

# 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

### 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 clode 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

## 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.

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

See  
req  
SS  
cou

# 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
- completing an “experience report”
- attaching evidence as indicated 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 procedure 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 of 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.

### 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?

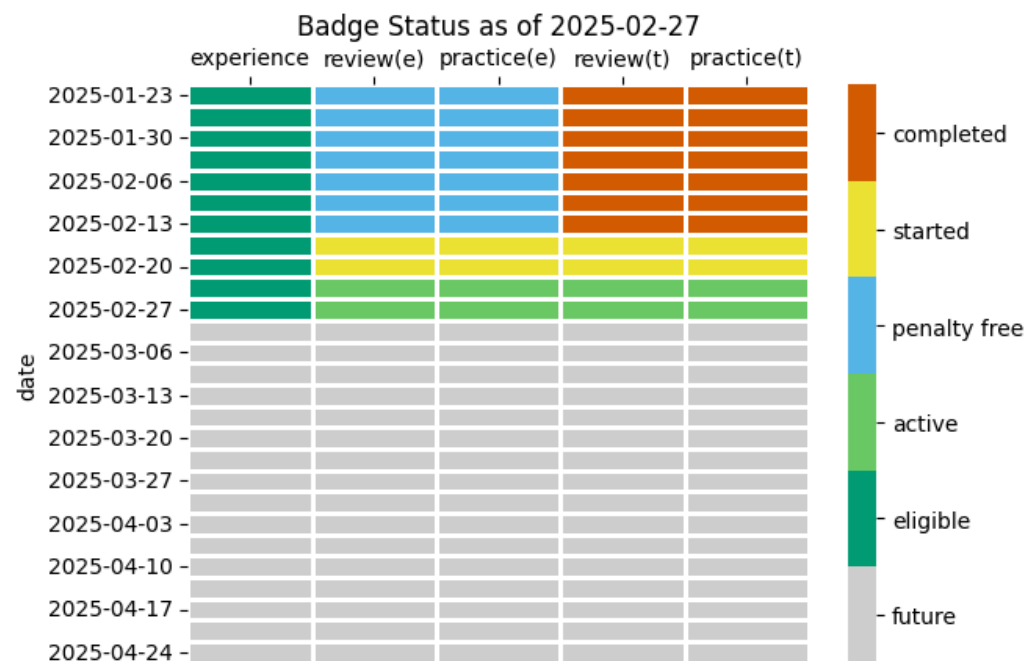
share your thoughts on this option [in the discussions for the class](#) and then log it for a community badge!

## 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 2025-02-27')



# Getting Feedback

Who should you request/assign?

course item type	issue assignee	PR reviewer
prepare work	not required; can be student	none required; 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	@instuctors (will then convert to 1/3 people)
review badge	not required; can be student	@instuctors (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.

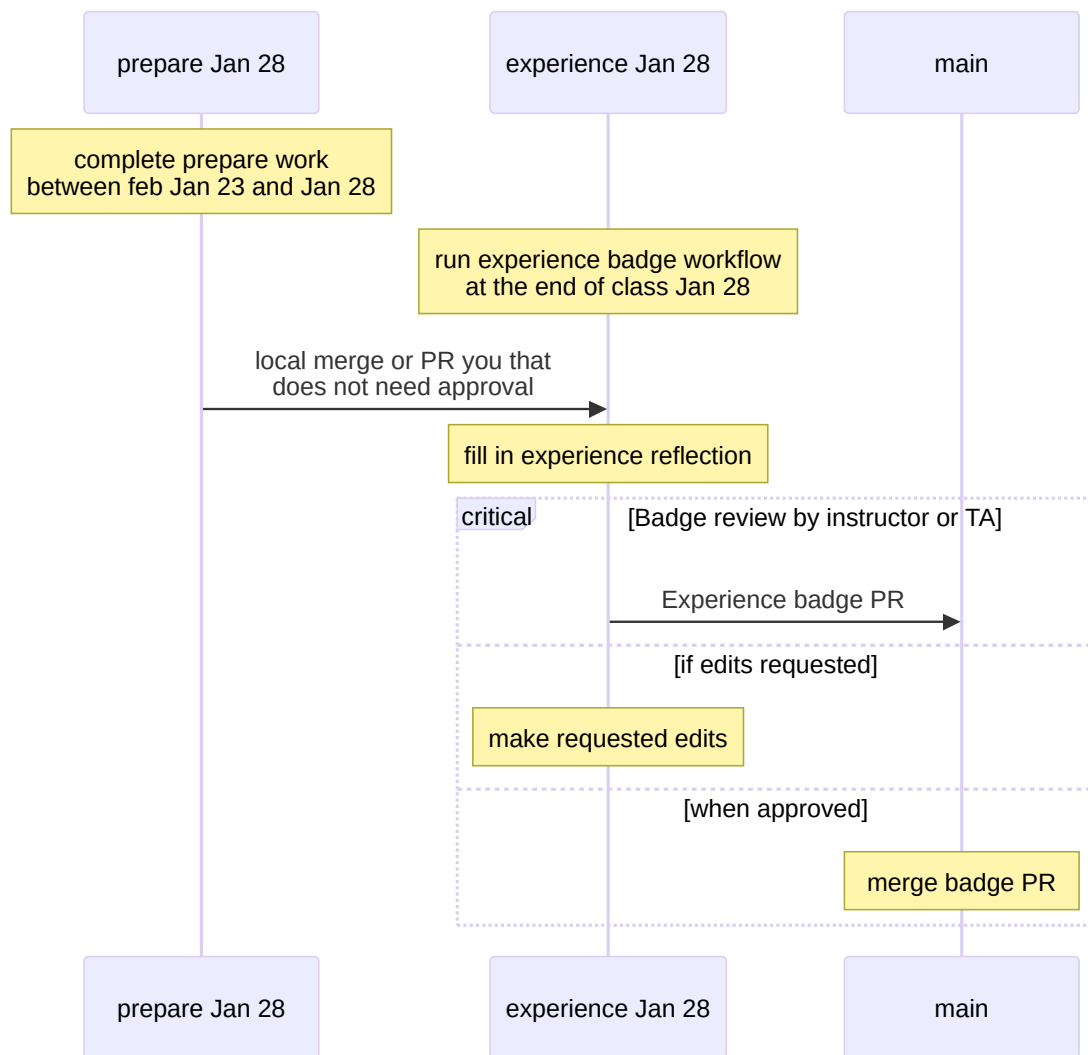
## Procedures

### Prepare work and Experience Badges Process

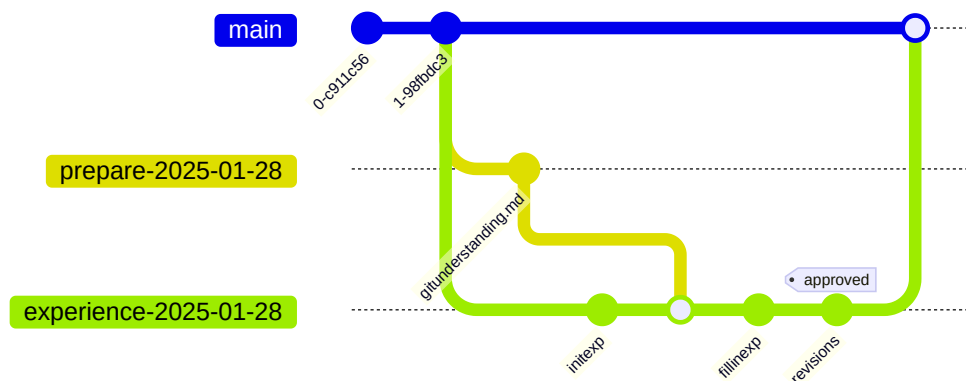
This is for a single example with specific dates, but it is similar for all future dates

The columns (and purple boxes) correspond to branches in your KWL repo and the yellow boxes are the things that 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:



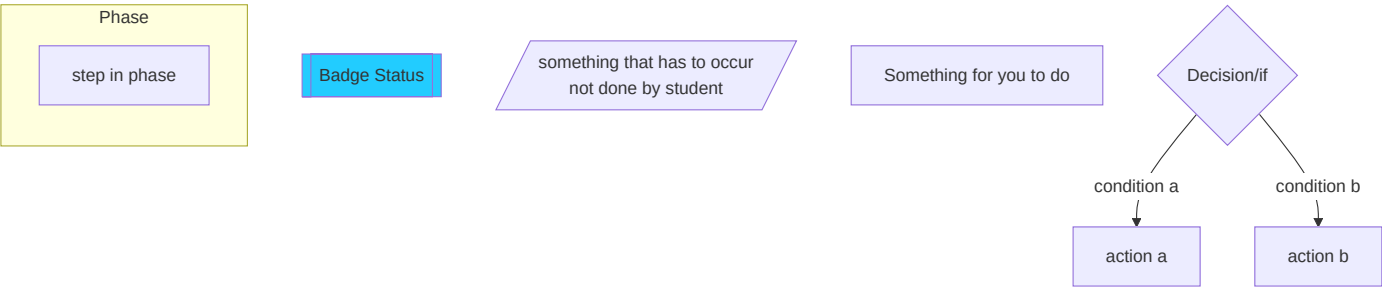
You can merge the prepare into the experience with a PR or on the command line, your choice.

