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. Working Offline
- 4. How do git branches work?
- 5. When do I get an advantage from git and bash?
- 6. Patterns in git and bash
- 7. Why are these tools like this?
- 8. What *is* a commit?
- 9. How do programmers communicate about code?
- 10. What *is* git?
- 11. What is a commit number?
- 12. How does git *make* a commit?
- 13. How can I automate things with bash?

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 2024 with Professor Brown.

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

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 Office hours: listed on communication page

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.

You can call me Professor Brown or Dr. Brown, I use she/her pronouns.

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 Dr. Brown 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

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
- 13 lab check outs

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

For an A you can choose:

- 18 review + 3 build
- 18 pracitce + 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
- completeing an “experience report”
- attaching evidence as indiated in notes OR attending office hours to show the evidence

Tip

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

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

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

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

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

Do you earn badges for prepare for class?

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

What do you do when you miss class?

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

What if I have no questions?

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

Lab Checkouts

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

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

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

Review and Practice Badges

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

You can earn review and practice badges by:

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

Where do you find assignments?

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

You should create one PR per badge

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

Important

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

Explore Badges

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

Details and more ideas are on the explore page.

You can earn an explore badge by:

- creating an issue proposing your idea (consider this ~15 min of work or less)
- adjusting your idea until given the proceed label
- completing your exploration
- submitting it as a PR

- making any requested changes
- merging the PR after approval

For these, ideas will almost always be approved, the proposal is to make sure you have the right scope (not too big or too small). Work submitted for explore badges will be assessed for depth beyond practice badges and correctness. Revisions will be more common on the first few as you get used to them, but typically decraese 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 compelte build are uncommon. If you do not submit work for intermediate review, you may need to revise the complete build. The build proposal will assessed for relevance to the course and depth. The work will be assessed for completeness in comparison to the propsal 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

```
---------------------------------------------------------------------------
TypeError                                 Traceback (most recent call last)
Cell In[1], line 31
     28 complete_deadline = date.today() - timedelta(14)
     30 # mark eligible experience badges
---> 31 badge_target_df['experience'][date.fromisoformat(badge_target_df['date']) <= today] = 'eligible'
     32 # mark targets, cascading from most recent to oldest to not have to check < and >
     33 badge_target_df['review_target'][badge_target_df['date'] <= today] = 'active'

TypeError: fromisoformat: argument must be str
```

November 5 above will actually be on November 6

Getting Feedback

Who should you request/assign?

course item type	issue asignee	PR reviewer
prepare work	not required; can be student	none requierd; merge to experience branch
experience badge	N/A	@VioletVex (will be automatic)
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 @brownsarahm (also student optionally)	@brownsarahm
build badge	proposal, assigned to @brownsarahm (also student optionally)	@brownsarahm
anything merged pre-emptively in penalty free	@brownsarahm	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 Dr. Brown.

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 badges must be *started* within one week of when the 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 ineligble 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

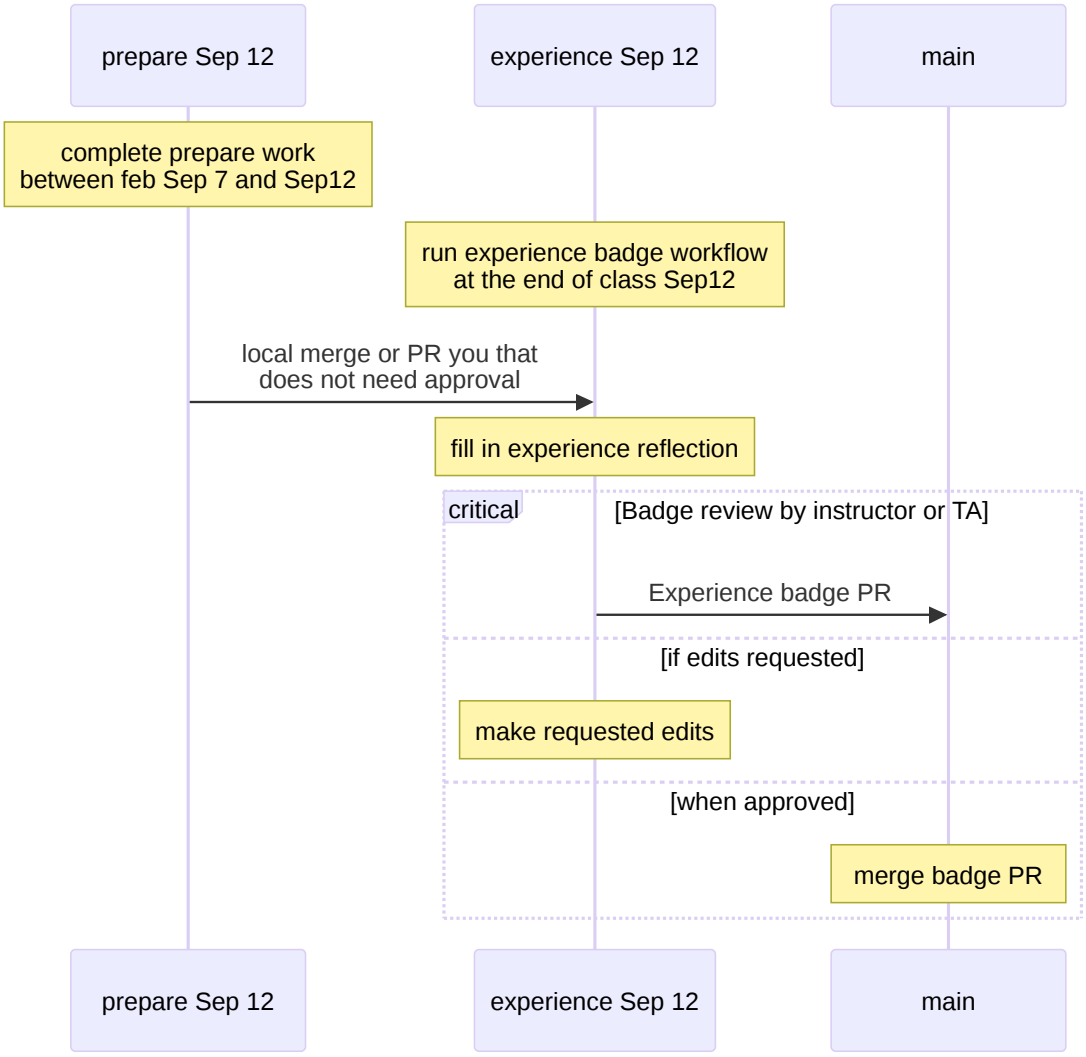
At most one build badge will be evaluated every 4 weeks. This means that if you want to earn 3 build badges, the first one must be in 8 weeks before the end of the semester, March 4. The second would be due April 1st, and the third submitted by the end of classes, April 29th.

Procedures

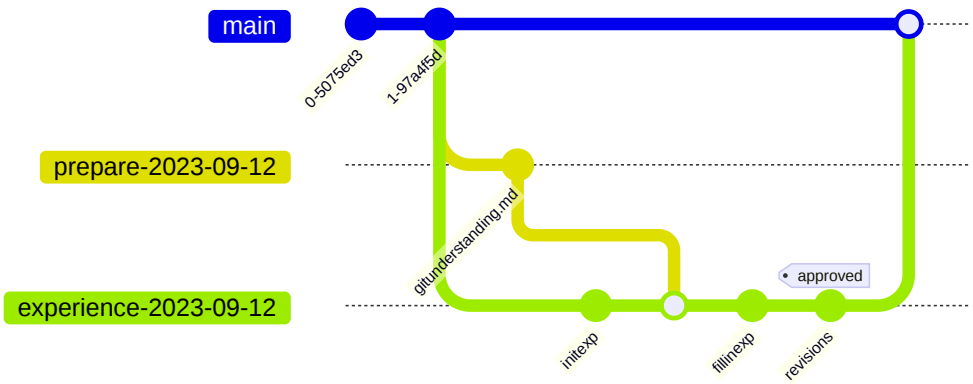
Prepare work and Experience Badges Process

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:



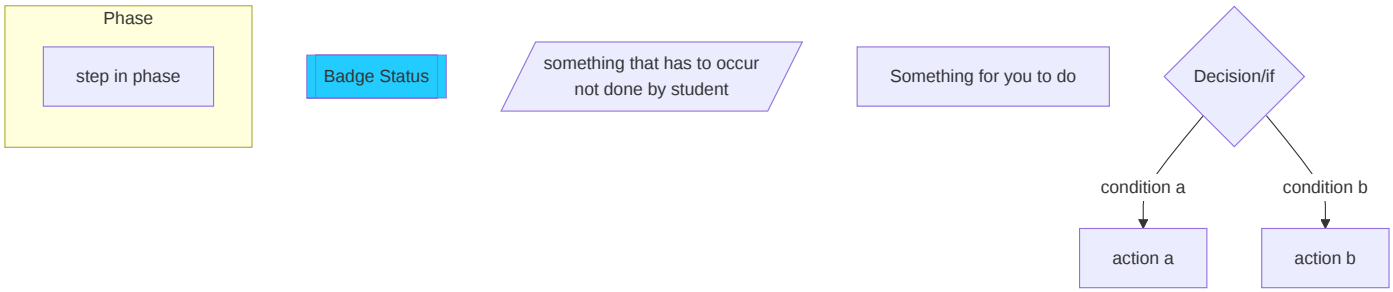
You can merge the prepare into the experience with a PR or on the command line, your choice.

