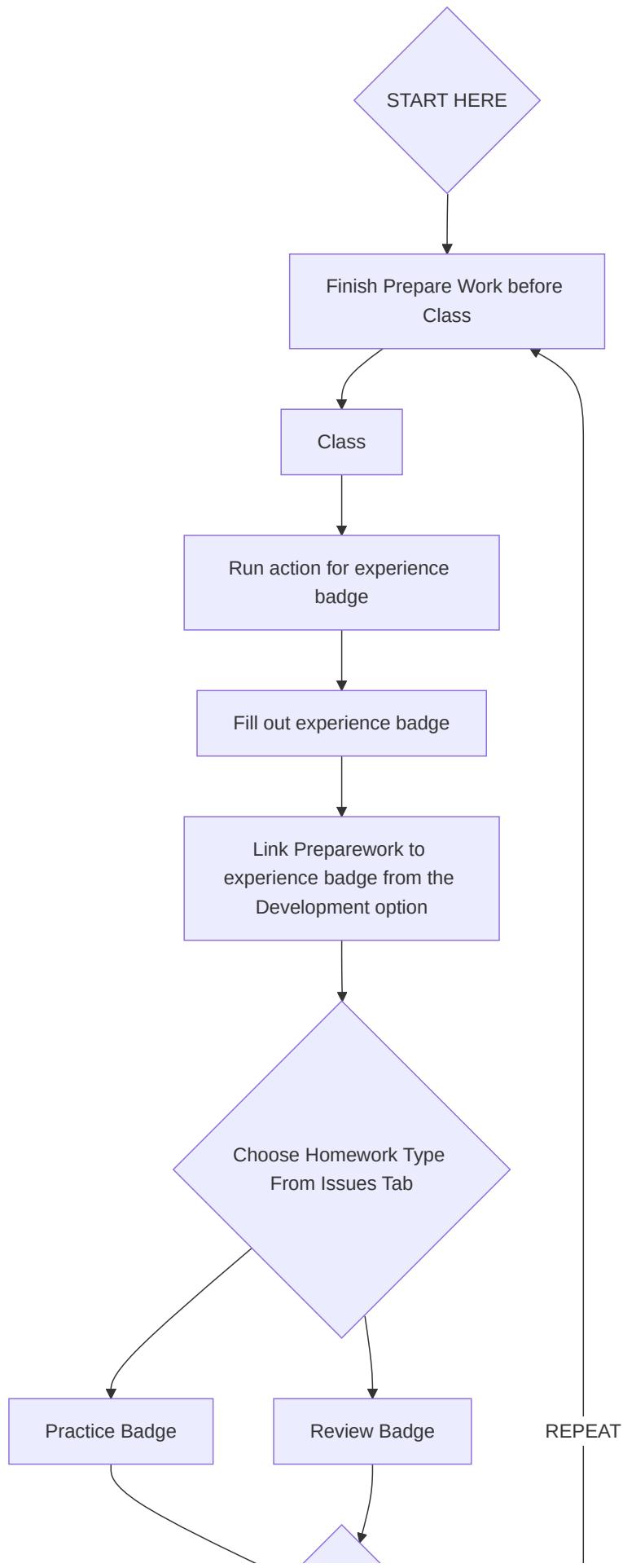
# General Tips and Resources

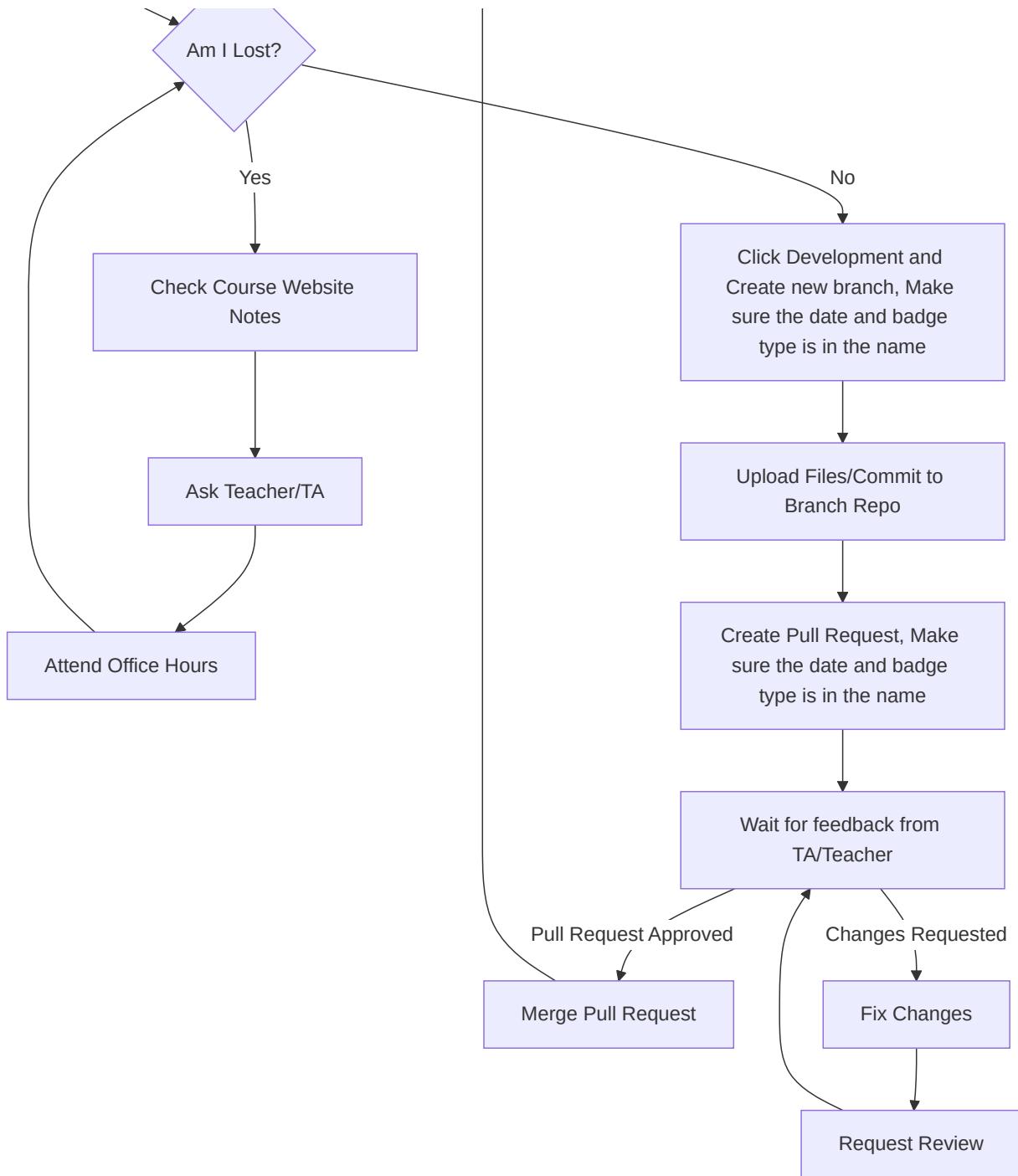
This section is for materials that are not specific to this course, but are likely useful. They are not generally required readings or installs, but are options or advice I provide frequently.

## on email

- [how to e-mail professors](#)

## Class Workflow





## How to Study in this class

In this page, I break down how I expect learning to work for this class.

Being a great programmer does not require memorizing all of the specific commands, but instead knowing the common patterns and how to use them to interpret others' code and write your own. Being efficient requires knowing how to use tools and how to let the computer do tedious tasks for you. This is how this course is designed to help you, but you have to get practice with these things.

Using reference materials frequently is a built in part of programming, most languages have built in help as a part of the language for this reason. These tools can help you when you are writing code and forget a specific bit of syntax, but these tools will not help you *read* code or debug environment issues. You also have to know how to effectively use these tools. Knowing the common abstractions we use in computing and recognizing them when they look a little bit differently will help

you with these more complex tasks. Understanding what is common when you move from one environment to another or to This course is designed to have you not only learn the material, but also to build skill in learning to program. Following these guidelines will help you build habits to not only be successful in this class, but also in future programming.