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

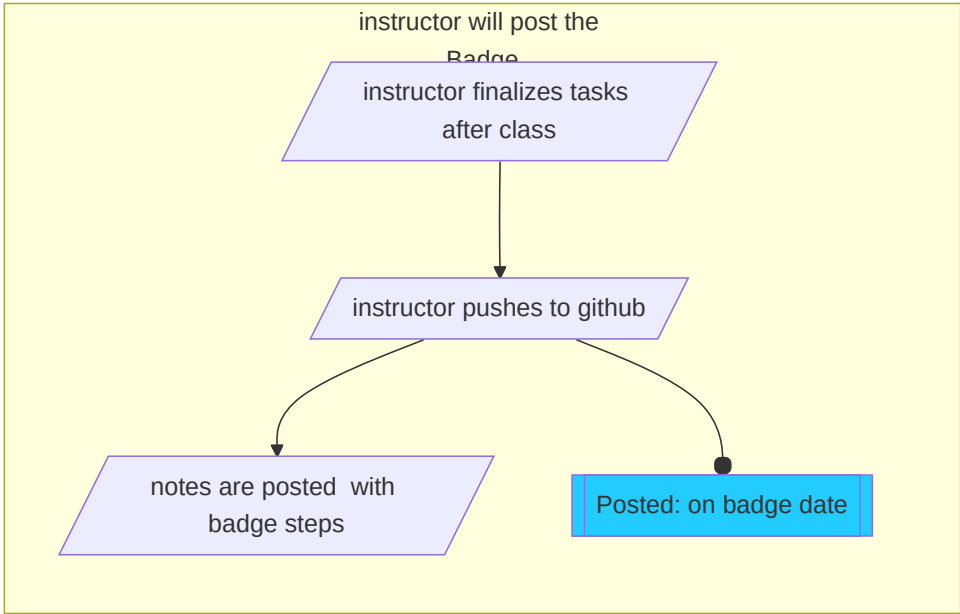
You can, once you know how, do this offline and do the merge with in the CLI instead of with a PR.

## Review and Practice Badge

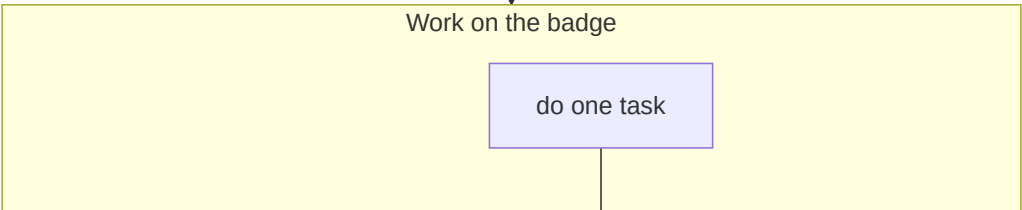
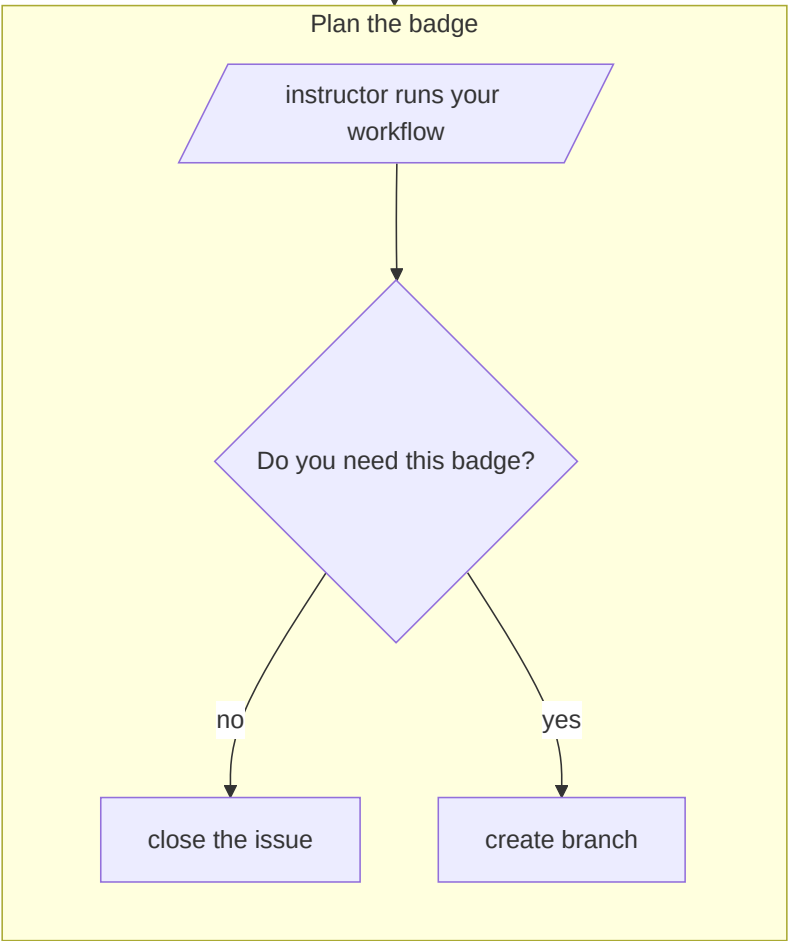
Legend:

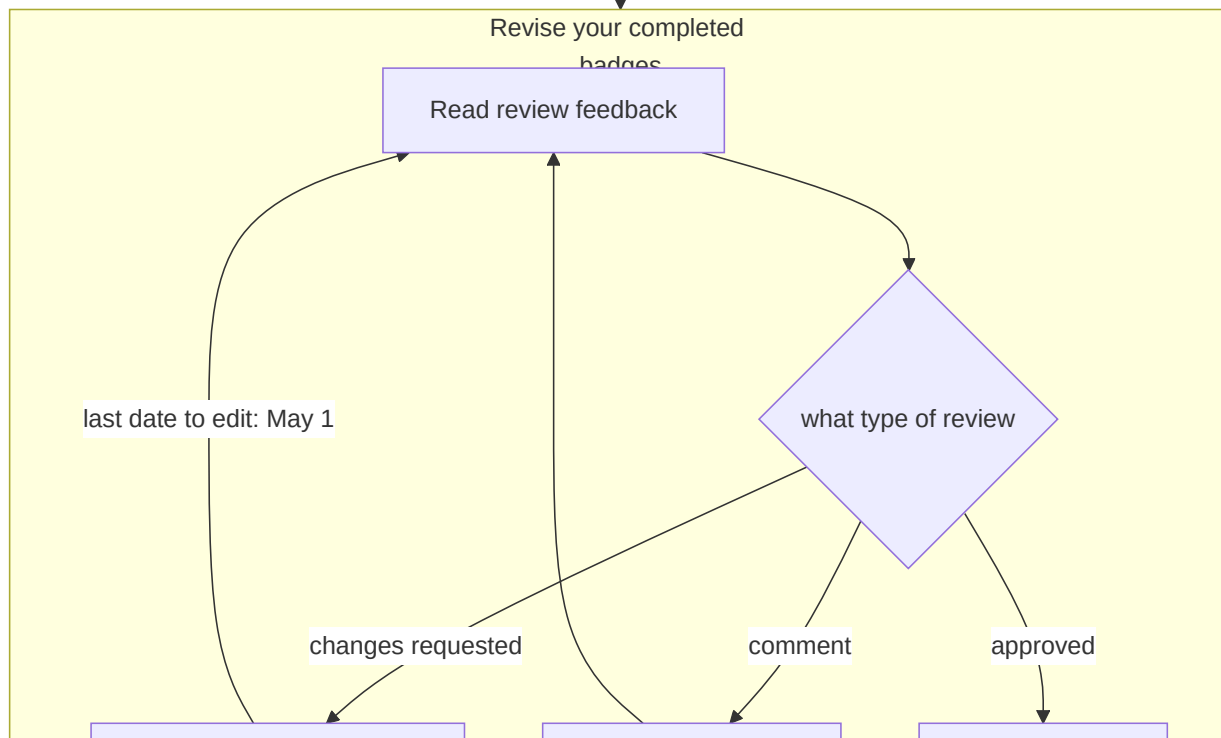
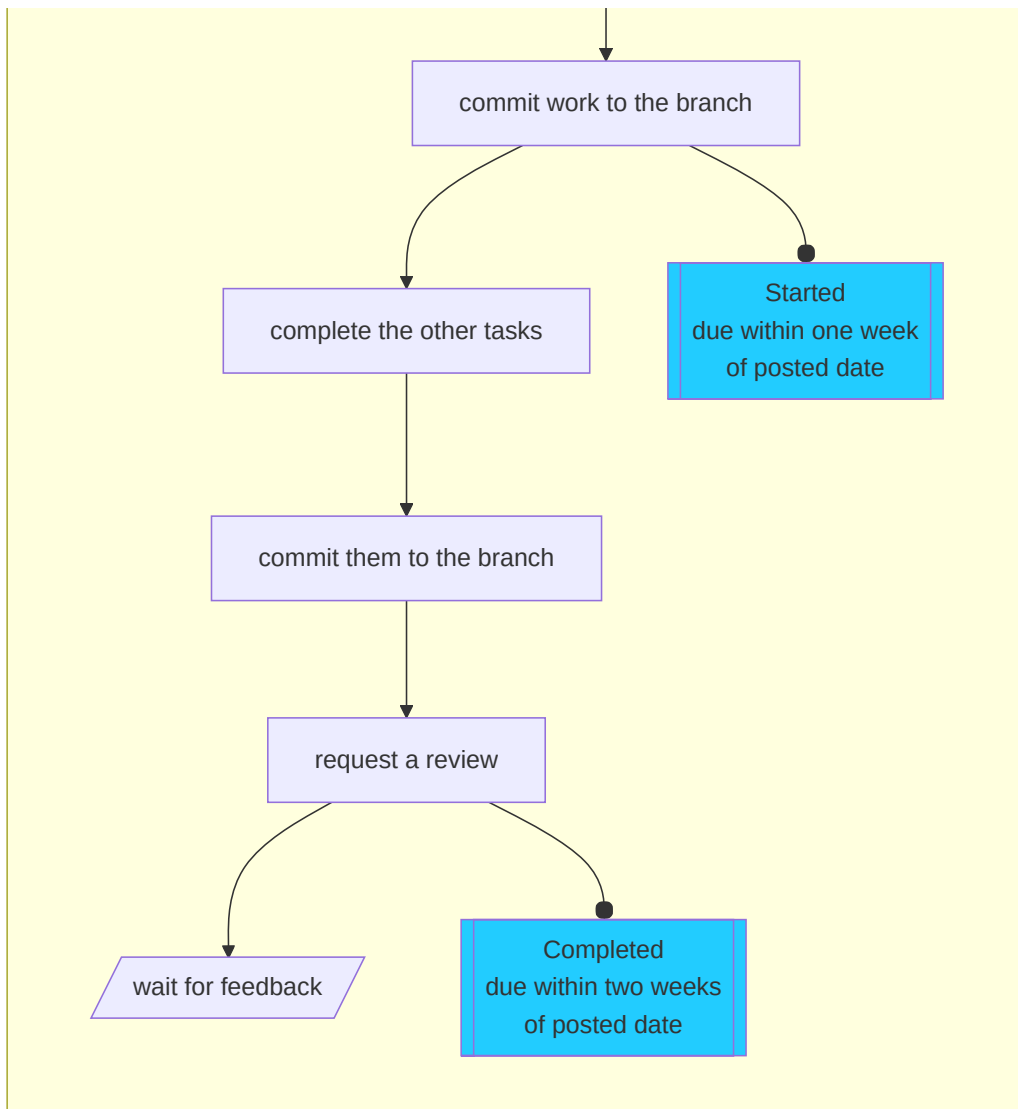