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

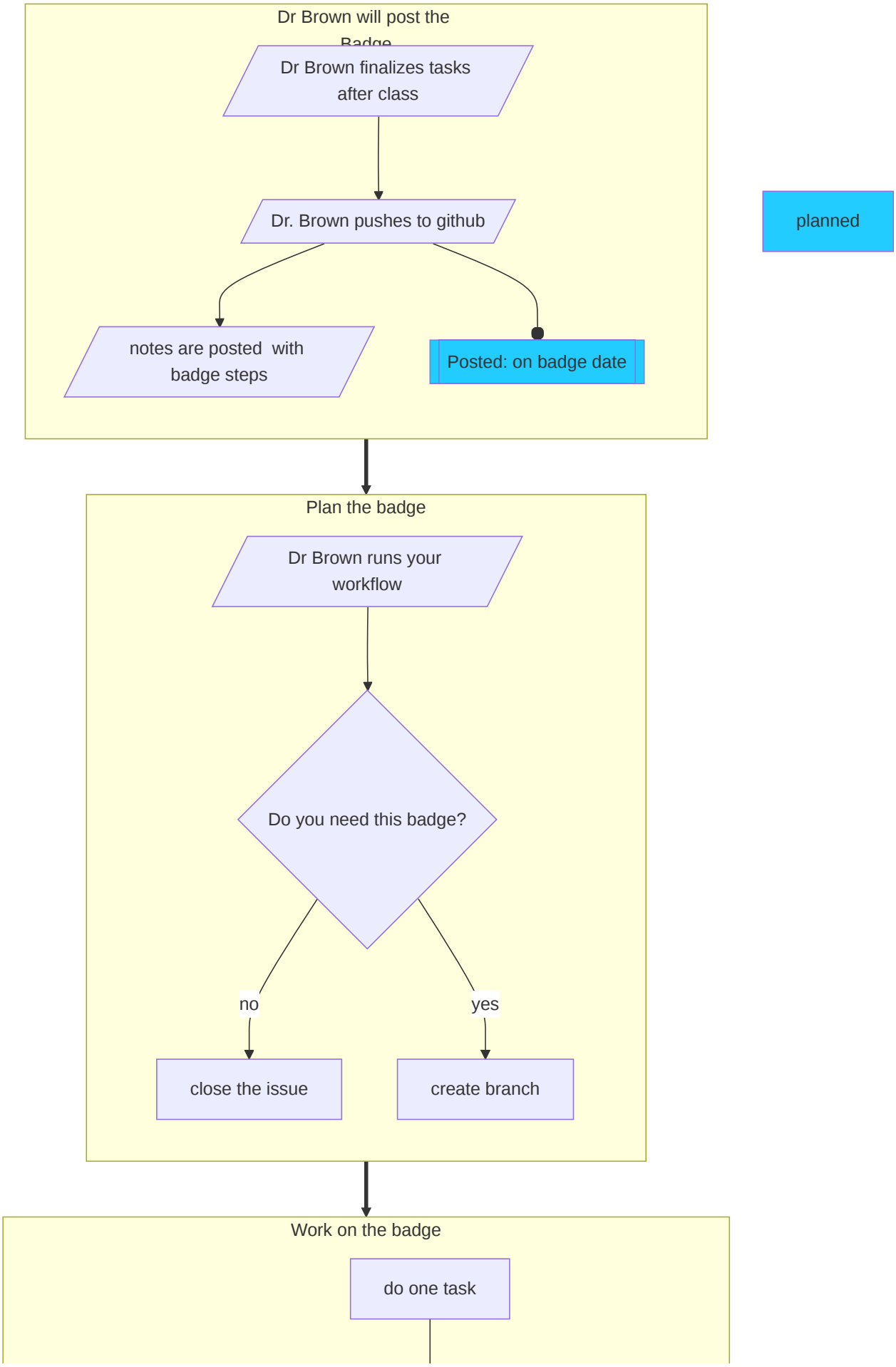
You can, once you know how, do this offline and do the merge with in the CLI instead of with a PR.

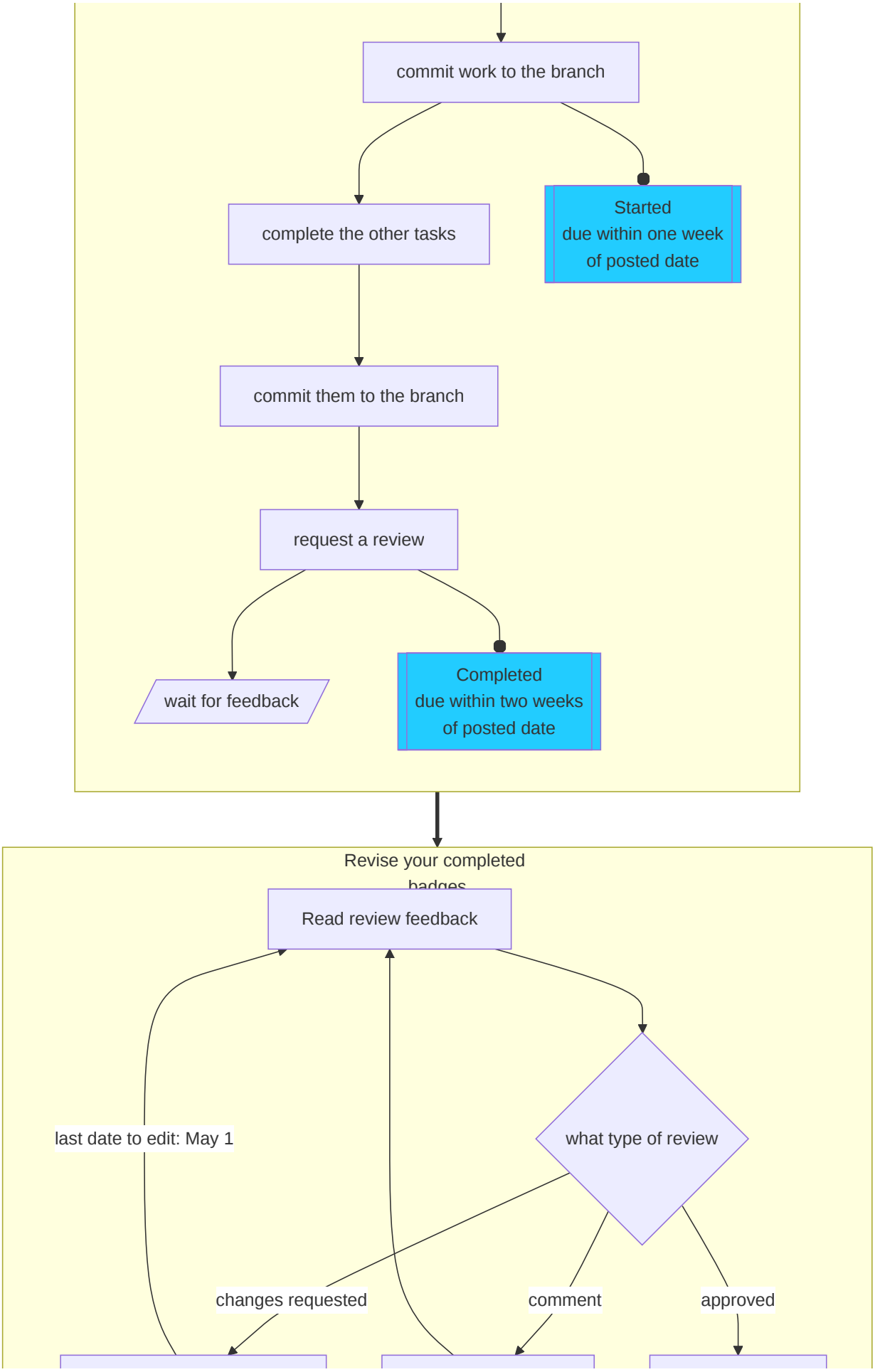
Review and Practice Badge

Legend:



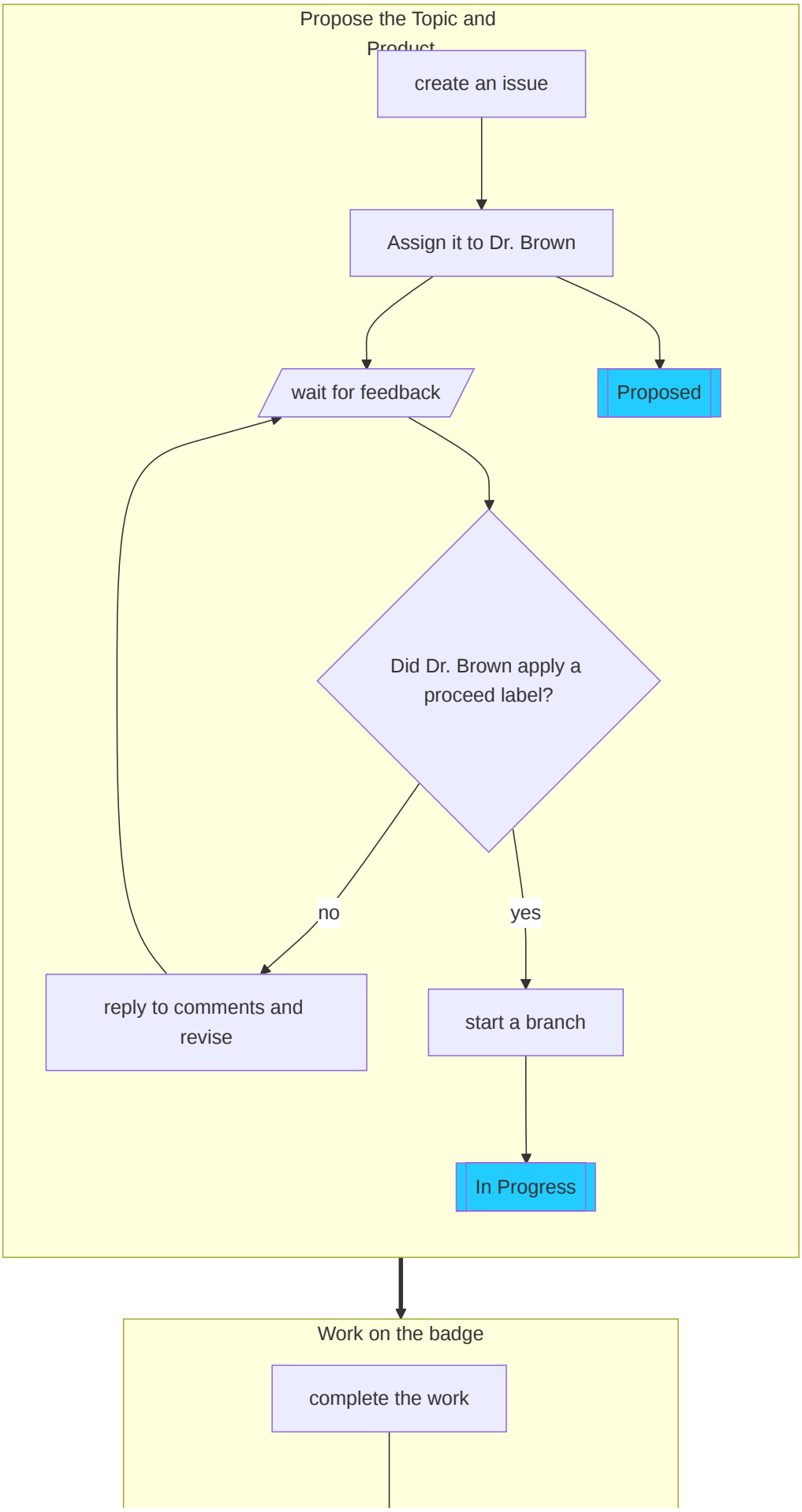
This is the general process for review and practice badges