## Why this way?

Learning requires iterative practice. In this class, you will first get ready to learn by preparing for class. Then, in class, you will get a first experience with the material. The goal is that each class is a chance to learn by engaging with the ideas, it is to be a guided inquiry. Some classes will have a bit more lecture and others will be all hands on with explanation, but the goal is that you *experience* the topics in a way that helps you remember, because being immersed in an activity helps brains remember more than passively watching something. Then you have to practice with the material

Preparing for class will be activities that help you bring your prior knowledge to class in the most helpful way, help me mee

You will be making a lot of documentation of bits, in your own words. You will be directed to try things and make notes. This based on a recommended practices from working devs to [keep a notebook]](<https://blog.nelhage.com/2010/05/software-and-lab-notebooks/>) or [keep a blog and notebook](#).

## Learning in class

### Important

My goal is to use class time so that you can be successful with *minimal frustration* while working outside of class time.

Programming requires both practical skills and abstract concepts. During class time, we will cover the practical aspects and introduce the basic concepts. You will get to see the basic practical details and real examples of debugging during class sessions. Learning to debug something you've never encountered before and setting up your programming environment, for example, are *high frustration* activities, when you're learning, because you don't know what you don't know. On the other hand, diving deeper into options and more complex applications of what you have already seen in class, while challenging, is something I'm confident that you can all be successful at with minimal frustration once you've seen basic ideas in class. My goal is that you can repeat the patterns and processes we use in class outside of class to complete assignments, while acknowledging that you will definitely have to look things up and read documentation outside of class.

Each class will open with some time to review what was covered in the last session before adding new material.

To get the most out of class sessions, you should have a laptop with you. During class you should be following along with Dr. Brown. You'll answer questions on Prismia chat, and when appropriate you should try running necessary code to answer those questions. If you encounter errors, share them via Prismia chat so that we can see and help you.

## After class

After class, you should practice with the concepts introduced.

This means reviewing the notes: both yours from class and the annotated notes posted to the course website.

When you review the notes, you should be adding comments on tricky aspects of the code and narrative text between code

A new book  
programming  
[Programmer](#)  
available for  
that linked ta

blocks in markdown cells. While you review your notes and the annotated course notes, you should also read the documentation for new modules, libraries, or functions introduced in that class.

If you find anything hard to understand or unclear, write it down to bring to class the next day or post an issue on the course website.

## GitHub Interface reference

This is an overview of the parts of GitHub from the view on a repository page. It has links to the relevant GitHub documentation for more detail.

### Top of page

The very top menu with the  logo in it has GitHub level menus that are not related to the current repository.

### Repository specific page

| Code   | Issues | Pull Requests | Actions | Projects | Security | Insights | Settings |
|--|--------|---------------|---------|----------|----------|----------|----------|
| <p><b>This is the main view of the project</b></p> <p>Branch menu &amp; info, file action buttons, download options (green code button)</p> <p>About has basic facts about the repo, often including a link to a documentation page</p> <p>File panel</p> <p>the header in this area lists who made the last commit, the message of that commit, the short hash, date of that commit and the total number of commits to the project.</p> <p>If there are actions on the repo, there will be a red x or a green check to indicate that if it failed or succeeded on that commit.</p> <p>Releases, Packages, and Environments are optional sections that the repo owner can toggle on and off.</p> <p><a href="#">Releases</a> mark certain commits as important and give easy access to that version. They are related to <a href="#">git tags</a></p> <p><a href="#">Packages</a> are out of scope for this course. GitHub helps you manage distributing your code to make it easier for users.</p> <p><a href="#">Environments</a> are a tool for dependency management. We will cover thigns that</p> <p>the header in this area lists who made the last commit, the message of that commit, the short hash, date of that commit and the total number of commits to the project.</p> |        |               |         |          |          |          |          |

If there are actions on the repo, there will be a red x or a green check to indicate that if it failed or succeeded on that commit. ^^^ file list: a table where the first column is the name, the second column is the message of the last commit to change that file (or folder) and the third column is when is how long ago/when that commit was made

README file

help you know how to use this feature indirectly, but probably will not use it directly in class. This would be eligible for a build badge.