This is the general process for review and practice badges



planned





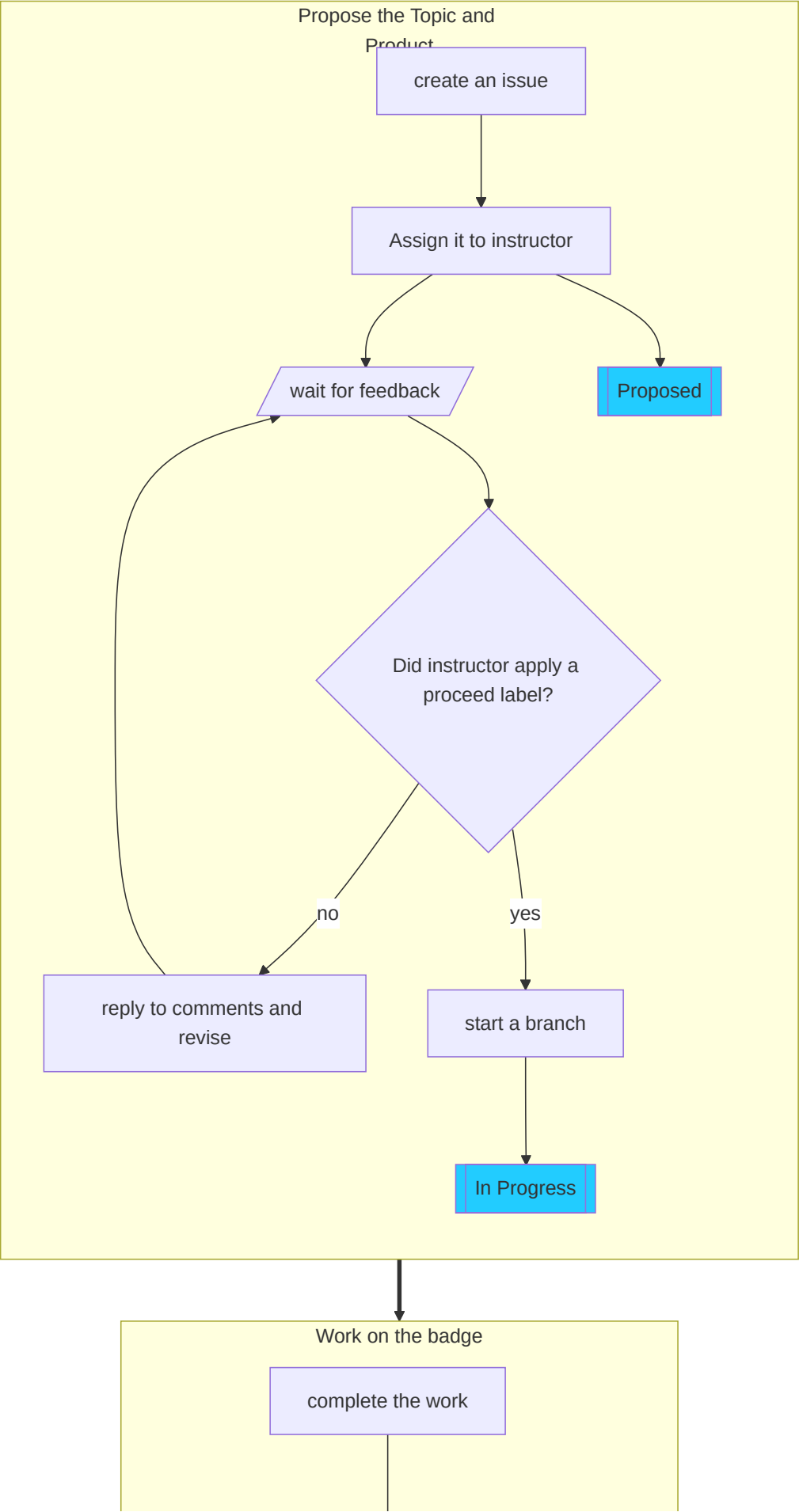
complete requested edits

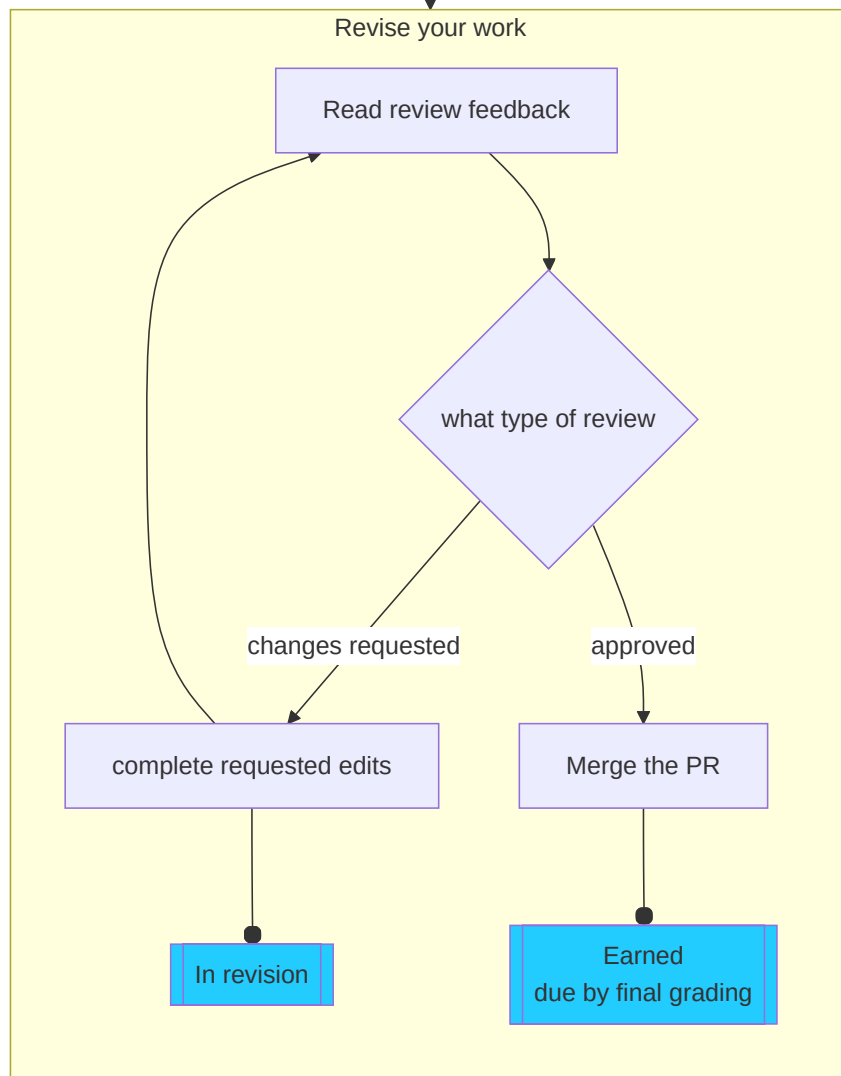
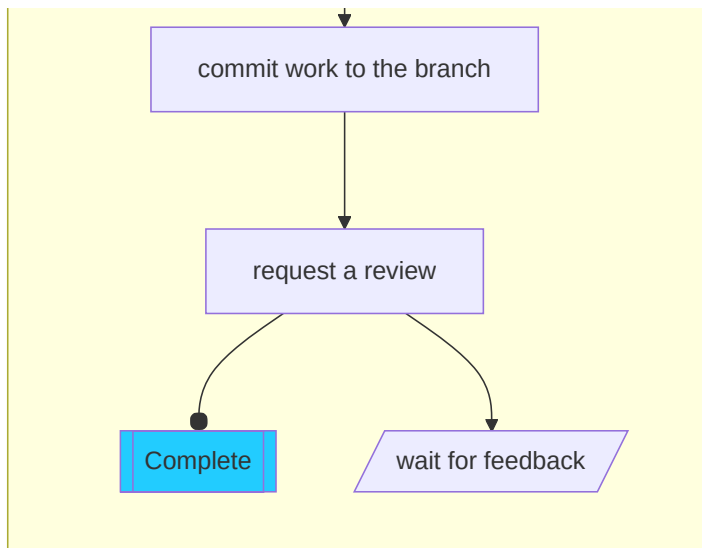
reply to comments