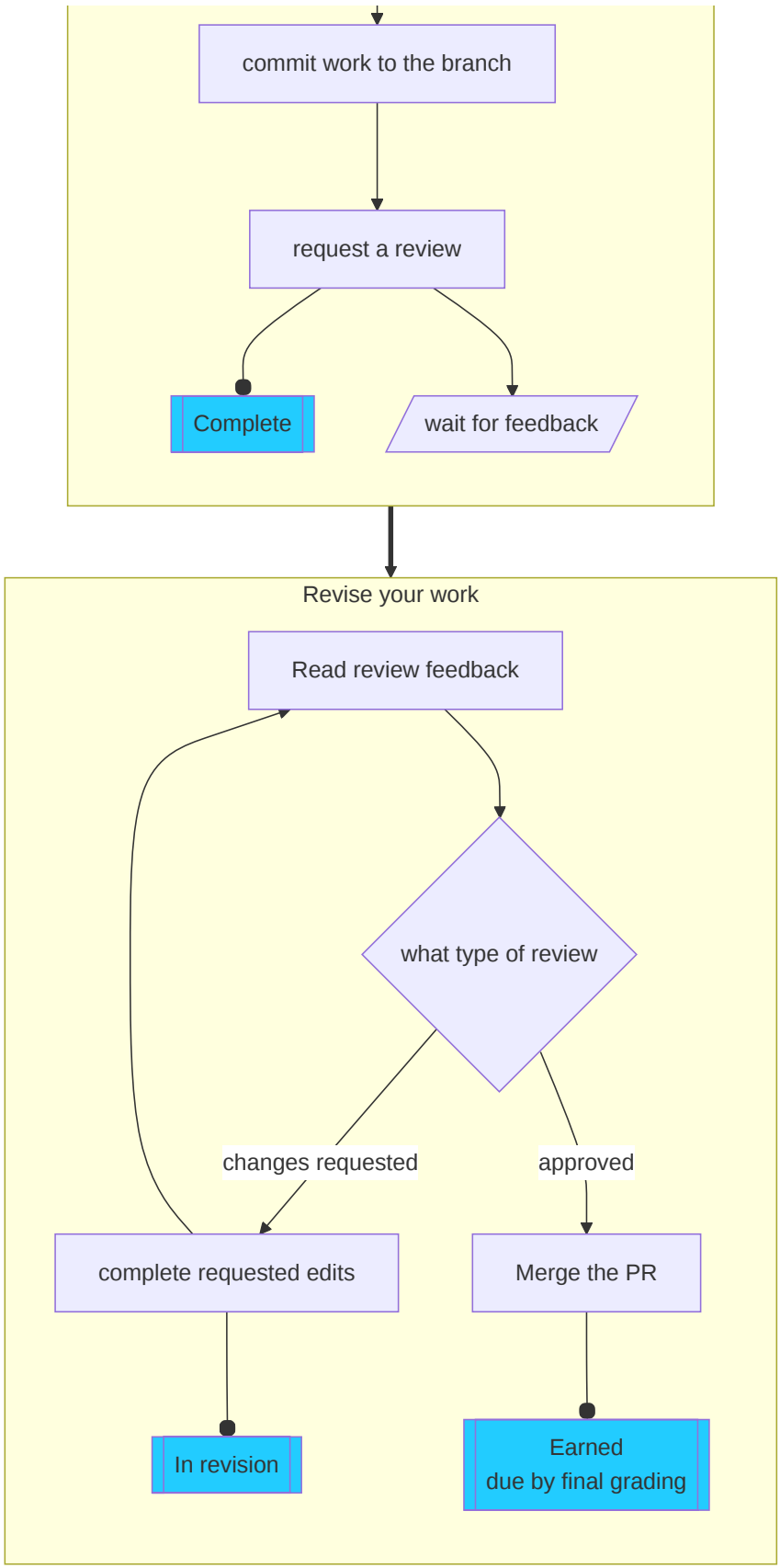

Dr Brown will post the
Badge
Dr Brown finalizes tasks after class
Dr. Brown pushes to github
notes are posted with badge steps
Posted: on badge date
planned
Plan the badge
Dr Brown runs your workflow
Do you need this badge?
no
yes
close the issue
create branch
Work on the badge
do one task


commit work to the branch
complete the other tasks
Started
due within one week
of posted date
commit them to the branch
request a review
wait for feedback
Completed
due within two weeks
of posted date
Revise your completed
badges
Read review feedback
last date to edit: May 1
what type of review
changes requested
comment
approved

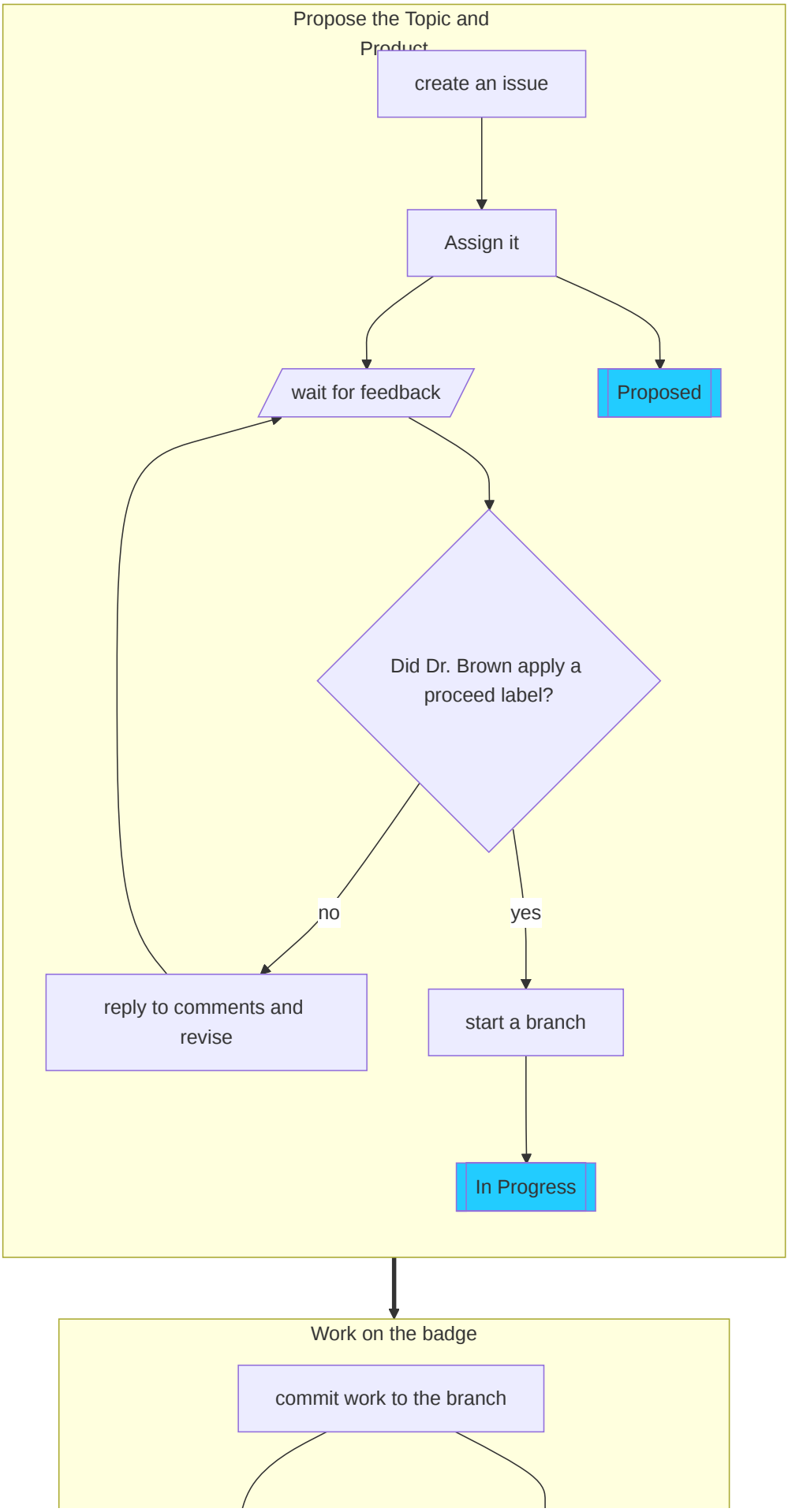

complete requested edits
reply to comments
Merge the PR
Earned
due by final grading