The bottom of the right panel has information about the languages in the project

## Language/Shell Specific References

- [bash](#)
- [C](#)
- [Python](#)

### Bash commands

| command   | explanation  |
|---|--|
| <code>pwd</code>                                  | print working directory  |
| <code>cd &lt;path&gt;</code>                      | change directory to path   |
| <code>mkdir &lt;name&gt;</code>                   | make a directory called name   |
| <code>ls</code>                                   | list, show the files   |
| <code>touch</code>                                | create an empty file   |
| <code>echo 'message'</code>                       | repeat 'message' to stdout   |
| <code>&gt;</code>                                 | write redirect   |
| <code>&gt;&gt;</code>                             | append redirect  |
| <code>rm file</code>                              | remove (delete) <code>file</code>  |
| <code>cat</code>                                  | concatenate a file to standard out (show the file contents)                                |
| <code>mv &lt;old_path&gt; &lt;new_path&gt;</code> | Moves file from path to another (can be used to rename only)                               |
| <code>cp &lt;file&gt; &lt;path&gt;</code>         | Copies the content of the file from into a new file (can be with or without the same name) |

# git commands

| command  | explanation  |
|--|--|
| <code>status</code>                            | describe what relationship between the working directory and git                         |
| <code>clone &lt;url&gt;</code>                 | make a new folder locally and download the repo into it from url, set up a remote to url |
| <code>add &lt;file&gt;</code>                  | add file to staging area   |
| <code>commit -m 'message'</code>               | commit using the message in quotes   |
| <code>push</code>                              | send to the remote   |
| <code>git log</code>                           | show list of commit history  |
| <code>git branch</code>                        | list branches in the repo  |
| <code>git branch new_name</code>               | create a <code>new_name</code> branch  |
| <code>git checkout -b new_Name</code>          | create a <code>new_name</code> branch and switch to it                                   |
| <code>git pull</code>                          | apply or fetch and apply changes from a remote branch to a local branch                  |
| <code>git commit -a -m 'msg'</code>            | the <code>-a</code> option adds modified files (but not untracked)                       |
| <code>git restore &lt;file&gt;</code>          | Undo changes made to file since last commit  |
| <code>git restore --staged &lt;file&gt;</code> | Remove file from staging area to be committed without undoing the changes done to it     |