Merge the PR

Earned  
due by final grading

Explore Badges

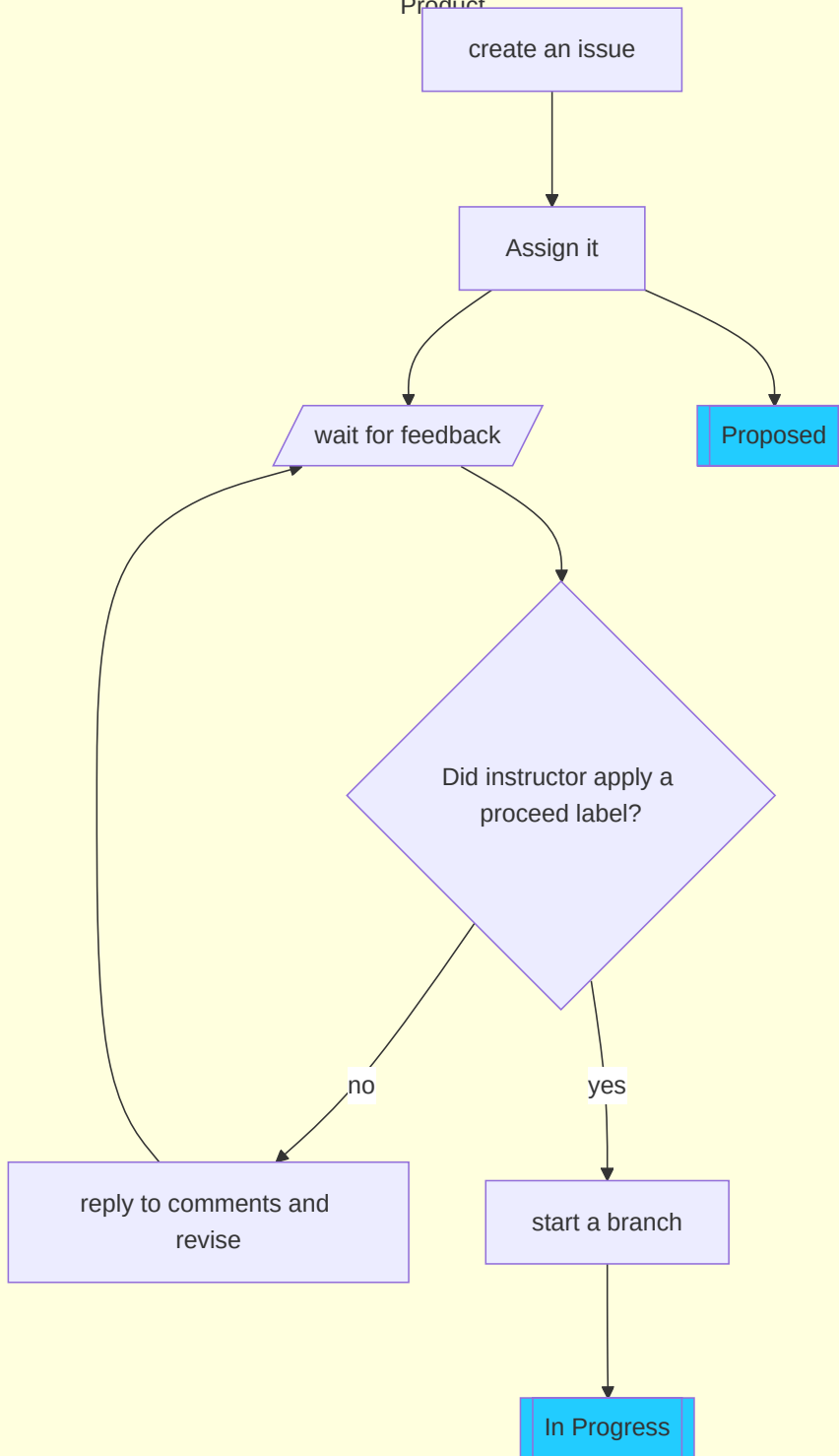






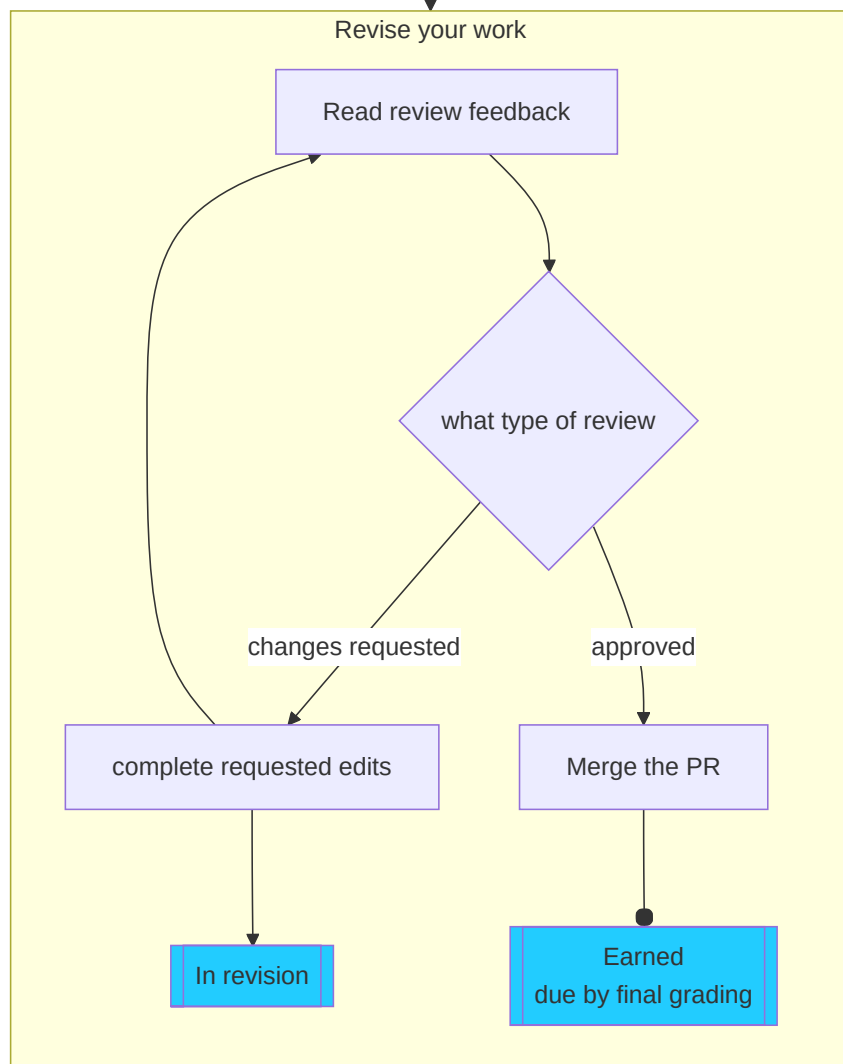
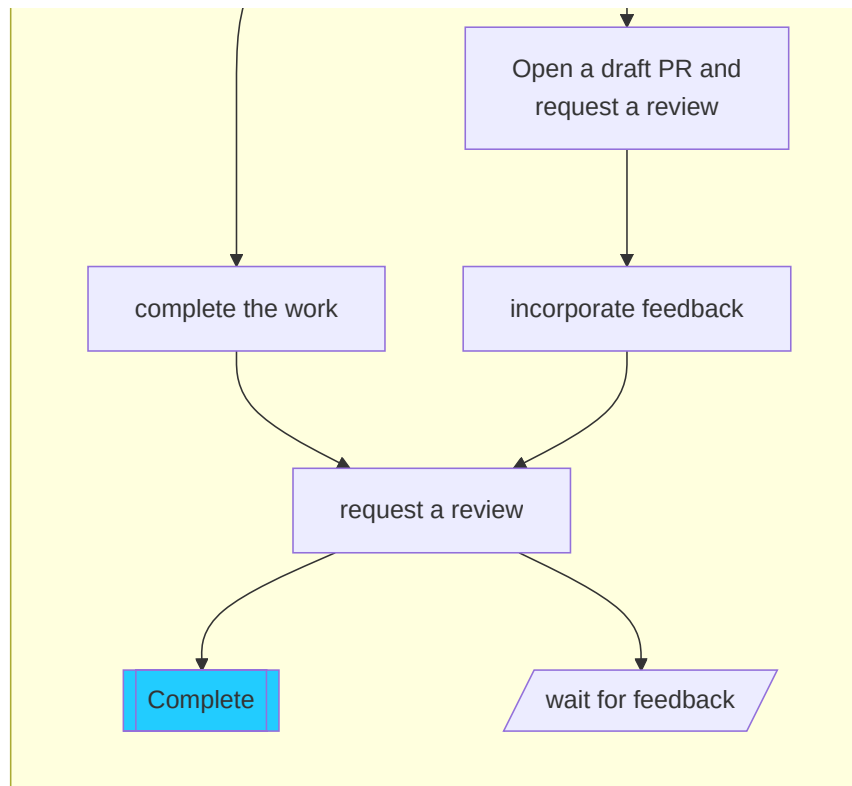
Build Badges

Propose the Topic and  
Product



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:

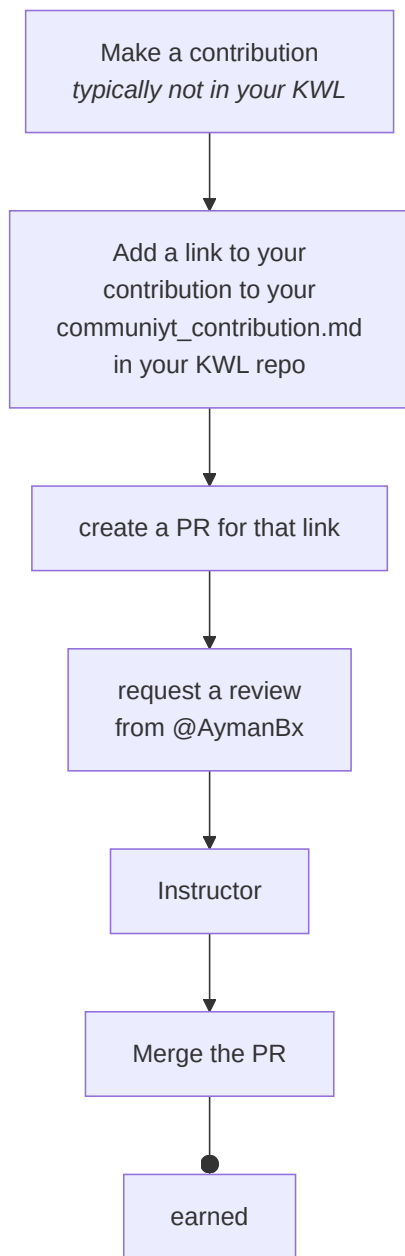
- In the `community_contributions.md` file on your kwl repo, add an item in a bulleted list (start the line with - )
- Include a link to your contribution like `[text to display](url/of/contribution)`
- create an individual [pull request](#) titled “Community-shortname” where `shortname` is a short name 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

! No

You  
help  
org



## 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

	threshold
letter	
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.