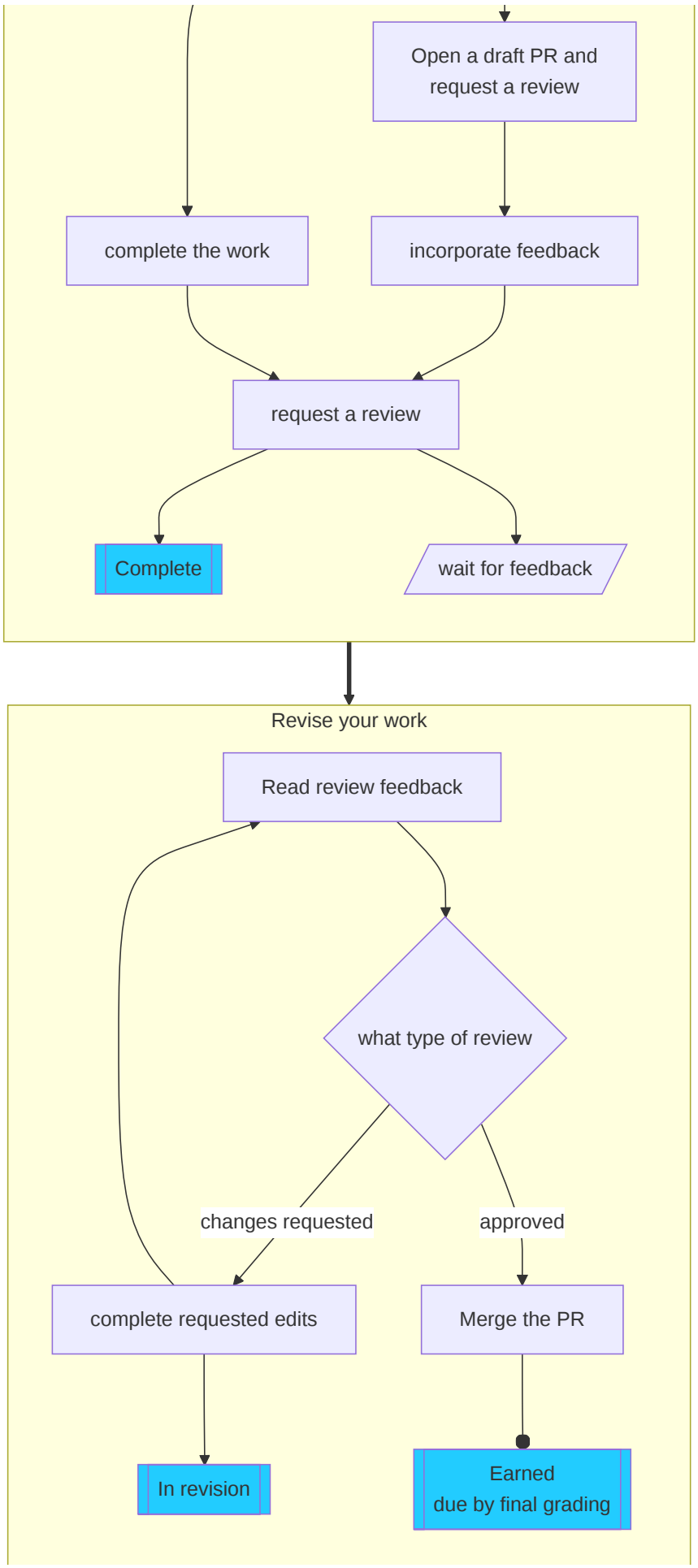
Explore Badges


Propose the Topic and Product
create an issue
Assign it to Dr. Brown
wait for feedback
Proposed
Did Dr. Brown apply a proceed label?
no
yes
reply to comments and revise
start a branch
In Progress
Work on the badge
complete the work


commit work to the branch
request a review
Complete
wait for feedback
Revise your work
Read review feedback
what type of review
changes requested
approved
complete requested edits
Merge the PR
In revision
Earned
due by final grading

Build Badges


Propose the Topic and Product
create an issue
Assign it
wait for feedback
Proposed
Did Dr. Brown apply a proceed label?
no
yes
reply to comments and revise
start a branch
In Progress
Work on the badge
commit work to the branch


Open a draft PR and request a review
complete the work
incorporate feedback
request a review
Complete
wait for feedback
Revise your work
Read review feedback
what type of review
changes requested
approved
complete requested edits
Merge the PR
In revision
Earned due by final grading

Community Badges

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

Logger Action

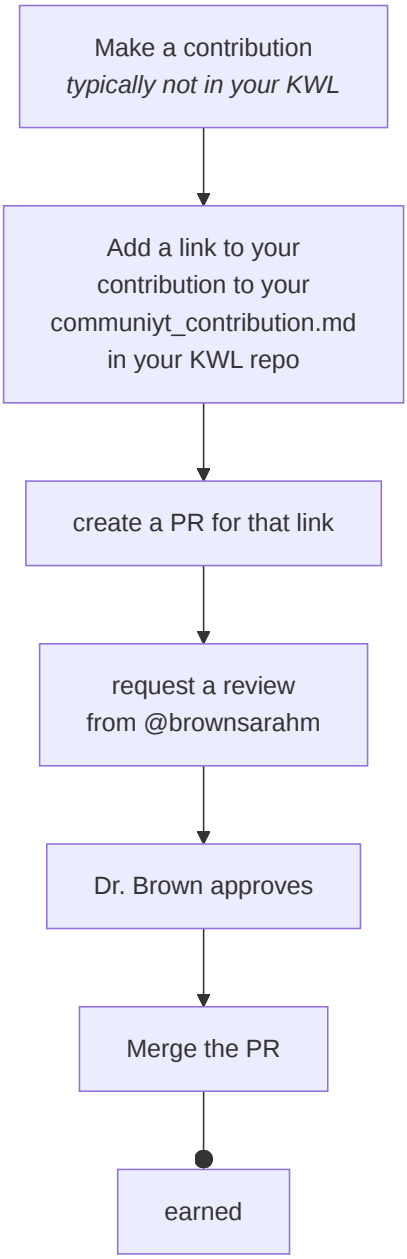
Your KWL repo has an action called “Community & Explore Badge Logger” that will help you

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 Dr. Brown will constitute credit for your grade
- request a review on that PR from @brownsarahm

> **Important**
>
> You want one contribution per PR` for tracking



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 litle bit and this page will be updated.

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

▶ Show code cell source

Grade cutoffs for total influence are:

▶ Show code cell source

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

The total influence of each badge is as follows:

▶ Show code cell source

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

Bonuses

In addition to the weights for each badge, there also bonuses that will automatically applied to your grade at the end of the semester. These are for longer term patterns, not specific assignments. You earn these while workng 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	13 lab badges	18	auto
Breadth	If review + practice badges >=18:	32	auto
Git-ing unstuck	fix large mistakes your repo using advanced git operations and submit a short reflection (allowable twice; Dr. Brown must approve)	9	event
Early bird	(review + practice) submitted by 9/26 >=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; Dr. Brown will log & award)	9	event
Community Star	10 community badges	18	auto
Hack the course - Contributor - Build	1 build that contributes to the course infrastructure/website +1 community or review	18	event
Hack the course - Contributor - Explore	1 explore that contributes to the course infrastructure/website + 2 community, with at least 1 review	18	event
Hack the course - Critic	5 total community badge, at least 2 reviews of other course contributions	9	event

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

Note

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

Note

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

Bonus Implications

Attendance and participation is *very* important:

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

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

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

Example 2:

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

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

Early Bird, Descriptive commits, Community Star, and Git-ing Unstuck are all equal to the half differnce 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)
```

```
291
```

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
- compiliation
- 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 accredidation 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 fall 2024.
>
> 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()
```

date	question	keyword	conceptual	practical	social	activity
2023-09-07	Welcome, Introduction, and Setup	intro	what is a system, why study tools	GitHub basics	class intros	create kwl repo in github, navigate github.com...
2023-09-12	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
2023-09-14	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
2023-09-19	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
2023-09-21	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...
2023-09-26	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...
2023-09-28	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
2023-10-03	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...
2023-10-05	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
2023-10-12	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
2023-10-17	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
2023-10-19	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
2023-10-24	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