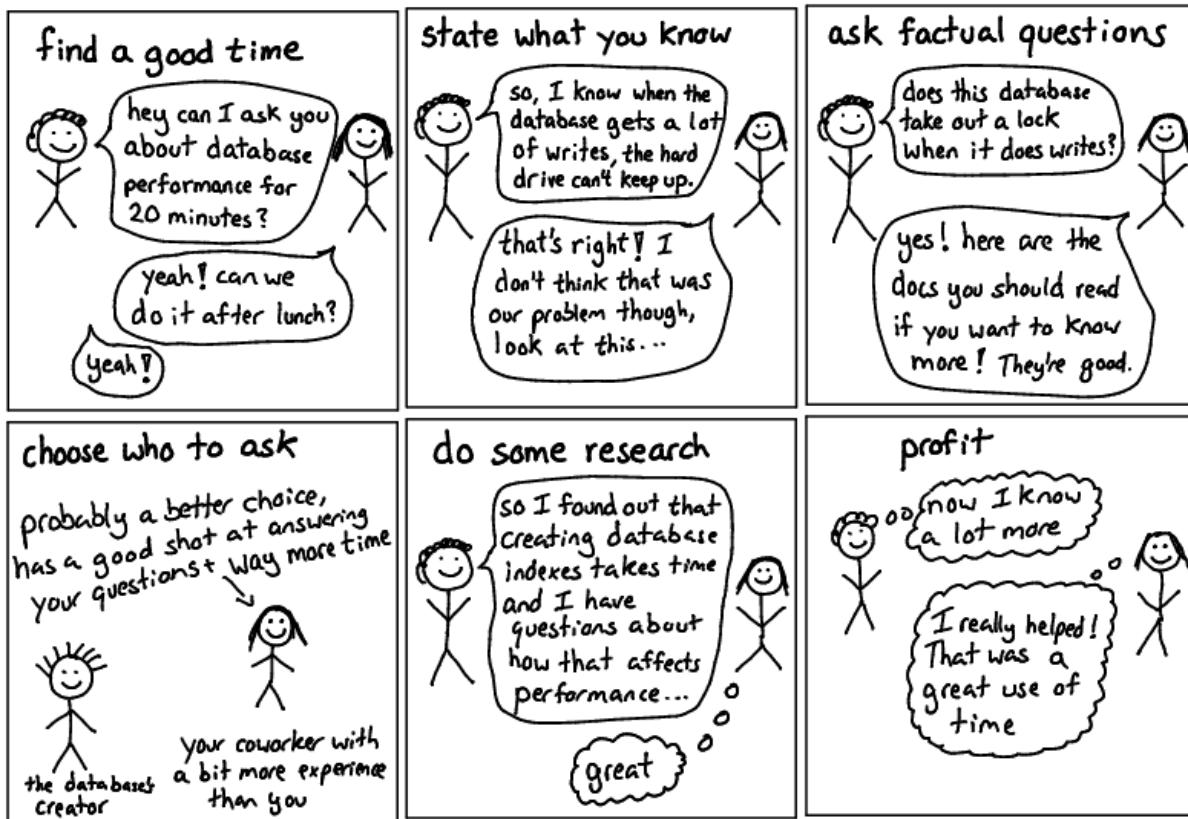
## Getting Help with Programming

This class will help you get better at reading errors and understanding what they might be trying to tell you. In addition here are some more general resources.

# Asking Questions

JULIA EVANS  
@b0rk

## asking good questions



One of my favorite resources that describes how to ask good questions is [this blog post](#) by Julia Evans, a developer who writes comics about the things she learns in the course of her work and publisher of [wizard zines](#).

## Describing what you have so far

Stackoverflow is a common place for programmers to post and answer questions.

As such, they have written a good [guide on creating a minimal, reproducible example](#).

Creating a minimal reproducible example may even help you debug your own code, but if it does not, it will definitely make it easier for another person to understand what you have, what your goal is, and what's working.

i Note

A fun v  
debugg

## Getting Organized for class

The only **required** things are in the Tools section of the syllabus, but this organizational structure will help keep you on top of what is going on.

Your username will be appended to the end of the repository name for each of your assignments in class.

## File structure

I recommend the following organization structure for the course:

```

CSC3392
|- kwl-
|- gh-inclass
|- semYYYY
|- ...

```

This is one top level folder will all materials in it. A folder inside that for in class notes, and one folder per repository.

Please **do not** include all of your notes or your other assignments all inside your portfolio, it will make it harder to grade.

## Finding repositories on github

Each assignment repository will be created on GitHub with the `compsys-progtools` organization as the owner, not your personal account. Since your account is not the owner, they do not show on your profile.

If you go to the main page of the [organization](#) you can search by your username (or the first few characters of it) and see only your repositories.

## More info on cpus

| Resource  | Level | Type    | Summary   |
|---|-------|---------|---|
| <a href="#">What is a CPU, and What Does It Do?</a> | 1     | Article | Easy to read article that explains CPUs and their use. Also touches on "buses" and GPUs.        |
| <a href="#">Processors Explained for Beginners</a>  | 1     | Video   | Video that explains what CPUs are and how they work and are assembled.                          |
| <a href="#">The Central Processing Unit</a>         | 1     | Video   | Video by Crash Course that explains what the Central Processing Unit (CPU) is and how it works. |

## Windows Help & Notes

### CRLF Warning

This is GitBash telling you that git is helping. Windows uses two characters for a new line `CR` (carriage return) and `LF` (line feed). Classic Mac Operating system used the `CR` character. Unix-like systems (including MacOS X) use only the `LF` character. If you try to open a file on Windows that has only `LF` characters, Windows will think it's all one line. To help you, since git knows people collaborate across file systems, when you check out files from the git database (`.git/` directory) git replaces `LF` characters with `CRLF` before updating your working directory.