### i 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

### i 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.

## 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()
```

 No

the  
pro  
bec  
revi  
like

You  
and  
deta

	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	commuication 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 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

	question	keyword	conceptual	practical	social	activity
date						
2025- -	How can I work on a remote server?	server	server, hpc, large files	ssh, large files, bash head, grep, etc	hidden impacts of remote computation	log into a remote server and work with large f...
2025- -	What is an IDE?	IDE	IDE parts	compare and contrast IDEs	collaboraiton 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 authenitcate more securely from a te...	server use	ssh keys, hpc system strucutre	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 vocabular...	build code in C and examine intermediate outputs
2025- -	What happens when we run code?	hardwar	von neuman architecture	reading a basic assembly language	historical context of computer architecures	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 developents, ...	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 brea...	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 built-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 prisma 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 prisma 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 notificaiton [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 .yaml 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 prisma, 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
- ☒ Assignment or issue that needs to be fixed in our repo/project

What are Pull Requests for?

- ☐ bug reporting and tracking
- ☒ 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
- ☒ 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
- ☒ 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 of your repo that has one file that `is` new (different `from` the main branch where that file exists)  
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

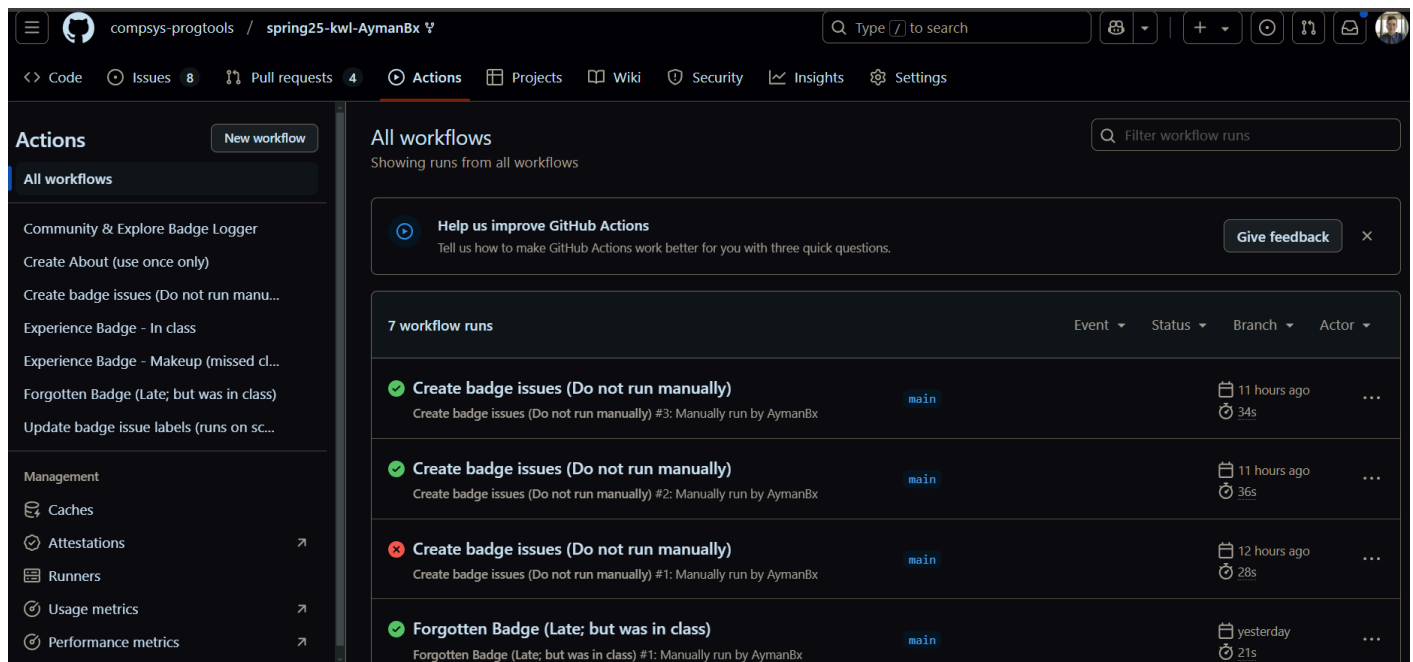
- code review (working independently and then reviewing)
- pair programming (sitting together and discussing while writing)

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.



*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?

- ☒ 4
- ☐ 5
- ☐ 3

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

- ☐ 4
- ☒ 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 **d**irectory with `cd`

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

We can **mak** a new **d**irectory 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 use the `tab` key 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 console outputs:

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

```
cd Doc
```

Press `tab` ... The console 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](#)

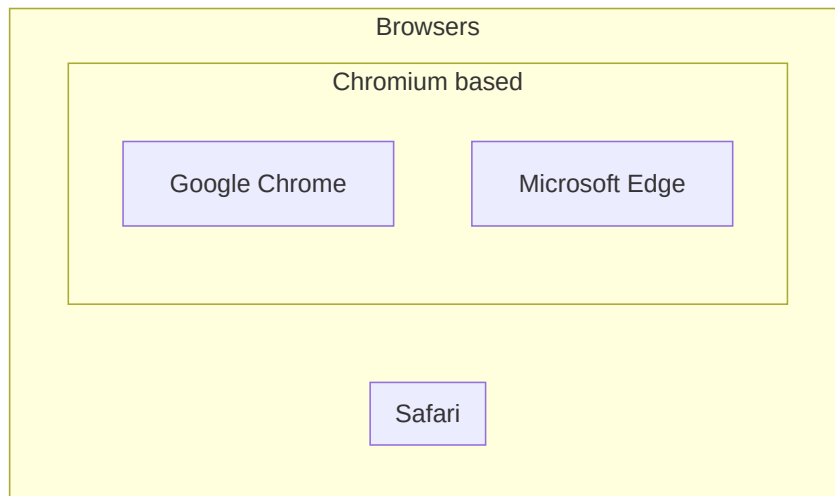
1. Go to the Actions tab
2. Click the Experience Badge (inclass) on the left hand side
3. Click the run workflow button on the right
4. Click the run workflow button in the popup

## 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 recieved 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
- ☒ cd
- ☐ pwd
- ☐ mkdir

**Note**

Remember: cd stands for **change directory**

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

- ☐ absolute path only
- ☐ relative path only
- ☒ 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 your **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 further 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 happened 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`.

#### 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
```

#### Im

If yo  
into

a  
wil  
the  
you  
pre  
mod  
ente



```
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
```

#### 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)

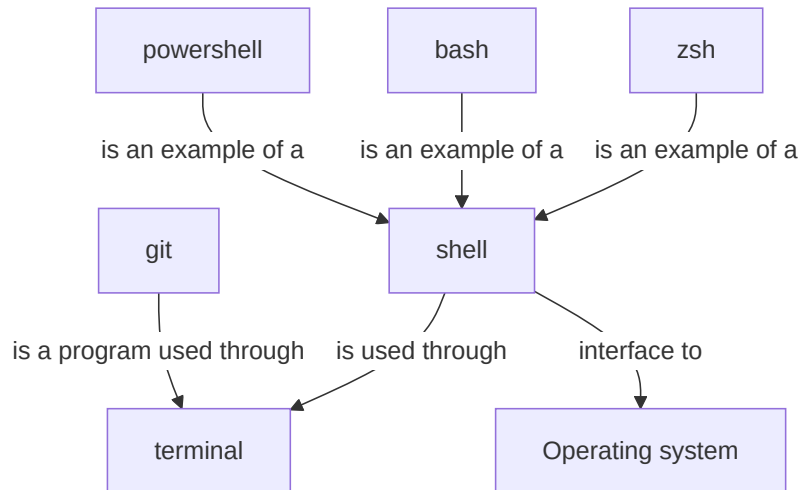
```
git push
```

```
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
```