date	question	keyword	conceptual	practical	social	activity
2023-10-26	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...
2023-10-31	What is an IDE?	IDE	IDE parts	compare and contrast IDEs	collaboraiton features, developer communities	discussions and sharing IDE tips
2023-11-02	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
2023-11-07	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
2023-11-09	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
2023-11-14	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...
2023-11-16	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...
2023-11-21	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
2023-11-28	What *is* a computer?	architecture	physical gates, history	interpretting specs	social context influences technology	discussion
2023-11-30	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
2023-12-05	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
2023-12-07	How does this all work together	review	all	end of semester logistics	group work final	review quiz, integration/reflection questions
2023-12-12	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()
```

date	topic	activity
2023-09-08	GitHub Basics	syllabus quiz, setup
2023-09-15	working at the terminal	organization, setup kwl locally, manage issues
2023-09-22	offline branches	plan for success, clean a messy repo
2023-09-29	tool familiarity	work on badges, self progress report
2023-10-06	unix philosophy	design a command line tool that would enable a...
2023-10-13	git plumbing	git plumbing experiment
2023-10-20	git plumbing	grade calculation script, self reflection
2023-10-27	scripting	releases and packaging
2023-11-03	remote, hpc	server work, batch scripts
2023-11-10	Compiling	C compiling experiments
2023-11-17	Machine representation	bits and floats and number libraries
2023-12-01	hardware	self-reflection, work, project consultations
2023-12-08	os	hardware simulation

Support Systems

> **Warning**
>
> these links may be outdated, will update soon

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, MySSP (Student Support Program) App, the Wellness Resource Center, and Well-being Coaching.

Academic Enhancement Center

Academic Enhancement Center (for undergraduate courses): Located in Roosevelt Hall, the AEC offers free face-to-face and web-based services to undergraduate students seeking academic support. Peer tutoring is available for STEM-related courses by appointment online and in-person. The Writing Center offers peer tutoring focused on supporting undergraduate writers at

any stage of a writing assignment. The UCS160 course and academic skills consultations offer students strategies and activities aimed at improving their studying and test-taking skills. Complete details about each of these programs, up-to-date schedules, contact information and self-service study resources are all available on the AEC website.

- **STEM Tutoring** helps students navigate 100 and 200 level math, chemistry, physics, biology, and other select STEM courses. The STEM Tutoring program offers free online and limited in-person peer-tutoring this fall. Undergraduates in introductory STEM courses have a variety of small group times to choose from and can select occasional or weekly appointments. Appointments and locations will be visible in the TutorTrac system on September 14th, FIXME. The TutorTrac application is available through URI Microsoft 365 single sign-on and by visiting aec.uri.edu. More detailed information and instructions can be found on the AEC tutoring page.
- **Academic Skills Development** resources helps students plan work, manage time, and study more effectively. In Fall FIXME, all Academic Skills and Strategies programming are offered both online and in-person. UCS160: Success in Higher Education is a one-credit course on developing a more effective approach to studying. Academic Consultations are 30-minute, 1 to 1 appointments that students can schedule on Starfish with Dr. David Hayes to address individual academic issues. Study Your Way to Success is a self-guided web portal connecting students to tips and strategies on studying and time management related topics. For more information on these programs, visit the Academic Skills Page or contact Dr. Hayes directly at davidhayes@uri.edu.
- The **Undergraduate Writing Center** provides free writing support to students in any class, at any stage of the writing process: from understanding an assignment and brainstorming ideas, to developing, organizing, and revising a draft. Fall 2020 services are offered through two online options: 1) real-time synchronous appointments with a peer consultant (25- and 50-minute slots, available Sunday - Friday), and 2) written asynchronous consultations with a 24-hour turn-around response time (available Monday - Friday). Synchronous appointments are video-based, with audio, chat, document-sharing, and live captioning capabilities, to meet a range of accessibility needs. View the synchronous and asynchronous schedules and book online, visit uri.mywconline.com.

General Policies

> **Warning**
>
> links on this page may be outdated, will update soon

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 Response Team at www.uri.edu/brt. 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 accomodations that the DAI office provides will not apply to this course, because of how it is designed: there are no exams for you to get extra time on, and no slides for you to get in advance. However, I am happy to work with you to help you understand how to use the build in support systems for the course.

URI wide:

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

Academic Honesty

Students are expected to be honest in all academic work. A student's name or email address associated with a commit 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, with outside content 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 incuding, IDE-integrated GitHub co-pilot (or similar, IDE embedded tool) for any code that is required for this course because the code is necessary to demonstrate examples, but language syntax is not the core learning outcome.

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

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

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

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

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

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

Attendance

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

Viral Illness Precautions

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. Those who test positive for COVID-19 should follow the isolation guidelines from the Rhode Island Department of Health and CDC.

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, 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 Dr. Brown.

For *extended excused absences*, (3 or more classes) email Dr. Brown 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/fall2024/syllabus/c
```

put this on a branch called `watch_community_badge` and title your PR “Community-Watch”

Help Hours

Day	Time	Location	Host
Wednesday	10am-6pm (appt)	Zoom	Marcin
Monday	5-7pm	Swan 305	Skye
Friday	4-6pm	Zoom	Dr. Brown

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.

Warning

Monday 10/14 5-7pm Help Hours will instead be on Tuesday 10/15 5-6pm because of the holiday schedule shift

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 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 tpo 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.

For E-mail

- use e-mail only for things that **need to be private to Dr. Brown and not seen by TAs**
- other messages may be addressed on your repo or the course website, without a response via email
- Include `[CSC311]` in the subject line of your email along with the topic of your message. This is important, because your messages are important, but I also get a lot of e-mail. Consider these a cheat code to my inbox: I have setup a filter that will flag your e-mail if you include that in subject to ensure that I see it.

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

- Dr. Sarah Brown
- Please address me as Dr. Brown or Professor Brown,
- Ms./Mrs. are not acceptable

You can see more about me in the about section of the syllabus.

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 trickest times.

I was given this excerpt:

```
echo "# fall2024" >> README.md
git init
git add README.md
git commit -m "first commit"
git branch -M main
git remote add origin https://github.com/compsys-progtools/fall2024.git
git push -u origin main
```

but since I had content already I needed to skip several of these steps I needed to know what each one did to skip the right ones.

Assume you have dates stored as a date type, is it the same to add 365 days and add 1 year?

In Python, let's see

```
from datetime import date, timedelta
date.today() + timedelta(days=365)
```

What if we do last year?

```
date(2023,9,3) + timedelta(days=365)
```

I look forward to getting to know you all better.

1.3. 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

or open ended

And I can share responses, grouped up

1.4. 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 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

If you do email me about missing a single clss, I will likely not reply. Not because I do not care about your long term success; I do! I just get too many emails and cannot do the more important parts of my job if I answser every single email. Skipping these emails gives me more time to help students who actually need my help.

1.4.1. 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

1.4.2. 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

1.5. 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

1.6. How does this work?

1.6.1. In class:

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

1.6.2. 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

1.7. 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.

see the link on prismia if you missed class

There is a Glossary!!

repository

pro tip: links are often hints or more information

1.8. 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
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 `day1_kwl`) (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

1.9. 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.

1.9.1. GitHub Actions

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

Action files are stored in the `.github/workflows` folder in yaml files (key value pairs that hold settings and script steps).

Some run automatically, like on any PR or when a commit is to main. The ones in your KWL repo are "manual" so we run them from the Actions tab.

1.9.2. Fix your repo

> **Important**
>
> 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.
>
> You can do this before or in Lab on Monday.

On each file in the `.github/workflows` folder that ends in `.yml` edit in some small way (eg add an additional blank line in a place where there is a blank line already) and commit directly to main.

1.9.3. Get Credit for class

You can use the same content you sent on prismia at the end of class, but for the script to count it at the end of the semester, it has to be in your repo.

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 `2024-09-05`
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 2024-09-05
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

1.10. 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 llike to first learn it? (nothing written to submit, but you can use the issue to take notes if you would like)

1.11. 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)

1.12. We havea Glossary!!

For example, the term we used above:

repository

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

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

Key terms for the first class:

- repository
- git
- github
- PR

1.13. Questions after class

1.13.1. How do I actually use Git/Github with one of my coding projects?

We will build that up

1.13.2. What conceptually are we supposed to take from today lesson aside from just how pull requests work?

the key vocabulary.

1.13.3. What kind of projects will we be completing?

Optionally, you may complete projects. You can see examples in the build section.

1.13.4. Will this class go over topics that will help me on CSC 212 content?

the collaboration on github will help

1.13.5. what a day to day looks like in this class

a lot like this class, but the 3rd class will be more typical than the first 2.

1.13.6. will we need to access git without using github

we will use the git program without GitHub yes, locally on your computer

1.13.7. how to create branches and merge branches to the main. Other basic features of github

We will learn over time

1.13.8. I want to learn all of github's functions.

We will not get to *every* function, but we will cover the main categories

1.13.9. Something that I want to learn more about is using git from the command line

Soon!

1.13.10. Nothing specific, I would just like to learn more about GitHub in general because I know how important of a tool it is

perfect! that is what we will do

1.13.11. just how to utilize github more and all the futures we haven't covered

we will get there

2. More orientation

Today we will:

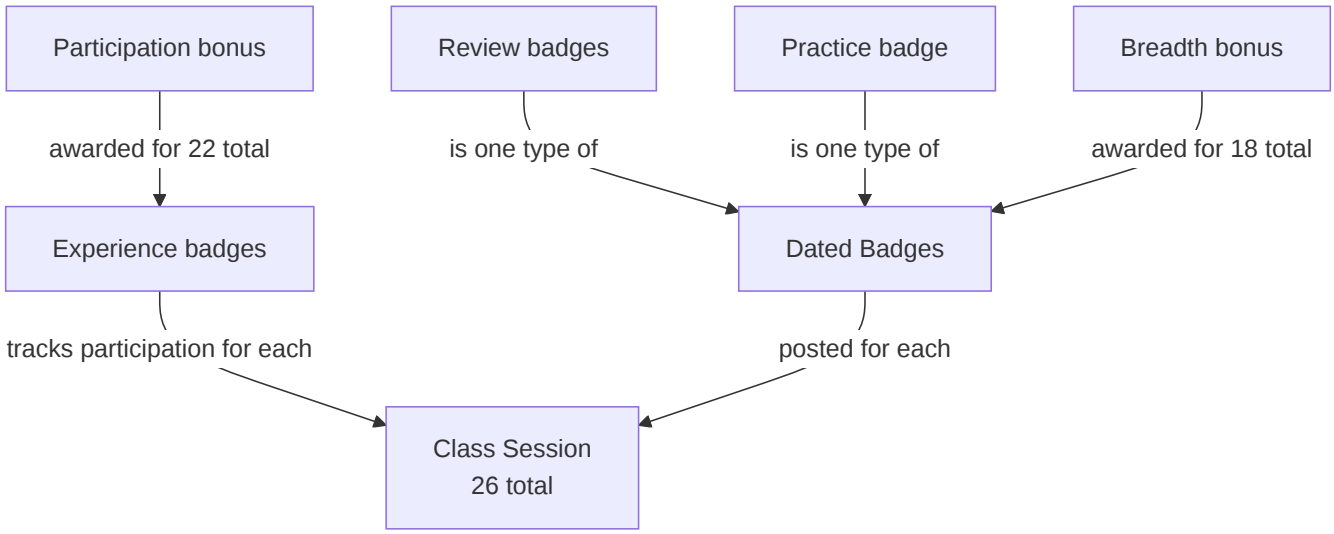
- continue getting familiar with the structure of GitHub
- clarify more how the course will flow

- practice with new vocabulary

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

conceptmap)=

2.1. Reminder about class structure



This is called a concept map, you would read it along the arrows, so this corresponds to the following bullets:

- review badges are one type of dated badge
- practice badges
- dated badges are posted for each class session
- experience badges track participation for each class session

2.2. Warm up

1. Navigate to your KWL repo
2. Find the issues tab
3. Open the prepare-2024-09-10 issue and discuss the questions with your classmates at your table

*hint: my KWL repo URL is: `https://github.com/compsys-progtools/fall24-brownsarahm`

Becoming an expert requires:

- different types of study
- pracitce and feedback to get better

2.3. Making up for action issue last week

To activate the workflows in your repo, edit them, even adding a space or deleting a blank line will make it work.