When working on Windows, when you make a file locally, each new line will have `CRLF` in it. If your collaborator (or server, eg GitHub) runs not a unix or linux based operating system (it almost certainly does) these extra characters will make a mess and make the system interpret your code wrong. To help you out, git will automatically, for Windows users, convert `CRLF` to `LF` when it adds your work to the index (staging area). Then when you push, it's the compatible version.

[git documentation of the feature](#)

## Jupyter Book - Issues During or After Installation

### 1. Check Python Installation:

Run `python --version`. If it shows a version, continue. If not, install Python from <https://www.python.org/downloads/>. If you get a “Permission Denied” message, see the “Adding Permissions” section below. If you know python is installed, see the “Checking Paths” section below.

### 2. Install Jupyter Book:

Ensure Jupyter Book is installed using `pip install -U jupyter-book`. If `jupyter-book --version` returns then “command not found,” see “Checking Paths” below. If you get a “Permission Denied” message, see the “Adding Permissions” section below.

### 3. Check Installation Errors:

If there were no errors during installation, skip to Step 4.

If there were errors with a path (e.g., missing “Scripts” folder), see the “Checking Path” section.

### 4. Check for Directory:

Ensure the following directories exist: (Can check through File Explorer or through terminal)

```
C:\Users\[YOUR USERNAME]\AppData\Roaming\Python\Python[VERSION#]\  
C:\Users\[YOUR USERNAME]\AppData\Roaming\Python\Python[VERSION#]\Scripts\
```

If it does, move on. If not, ensure that Python was installed correctly from the website download, NOT the Windows Store.

### 5. Final Troubleshooting:

If issues persist, contact your Professor or TA for help! :)

## Checking Paths

If you get a `Command Not Found` message when trying to run `python` or `pip`, most likely your environment variable paths missing. First ensure the following directories exist: (Can check through File Explorer or through terminal)

```
C:\Users\[YOUR USERNAME]\AppData\Roaming\Python\Python[VERSION#]\  
C:\Users\[YOUR USERNAME]\AppData\Roaming\Python\Python[VERSION#]\Scripts\
```

If they do, there are two methods to ensuring the path variables exist and adding them if not:

### Method 1

1. Go to your taskbar Search
2. Search for and open “Edit the system environment variables”
3. Click “Environment Variables...” in the window that opened
4. Click “Path” line then “Edit...”

### Method 2

1. Press `Windows + R`, type `sysdm.cpl`, and press Enter.
2. Go to the **Advanced** tab → **Environment Variables**.
3. Add the path containing the “Scripts” folder to the `Path` variable. Save and exit.

4. Reopen Git Bash and try `jupyter-book --version`. If it works, you're done!
  - If “Access denied” occurs, try `sudo jupyter-book --version`. Windows Defender may prompt you to unlock the file. After unlocking, try the command again.

## Adding Permissions

If you get a `Permission Denied` message when trying to run commands, Windows Security is blocking it.

- If you get a pop-up notification when trying to run a command, simply click “unblock” each time
  - If you don't get a pop-up notif, follow the steps below to add an exclusion for Python
1. Go to your taskbar Search
  2. Search for and open “Windows Security”
  3. Go to the “Virus & threat protection” section on the left
  4. Under “Virus & threat protection settings” click “manage settings”
  5. Scroll down to the “Exclusions” section and click “Add or remove exclusions”
  6. Click “Add an exclusion”, then “Folder”
  7. Find and select the folder Python installed in
    - Should be `C:\Users\[YOUR_USERNAME]\AppData\Roaming\Python\`