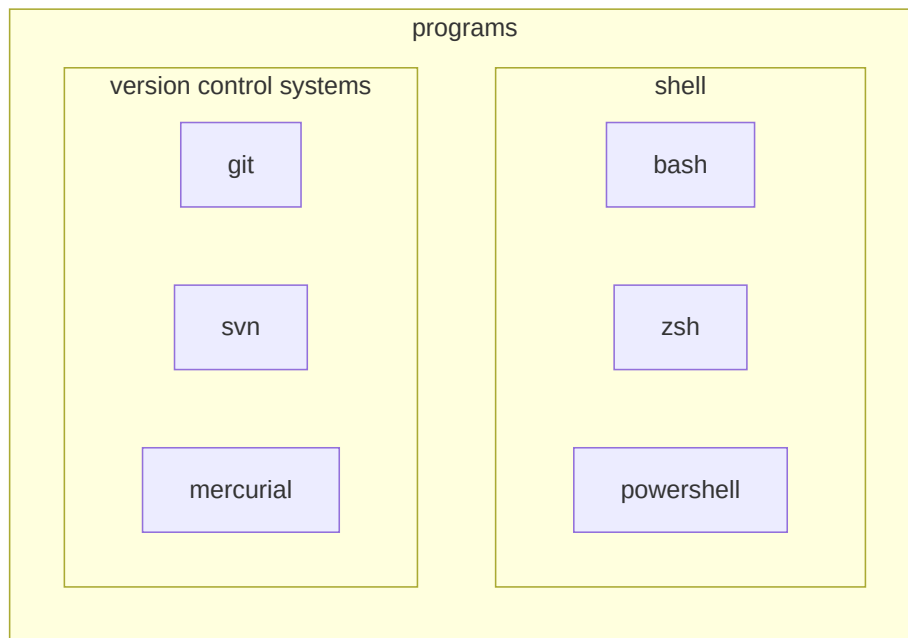
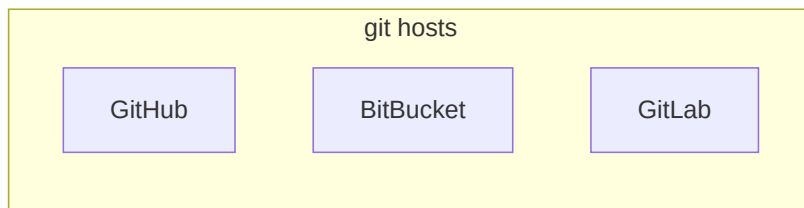
Now check out your online repo. Make sure your on the “1-create-an-about-file” branch

[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
<ul style="list-style-type: none"> <li>• [ ] Review the notes from 02-04</li> <li>• [ ] Ask questions about what we did in the Experience report for 02-04</li> </ul>	

## 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
```

```
.      .git
..     .github
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 linking 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 insertion(+)
 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?

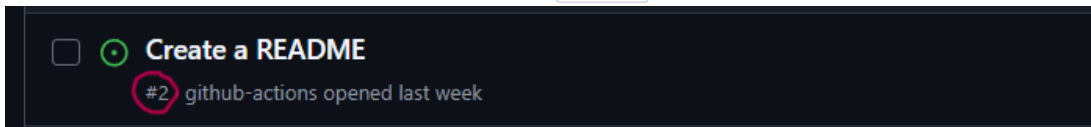
We put the special keyword `closes` in the commit message when we committed the change.

"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”).

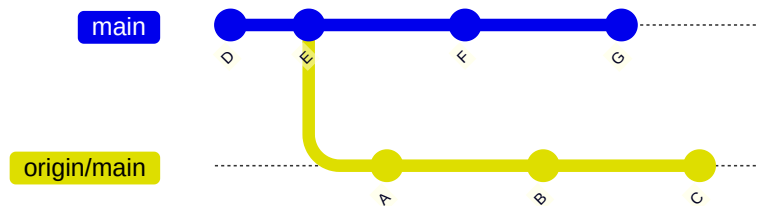
We can see that if this updates the working directory too:

```
ls
```

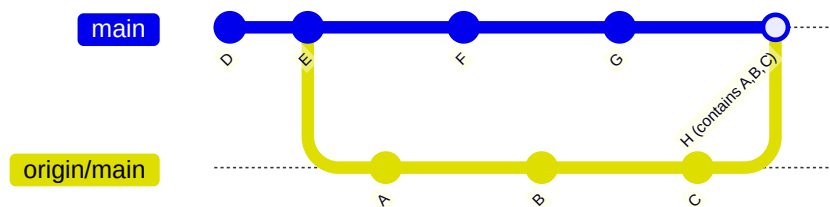
no changes yet. `fetch` updates the `.git` directory so that git knows more, but does not update our local file system.

```
about.md
```

## 5.5.1. Git Merge

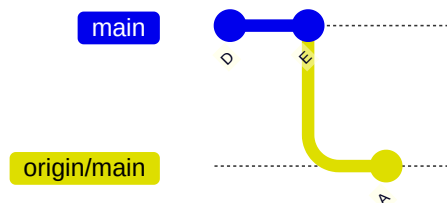


After merge, it looks like this:



There is a new commit with the content.

In our case it is simpler:

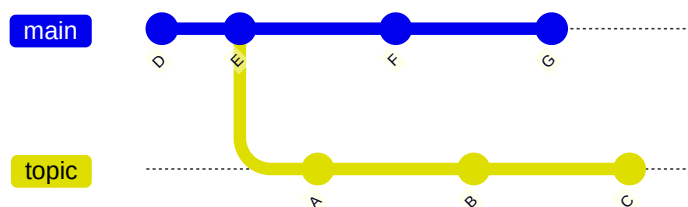


so it will fast forward

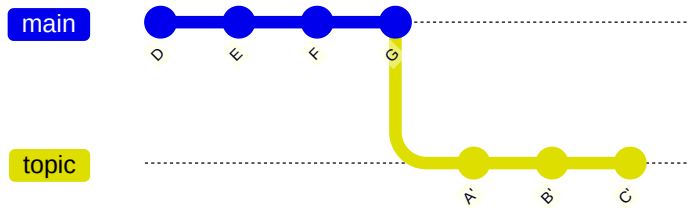


## 5.5.2. git rebase

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 unknown 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 (upstream/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 semester 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 semester 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 semester Masters
```

```
Expected graduation May 2026
- I graduated from highschool abroad
```

Commit changes directly to main

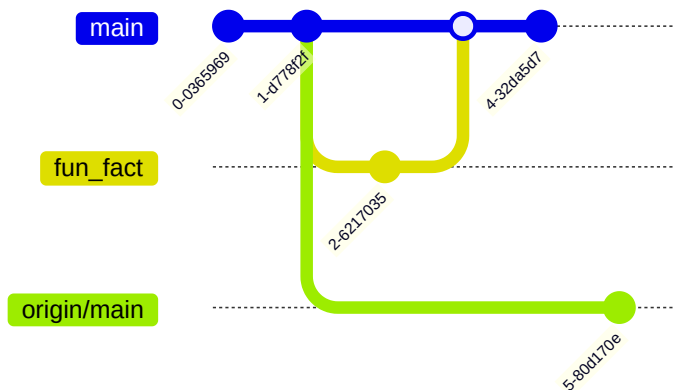


Now let's try updating our local repo with a `pull` command

```
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 semester 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 semester 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.

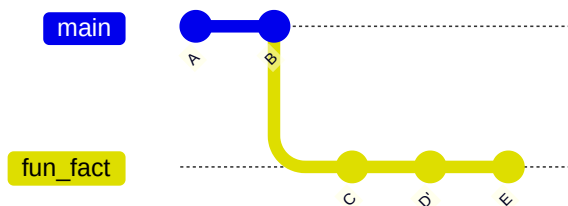
```
git push
```

```
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
```

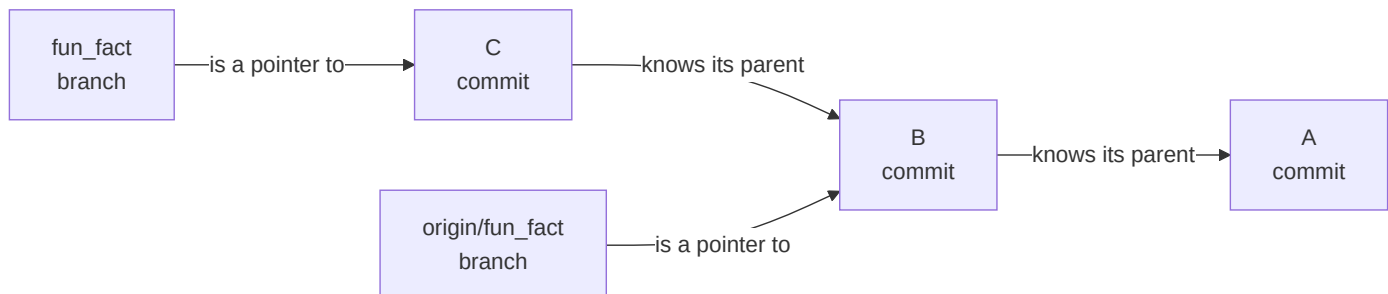
## 5.10. Summary

- branches do not synch 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 "practicse 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
  issuestatus    generate script to appy course issue status
  kwlcsv         generate the activity file csv file for the
  logquestion    award a bonus,
  mkchecklist    transform input file to a gh markdown check
  parsedate      process select non dates
  prfixlist      check json output for titles that will not
  processexport  transform output from mac terminal export t
  progressreport list PR titles from json or - to use std in
  titlecheck     check a single title
```

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

**Understanding this piece comes with practice.**

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

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

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

Ok, my command is still missing something, or has something extra (it says extra args, 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 outp
--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 "practicce 2-11"
```

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

Getting closer. I misspelled `Practice`.

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

```
missing or poorly formatted date
```

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

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.