We fixed the `forgottenexperience.yml` file and then ran it manually.

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

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

note, here we are learning *by example* and then *synthesizing* that into more concrete facts.

“just play with it”

- common attitude in CS
- not optimal for learning

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

To do this:

- 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.

2.3.1. 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

2.3.2. 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

2.4. 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

2.5. 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 loook 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?

2.6. 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

2.7. How does this work?

2.7.1. In class:

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

2.7.2. Outside of class:

1. Read 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 to prepare
4. Read articles/ watch videos to either fill in gaps or learn more details
5. Bring questions to class

I give a time breakdown in the syllabus.

2.8. Prepare for next class

1. 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)
2. 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)

2.9. 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 faciltate rendering later (what we will learn will connect a lot of ideas, mapping out where you start, sets you up for success)

2.10. Questions After Class

2.10.1. How can I create my own actions to run that will create issues and pull requests?

We will learn this a bit later, but you can read ahead in the docs.

2.10.2. What is the purpose of the wiki within GitHub?

To serve as documentation, wiki-style for the repo. It was more common before github added pages. A lot of open source projects now host documentation using github pages. We will learn how this works later, but this website uses pages.

2.10.3. Do we merge the pull request for the experience report into main?

Only when approved (but if you forget, @ me and I can fix it)

2.10.4. Just still how to better navigate github because while I understood most of it was still confusing at points

We will be doing all of these things over and over, so you will get them, it's okay!

2.10.5. Is it possible to rename a committed change?

You can change a commit message, yes!

2.10.6. How do we access the old file after we already commit changes?

On GitHub.com, you go to the commits on the repo (click the ⓘ icon or add `/commits` to the repo's URL) and then choose view at that point.

2.10.7. How can we edit our github using the terminal?

Next class!

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

everything is a file

3.1. 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`
- if you have WSL (if you do not, no need to worry) you should be able to set your linux shell to `bash`

Mac warns me that `bash` is not the default, but that is okay.

```
The default interactive shell is now zsh.
To update your account to use zsh, please run `chsh -s /bin/zsh`.
For more details, please visit https://support.apple.com/kb/HT208050.
```

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

```
pwd
```

```
/Users/brownsarahm
```

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

we start at home (~)

We can **c**hange **d**irectory with `cd`

```
cd Documents/inclass/
```

We can **m**a**k** a new **dir**ectory with `mkdir`

```
mkdir systems
```

What you want to have is a folder for class (mine is systems) in a place you can find it. (mine is in my inclass folder, I have a separate teaching folder for outside of class time, like where my draft of these noes is)

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

```
cd systems/
```

```

```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems
```

```
cd
```

Next, we go back to where we want to be

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


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

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems
```

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.

```
pwd
```

```
/Users/brownsarahm/Documents/inclass
```

```
cd systems/
```

We can use multiple to go up many levels

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

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

```
cd
```

Then we go back to where we want to b

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

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems
```

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

```

```

```
pwd
```

```
/Users/brownsarahm/Documents
```

```
cd ../
```

```

```

If you start to tab complete with a nonunique set of characters

```
cd Do
```

```
Documents/ Downloads/
```

it shows you th options and puts what you had back on the propmt.

Then we can tab complete by adding enough letters to be unique.

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

3.2. 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

> **Warning**
>
> If the template link does not work, you are not in the org yet

3.3. 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 beacuse we are learning git

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

3.3.1. 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 / gitscm in GitBash through browser

we will do opiton 2 for today

3.3.1.1. GitBash (windows mostly)

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

3.3.1.2. 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

3.3.1.3. 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`

also go to office hours later to get better authentication

3.3.2. Cloning a repository

We will create a local copy by cloning

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

```
Cloning into 'gh-inclass-brownsarahm'...
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 4 (delta 0), pack-reused 0 (from 0)
Receiving objects: 100% (5/5), done.
```

Confirm it worked with:

```
ls
```

```
gh-inclass-brownsarahm
```

this folder was empty, but now we have a folder named like the repo

3.4. What is in a repo?

We can enter that folder

```
cd gh-inclass-brownsarahm/
```

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 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

`ls` can also take an explicit path as the argument to list a different folder than the current working directory.

```
ls .github/
```

```
workflows
```

We can use it with any relative path.

```
ls .github/workflows/
```

```
create_issues.yml
```

3.5. 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` directorty) to the current state of your `.git` directory.

3.6. Closing an Issue with a commit

Next, we ran the one action in the gh-inclas repo, it creates 3 issues.

Now we can make the about file.

We added a file from the code tab in browser, titled it about.md and put some content in.

In the commit messag, we used `closes #x` where `x` is the issue to close the issue.

If we look locally, we do not see the file.

```
ls
```

With git status, we see it thinks it is still up to date

```
git status
```

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

nothing to commit, working tree clean
```

This is because it only compares with the `.git` directory, not the internet. So, we can update that repo without changing the working directory with `fetch`

```
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), 960 bytes | 240.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-brownsarahm
   0373342..0e7c990  main       -> origin/main
```

Now, we can do status again.

```
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
```

now it knows we are behind

```
ls
```

```

```

but the working directory is still the same

we can use pull to update our local main branch

```
git pull
```

```
Updating 0373342..0e7c990
Fast-forward
 about.md | 3 +++
 1 file changed, 3 insertions(+)
 create mode 100644 about.md
```

and check the working directory

```
ls
```

```
about.md
```

and it is there!

3.7. 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.

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-brownsarahm
 * [new branch]      1-create-a-readme -> origin/1-create-a-readme
```

and the second, this switches branches.

```
git checkout 1-create-a-readme
```

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

again, we check the status

```
git status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

nothing to commit, working tree clean
```

now it says we are on a new branch

3.8. Creating a file on the terminal

The `touch` command creates an empty file.

```
touch README.md
```

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

```
ls
```

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

```
git status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

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

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

```
nano README.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.

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
```

and we can see the contents

```
git status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

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

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

In this case both say to `git add` to track or to include in what will be committed. Under untracked files is 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 commiting at all, and we can group many changes into a single commit.

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

```
git add .
```

And again, we will check in with git

```
git status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

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

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.

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

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

```
git commit -m 'create readme closes #1'
```

Again, we use a closing keyword so that it will close the issue.