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>

Terminal is an interface for a shell

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 <11829133+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 <11829133+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 <11829133+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  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
```

---

```
$ 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
```

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 **9942c3c**00dcde51d52a1c110413522043eca07cd  
**8040553**99f6565aca3cb188fe771ef2dca99f959

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	
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.

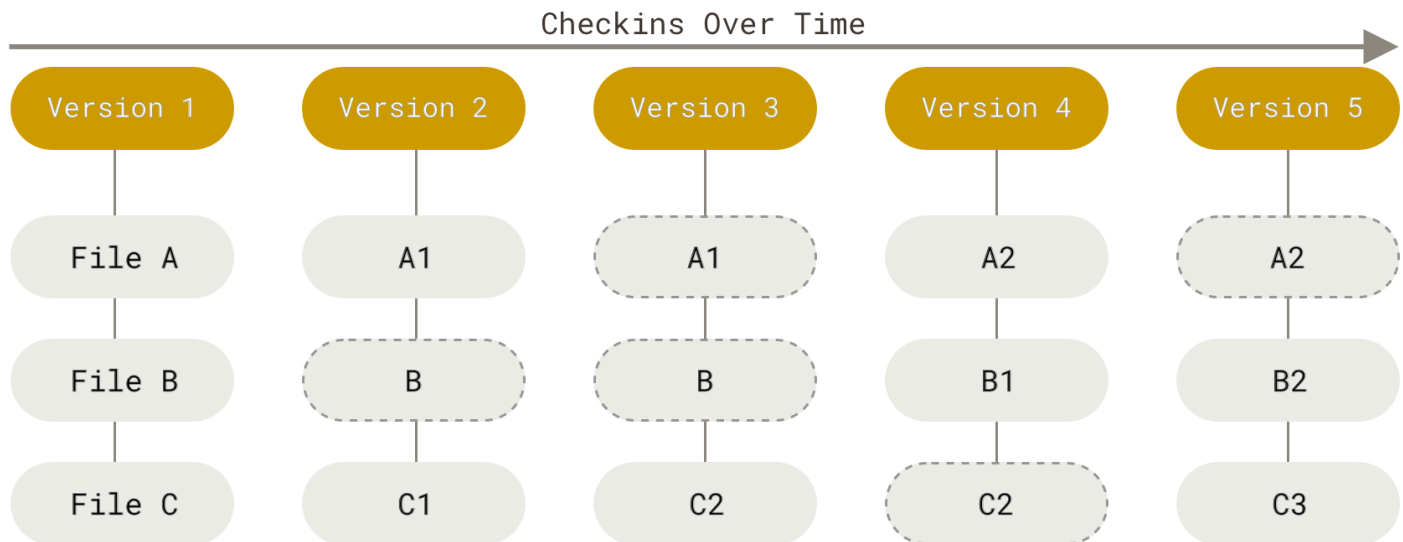
Now we need to contend with multiple definitions

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 (`b0b24aa53537557c582b6f61d409e9b2ee9133f`)

`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 (`b0b24aa53537557c582b6f61d409e9b2ee9133f`)

`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 normla 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 hashes 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
```ecution"]
```

```
++{"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
```"]
ls .git
```

```
console
COMMIT_EDITMSG  REBASE_HEAD  index        packed-refs
FETCH_HEAD     config       info         refs
HEAD           description  logs
ORIG_HEAD
```ks         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/head
```organization
```

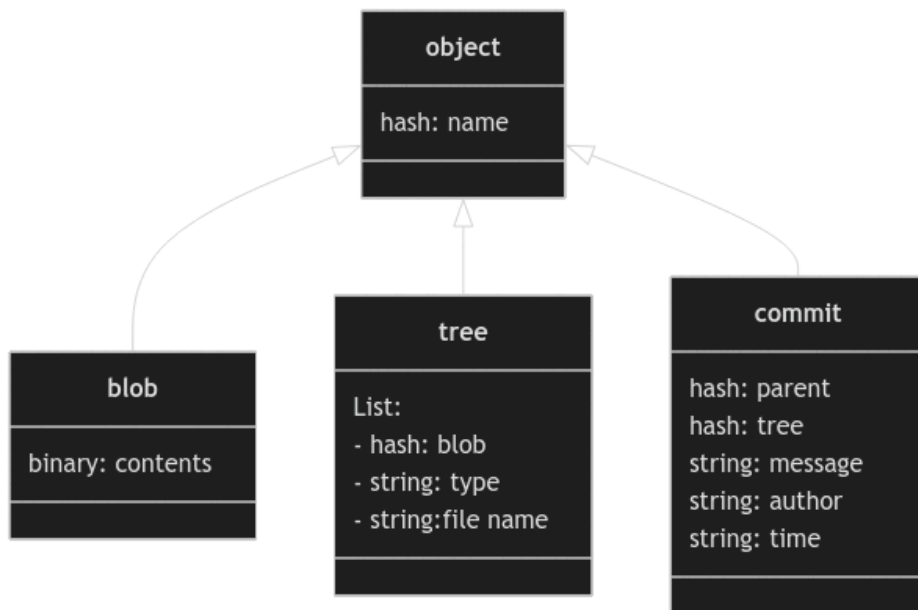
which stores our current branch

```
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□hŸ?%?Cu?↓.?
```Gb?????|Ez8
```

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

```
```onsole
blob
```

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

```
console
<last blob
```ntent 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 branchse 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?

[Chceck it out](#)

It's a way of assuring anyone viewing commits on GitHub tht 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 `gh` 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 to the repo during class on 9/19. \n\n wait until class time (or when you make up class by following the notes) to merge 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 to the repo during class on 9/19. \n\n wait until class time (or when you make up class by following the notes) to merge 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](https://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
about.md        scratch.ipynb

```

## 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
- ☒ 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_imp.py
_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
about.md         scratch.ipynb
```

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 workign 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 | utitly 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
```

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 | 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 |
```

```
$ 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/                 tsets_abc.py
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 **a**ll 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 separat 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 you would 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 supposed 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 | 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
```

### 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...
```

#### 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 | 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 |
```

```
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 | 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 |
```

### ! 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`
- ☒ `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 yml 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** yml 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 93d4ae322683efa703dedbb9acb8431807a38afd
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 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>  
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 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

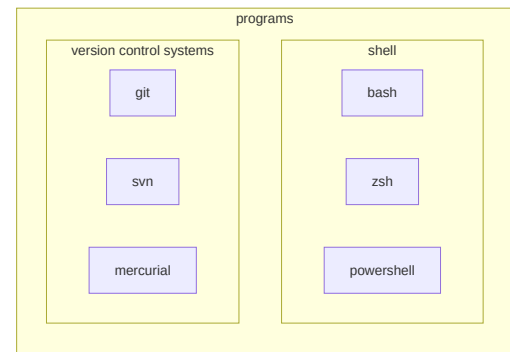
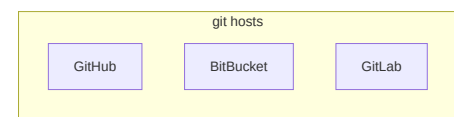
      - name: Install Utils
        run: |
          pip install git+https://github.com/compsys-progtools/

      - name: Get badge requirements
        run: |
          # prepare badge lines
          pretitle="prepare-${(cspt getbadgedate --prepare)}
          cspt getassignment --type prepare | gh issue create -
          # review badge lines
          rtitle="review-${(cspt getbadgedate --review)}
          cspt getassignment --type review | gh issue create -
          # practice badge lines
          pratitle="practice-${(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 practice
      # 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:
```

<code>badgecounts</code>	check if early bonus is met from output of <code>`gh pr list...</code>
<code>combinecounts</code>	combine two yaml files by adding values
<code>createtoyfiles</code>	from a yaml source file create a set of toy files with...
<code>earlybonus</code>	check if early bonus is met from output of <code>`gh pr list...</code>
<code>exportac</code>	export ac files for site from lesson
<code>exporthandout</code>	export prismia version of the content
<code>exportprismia</code>	export prismia version of the content
<code>getassignment</code>	get the assignment text formatted
<code>getbadgedate</code>	cli for calculate badge date
<code>grade</code>	calculate a grade from yaml that had keys of...
<code>issuestat</code>	
<code>issuestatus</code>	generate script to apply course issue status updates
<code>kwlcsv</code>	generate the activity file csv file for the site...
<code>mkchecklist</code>	transform input file to a gh markdown checklist, if the...
<code>parsedate</code>	process select non dates
<code>prfixlist</code>	check json output for titles that will not be counted...
<code>processexport</code>	transform output from mac terminal export to myst...
<code>progressreport</code>	list PR titles from json or - to use std in that have...
<code>titlecheck</code>	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, let's 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 the
...
- 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 tools
...
```

```
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
```

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.

## 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.


## Minimum Rows

## Warning

To be updated

## Required Files

This lists the files for reference, but mostly you can keep track by badge issue checklists.

 Tip  
You  
on y

date	file	type
2025-01-28	brain.md	/_practice
2025-01-30	gitoffline.md	/_review
2025-01-30	gitoffline.md	/_practice
2025-02-11	branches.md	/_review
2025-02-11	branches-forks.md	/_practice
2025-02-13	command-help.md	/_practice
2025-02-18	commit-def.md	/_review
2025-02-20	terminal_review.md	/_review
2025-02-20	software.md	/_prepare
2025-02-20	terminalpractice.md	/_practice
2025-02-20	terminal_organization_adv.md	/_practice
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. 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	

## 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

```
```\nto make a code block use 3 back ticks\n```
```

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 there 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](#).

## Build Badges

Build may be individual or in pairs.

## 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

### 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 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 resource
```

## 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

This project will provide information for a potential contributor to add new features to a code base. The

### ### 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. add a license, readme, and contributor file

1. add [code tours](<https://marketplace.visualstudio.com/items?itemName=vs1s-contrib.codetour>) that help

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 multiple 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)

## **yaml**

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

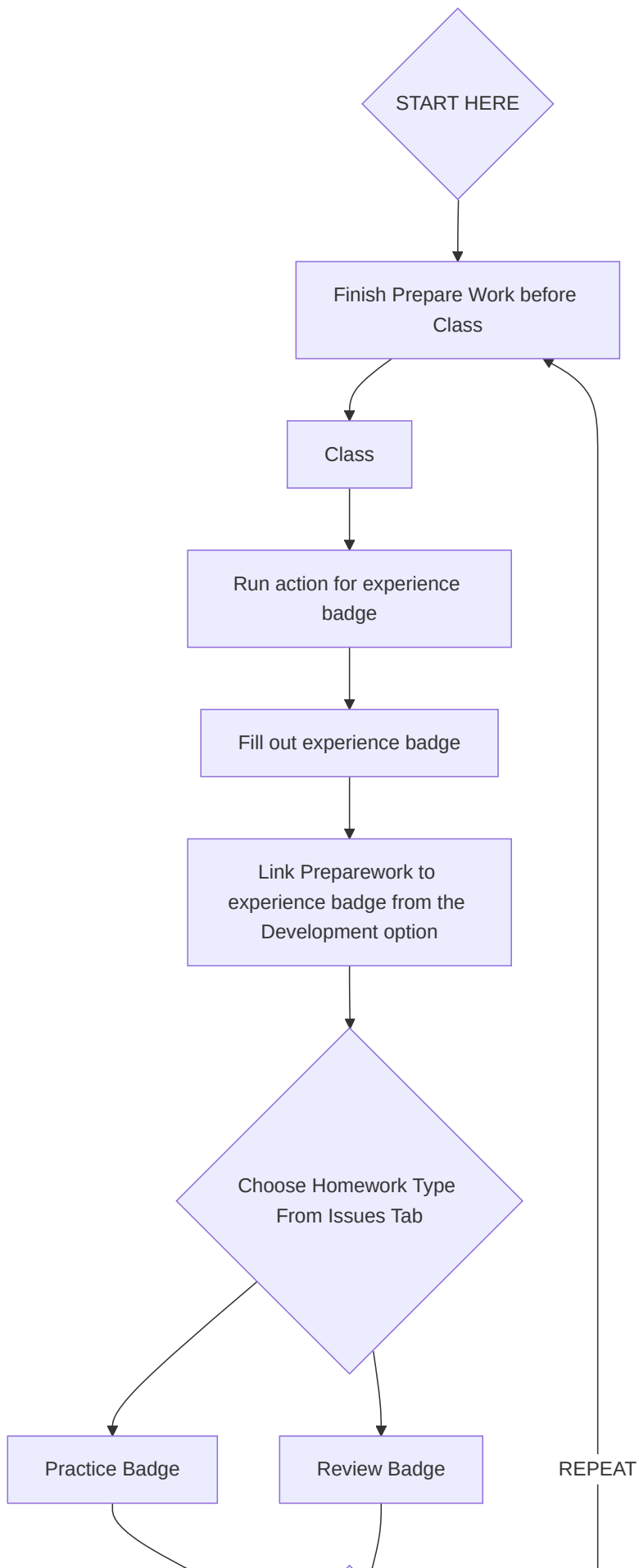
# General Tips and Resources

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.

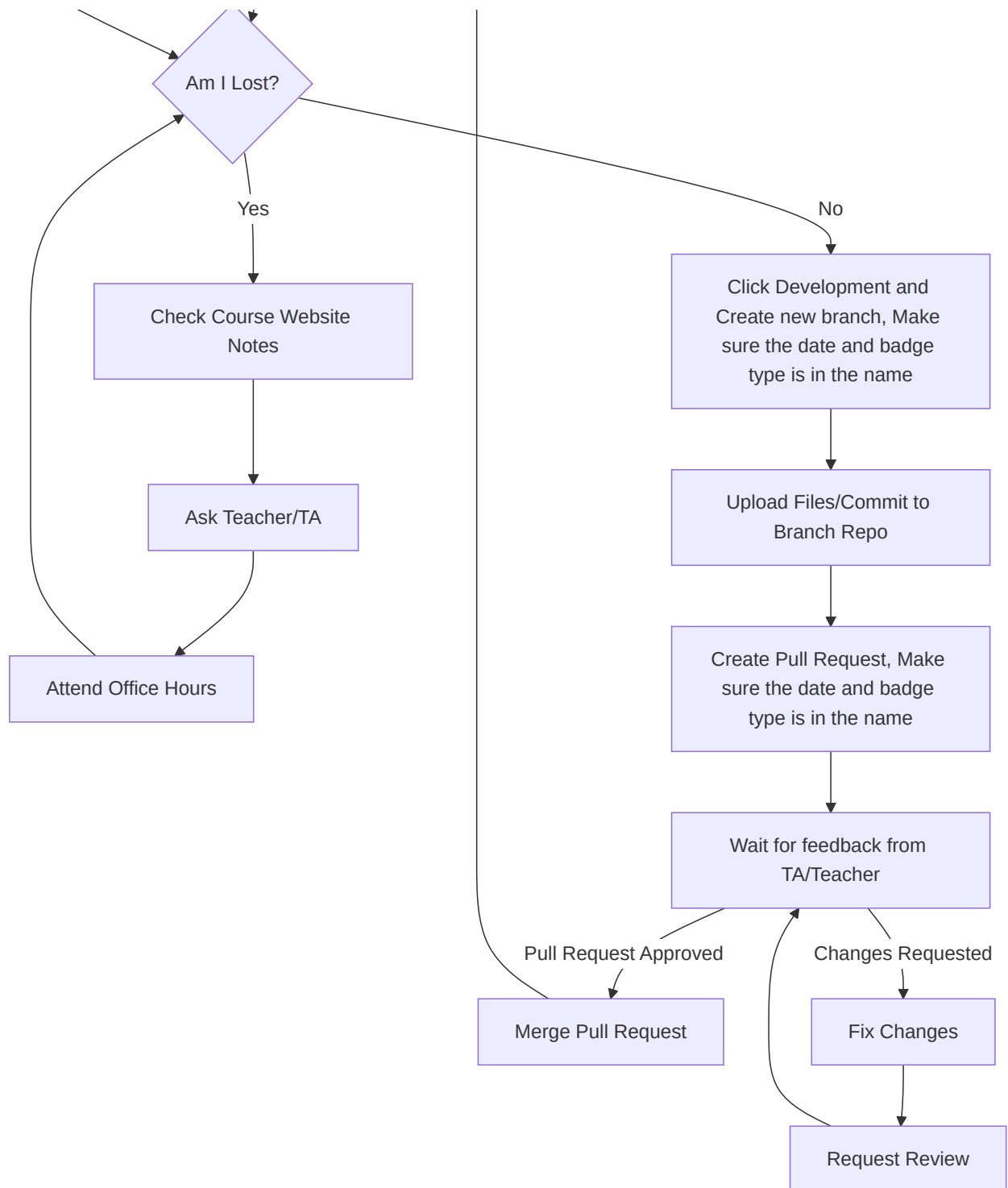
## **on email**

- [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.

Begin 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 see

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 is based on a recommended practice 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.

A new book  
programm  
[Program](#)  
available  
that links

## 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 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](#)

**This is the main view of the project**

Branch menu & info, file action buttons, download options (green code button)

File panel

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.

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.

About has basic facts about the repo, often including a link to a documentation page

Releases, Packages, and Environments are optional sections that the repo owner can toggle on and off.

[Releases](#) mark certain commits as important and give easy access to that version. They are related to [git tags](#)

[Packages](#) are out of scope for this course. GitHub helps you manage

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.

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

distributing your code to make it easier for users.

[Environments](#) are a tool for dependency management. We will cover things that 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

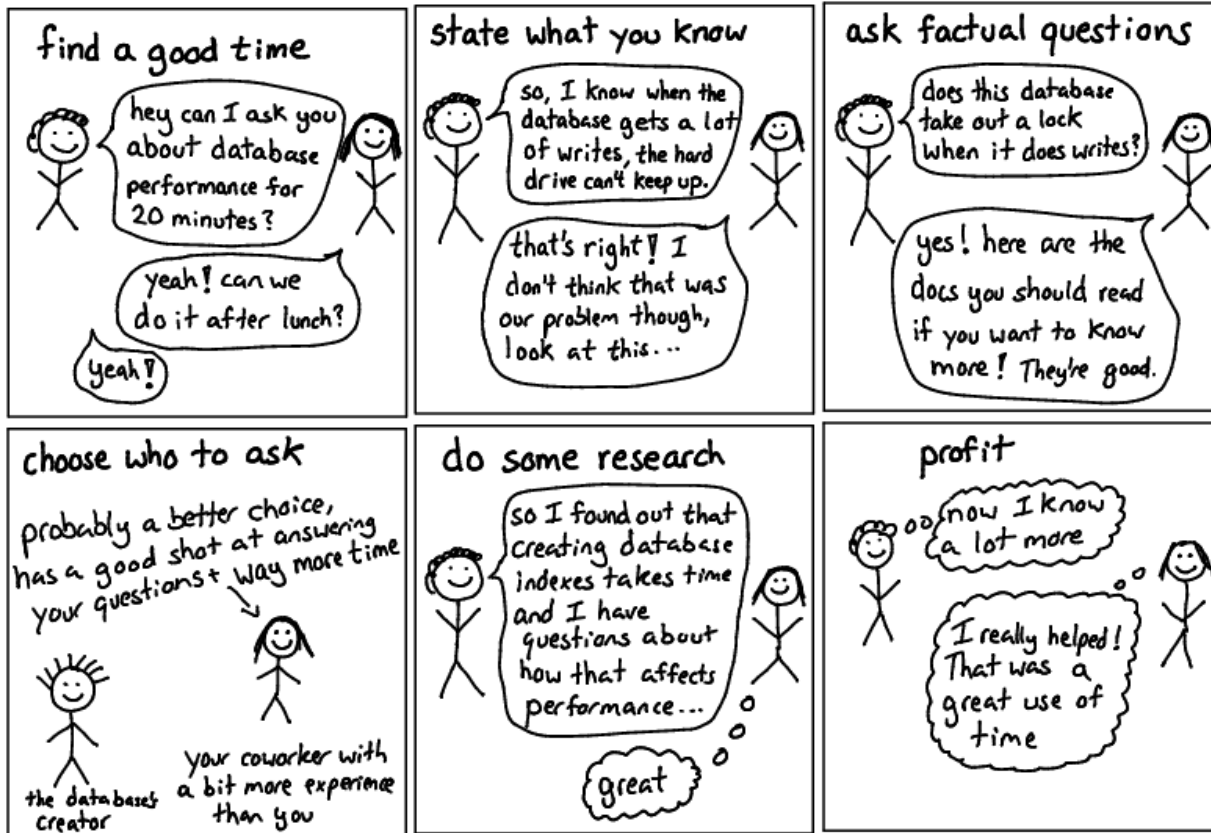
## Getting Help with Programming

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.

## 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.

**N**  
A fu  
[deb](#)

## 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\`