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

one more check

```
git status
```

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

nothing to commit, working tree clean
```

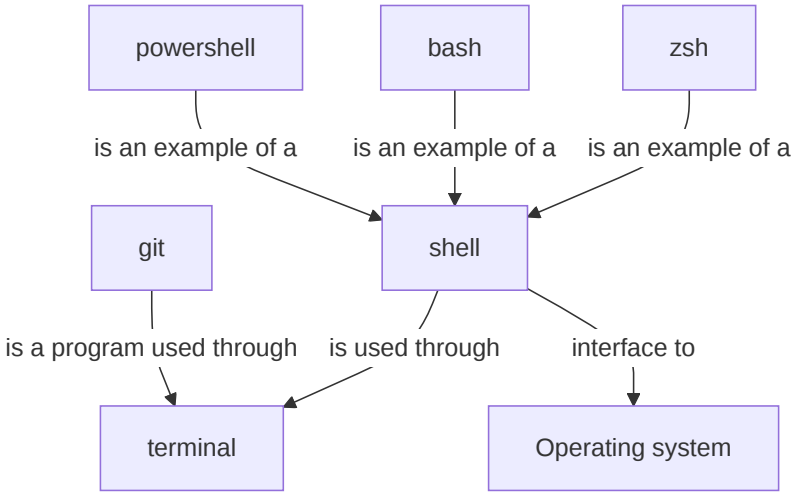
> **Warning**
>
> when you make your first commit you will need to do some config steps to set your email and user name.

and push to send to gihub.com

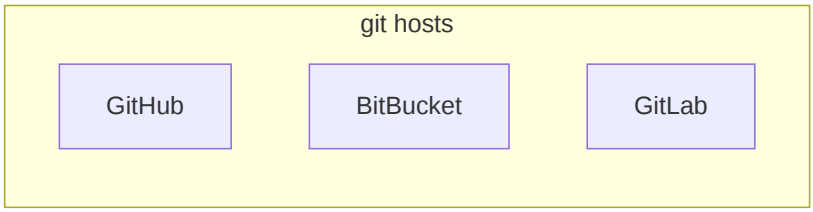
```
git push
```

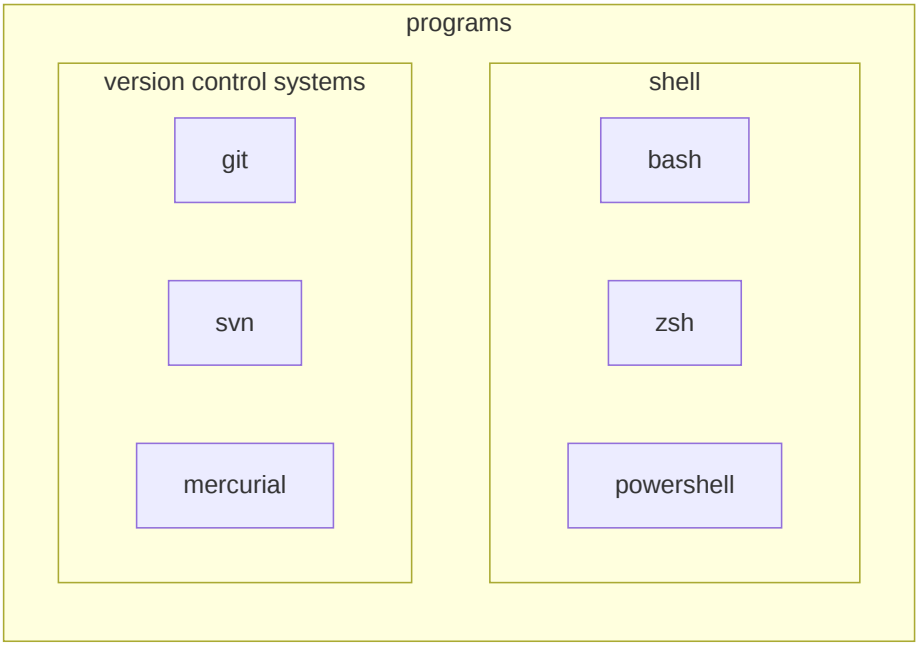
```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 353 bytes | 353.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/compsys-progtools/gh-inclass-brownsarahm.git
   0e7c990..c7375fa  1-create-a-readme -> 1-create-a-readme
```

3.9. Summary



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





Today's bash commands:

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

We also learned some git commands

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

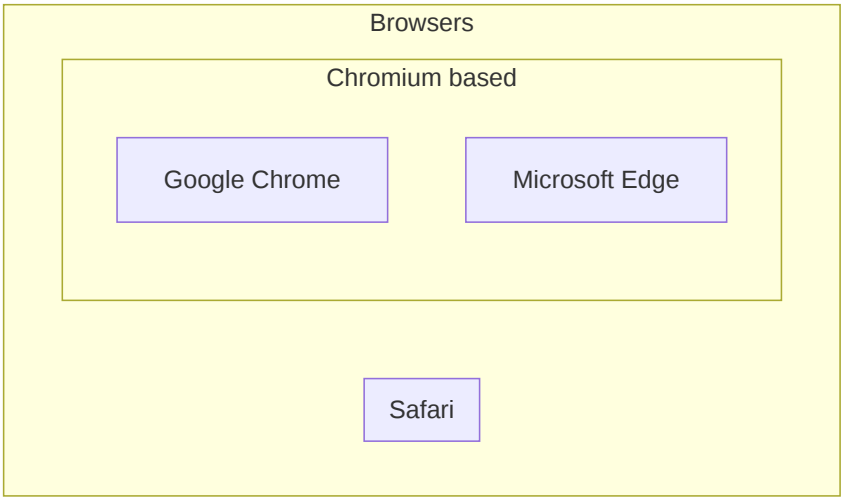
command	explanation
`push`	send to the remote

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. Experience Report Evidence

3.14. Questions After Today's Class

> **Warning**
>
> will be added

4. How do git branches work?

Today we are going to work with branches offline to begin to understand them better.

4.1. Git cheatsheets

High quality, correct refrences are important to gaining an understanding and being able to use tools well.

> **Important**
>
> Checking for factual information is not reliable in LLMs. They can be a helpful coding assistant when you know what you want, buut they generate text that cannot be fact checked.

- visual cheatsheet
- github cheatsheet in many languages
- complete official docs

We are going to work between the browser and locally in order to simulate scenarios that most often occur in collaboration, but can also occur working alone as well.

4.2. Back to the gh-inclass repo

Recall, We can move around and examine the computer's file structure using shell commands.

Open a new terminal window and navigate to your folder for the course (the one that contains other folders)

To confirm our current working directory we print it with `pwd`

```
pwd
```

```
/Users/brownsarahm
```

now change the path

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

then we can see what is there

```
ls
```

```
gh-inclass-brownsarahm
```

and finally go into the gh-inclass repo

```
cd gh-inclass-brownsarahm/
```

this confirms what we expect

Now let's see what is in our folder:

```
ls
```

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

and check in with git

```
git status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

nothing to commit, working tree clean
```

4.3. Branches do not sync automatically

Now we will look in GitHub and see what is there.

getting to GitHub from your local system

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

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

```
Opening github.com/compsys-progtools/gh-inclass-brownsarahm in your browser.
```

Once we are in the browser, we created a PR for the readme branch then merged it.

Once we merge the PR, the closing keyword that was in our commit message applies. The closinng keyword did not apply on the non default branch, but it does once that commit is added to the default branch, which, in this repo is `main`.

```
git  status
```

```
On branch 1-create-a-readme
Your branch is up to date with 'origin/1-create-a-readme'.

nothing to commit, working tree clean
```

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-brownsarahm
   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:

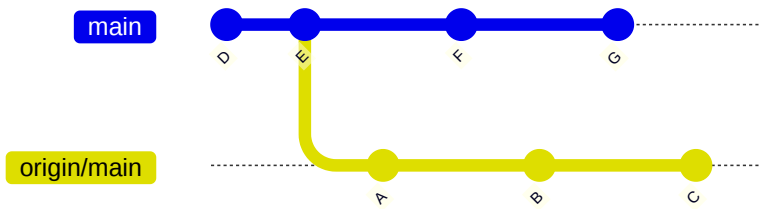
```
ls
```

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

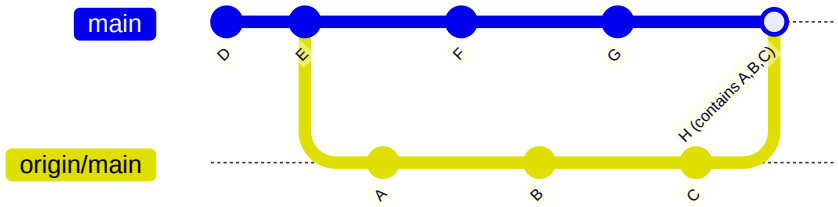
```
about.md
```

4.3.1. Git Merge

When branches need to be merged they could look like this:

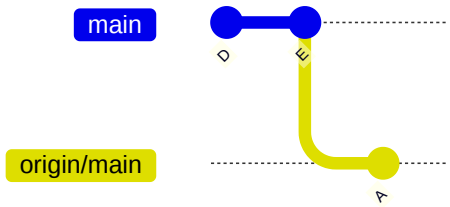


After merge, it looks like this:

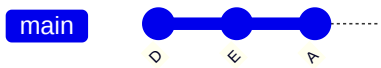


There is a new commit with the content.

In our case it is simpler:

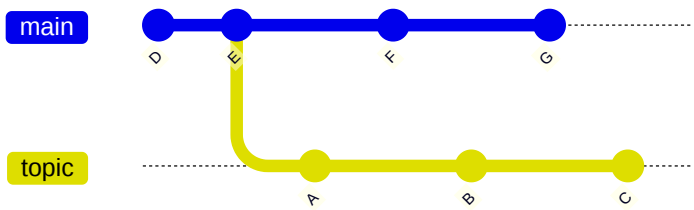


so it will fast forward

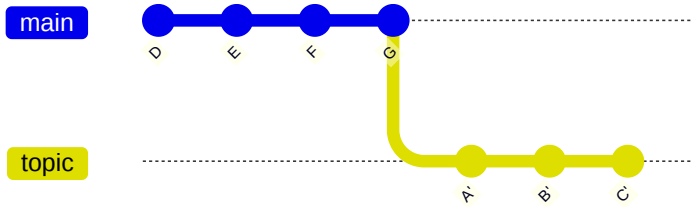


4.3.2. git rebase

If a repo is like this:



after:



4.4. 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

4.5. Viewing Commit History

We can see commits with `git log`

```
git log
```

```
commit 0c1271483e62e69b8b3fc329203617b7093413df (HEAD -> main, origin/main, origin/HEAD)
Merge: 0e7c990 c7375fa
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 17 12:50:51 2024 -0400

    Merge pull request #4 from compsys-progtools/1-create-a-readme

    create readme closes #1

commit c7375faca0043cf3c233d705201851a10e4e53ac (origin/1-create-a-readme, 1-create-a-readme)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 13:42:56 2024 -0400

    create readme closes #1

commit 0e7c990886ec282ba570b3400908ff46698e7dc0
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 13:21:42 2024 -0400

    start about file closes #3

commit 03733421ad69f816094fa62e3031a7703aa308e3
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 12:57:58 2024 -0400

    Initial commit
```

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

4.6. making a new branch locally

We've used `git checkout` to switch branches before. To also create a branch at the same time, we use the `-b` option.

A few examples:

If we use `checkout` without `-b` and an unkown branch name,

```
git checkout my_branch_checkedout
```

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

it give us an error, this does not work

but with `-b` we can make the branch and switch at the same time.

```
git checkout -b my_branch_checkedoutb
```

```
Switched to a new branch 'my_branch_checkedoutb'
```

so we see it is done!

`create` does not exist

```
git branch create my_branch_create
```

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

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

This version gives us two new observations

```
git branch my_branch; git checkout my_branch
```

```
Switched to branch 'my_branch'
```

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
* my_branch
  my_branch_checkedoutb
```

or again look at the log

```
git log
```

```
commit 0c1271483e62e69b8b3fc329203617b7093413df (HEAD -> my_branch, origin/main, origin/HEAD, my_branch_c
Merge: 0e7c990 c7375fa
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 17 12:50:51 2024 -0400

    Merge pull request #4 from compsys-progtools/1-create-a-readme

    create readme closes #1

commit c7375faca0043cf3c233d705201851a10e4e53ac (origin/1-create-a-readme, 1-create-a-readme)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 13:42:56 2024 -0400

    create readme closes #1

commit 0e7c990886ec282ba570b3400908ff46698e7dc0
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 13:21:42 2024 -0400

    start about file closes #3

commit 03733421ad69f816094fa62e3031a7703aa308e3
Author: Sarah Brown <brownsarahm@uri.edu>
```

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
```

First we'll go back to main

```
git checkout main
```

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

We will make a branch for our next task.

```
git chckeout fun_fact
```

```
git: 'chckeout' is not a git command. See 'git --help'.

The most similar command is
        checkout
```

I made a typo, but it offers a hint.

then it works when we fix the typo

```
git checkout -b fun_fact
```

```
Switched to a new branch 'fun_fact'
```

4.7. Creating a 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 th 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
```

```
# Sarah Brown

tenure year: 2027
- i skied competiively in hs
```

```
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(+)
```

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.

```
git log
```

```
commit 70759fda93a0b9c714a81f9de79df7fdfec3ab88 (HEAD -> fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 17 13:21:33 2024 -0400

    add fun fact

commit 0c1271483e62e69b8b3fc329203617b7093413df (origin/main, origin/HEAD, my_branch_checkedoutb, my_brar
Merge: 0e7c990 c7375fa
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 17 12:50:51 2024 -0400

    Merge pull request #4 from compsys-progtools/1-create-a-readme

    create readme closes #1

commit c7375faca0043cf3c233d705201851a10e4e53ac (origin/1-create-a-readme, 1-create-a-readme)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 12 13:42:56 2024 -0400

    create readme closes #1

commit 0e7c990886ec282ba570b3400908ff46698e7dc0
Author: Sarah Brown <brownsarahm@uri.edu>
```

Now we will push to github.

```
git push
```

```
fatal: The current branch fun_fact has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin fun_fact

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

It cannot push, because it does not know where to push, like we noted above that it did not comapre to origin, that was beecause it does not have an “upstream branch” or a corresponding branch on a remote server. It does not work at first because this branch does not have a remote.

To fix it, we do as git said.

```
git push --set-upstream origin fun_fact
```

```
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), 317 bytes | 317.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'fun_fact' on GitHub by visiting:
remote:      https://github.com/compsys-progtools/gh-inclass-brownsarahm/pull/new/fun_fact
remote:
To https://github.com/compsys-progtools/gh-inclass-brownsarahm.git
 * [new branch]      fun_fact -> fun_fact
branch 'fun_fact' set up to track 'origin/fun_fact'.
```

4.8. 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 second fun fact.

Then edit it locally to also have 2 fun facts.

```
nano about.md
```

```
```

```
cat about.md
```

```
# Sarah Brown

tenure year: 2027
- i skied competiively in hs
- i went to Northeastern
```

```
git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (3/3), 973 bytes | 324.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-brownsarahm
   70759fd..462402f  fun_fact   -> origin/fun_fact
Updating 70759fd..462402f
error: Your local changes to the following files would be overwritten by merge:
        about.md
Please commit your changes or stash them before you merge.
Aborting
```

It does not work because we have not committed.

This is helpful beacause it prevents us from losing any work.

```
git status
```

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

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")
```

we can add and commit at the same time using the `-a` option from the `git commit`

```
git commit -a -m 'local second fun fact'
```

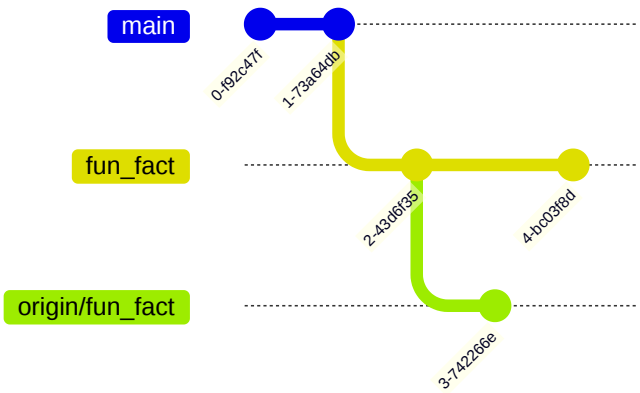
```
[fun_fact 62dcf61] local second fun fact
 1 file changed, 1 insertion(+)
```

Now we try to pull again

```
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 `fun_fact` and `origin/fun_fact` 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 second 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 second 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 second fun fact
No commands remaining.
You are currently rebasing branch 'fun_fact' 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
```

```
# Sarah Brown

tenure year: 2027
- i skied competiively in hs
<<<<<<< HEAD
- i started at uri in 2020
=======
- i went to Northeastern
>>>>>>> "local fun fact"
```

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

In this case, we will choose both, so my file looks like this in the end.

```
# Sarah Brown

tenure year: 2027
- i skied competiively in hs
- i started at uri in 2020
- i went to Northeastern
```

```
git status
```

```
interactive rebase in progress; onto 462402f
Last command done (1 command done):
   pick 62dcf61 local second fun fact
No commands remaining.
You are currently rebasing branch 'fun_fact' 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 second fun fact
No commands remaining.
You are currently editing a commit while rebasing branch 'fun_fact' 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 intructions again, and continue the rebase to combine our branches

```
git rebase --continue
```

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

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

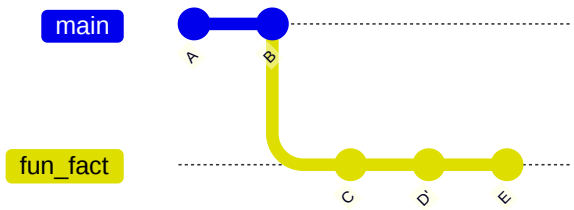
```
git push
```

```
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-brownsarahm.git
   462402f..c3e68a0  fun_fact -> fun_fact
```

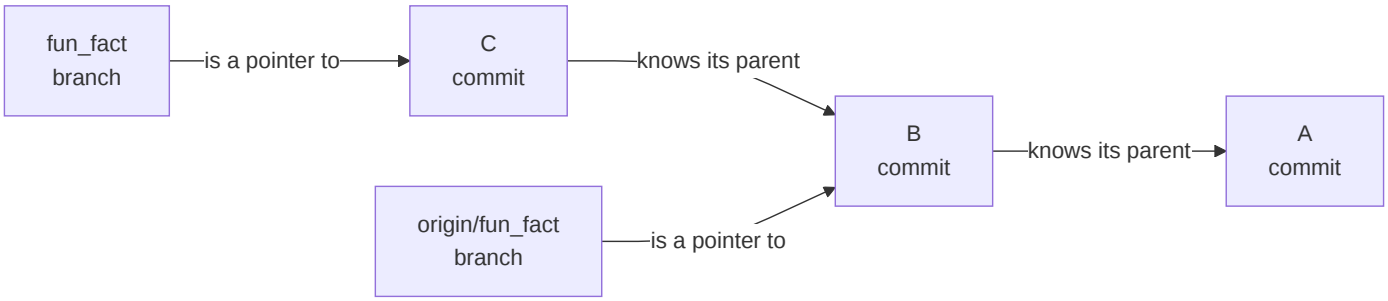
4.9. Summary

- 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.

4.9.1. New bash commands

command	explanation
`cat`	concatenate a file to standard out (show the file contents)

4.9.2. New git commands

command	explanation
`git log`	show list of commit history
`git branch`	list branches in the repo
`git branch new_name`	create a `new_name` branch
`git checkout -b new_Name`	create a `new_name` branch and switch to it
`git pull`	apply or fetch and apply changes from a remote branch to a local branch

4.10. 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

4.11. 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 regulary 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.

4.12. Experience Report Evidence

4.13. Questions After Today's Class

4.13.1. How do I know if everything I submitted was done properly, or see a grade of somesort. I know comments are left but sometimes when I check I don't really know if I did everything right.

If it's approved you completed the badge, if revisions are requested, then you need to revise.

Starting next week, we will learn how to produce a progress report.

4.13.2. Why wouldnt you just use vs code or git hub directly?

Using the terminal is generally faster once you learn it. It also allows you to write scripts.

Some things cannot be done in browser, plus when you are writing code you would be working outside of the browser.

4.13.3. How common are merging issues?

Merge conflicts are very common. It happens whenever a branch is edited in two places.

4.13.4. How can I merge the new branch locally, not from the website?

the `git merge` command

4.13.5. Is it possible to have more than one merge conflict on a line?

on a single line it will count it as a single conflict, because it counts by lines

4.13.6. What is the order of merge and rebase?

THey are two options for the same thing, but they put commits in a different order.

5. 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)

5.1. Setting the stage

To prepare for today's class you examined an open source project.

We noticed that while the contents inside and the distribution of languages used as well as the specific code, or the *content* of the files was all different, a lot of the *organization* was similar.

Most had certain community health files and basic info files:

- `CONTRIBUTING`
- `CODE_OF_CONDUCT`
- `README.MD`
- `LICENSE`
- `GOVERNANCE.MD`

5.2. Important references

Use these for checking facts and resources.

- bash
- git

5.3. Setup

First, we'll go back to our github inclass folder, locally

```
cd gh-inclass-brownsarahm/
```

And confirm we are where we want to be

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/gh-inclass-brownsarahm
```

Next get the files for today's activity:

1. Find your PR that I opened for you today that has the title, "9/19 in class activity"
2. Mark it ready for review to change from draft
3. Merge it

> **Note**
>
> in the issue PR it has the info about draft PRs in a link

Now we pull to get the files locally:

```
git pull
```

```
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 19 (delta 1), reused 17 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (19/19), 2.47 KiB | 126.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-brownsarahm
   0c12714..991ee65  main          -> origin/main
 * [new branch]      organizing_ac -> origin/organizing_ac
Already up to date.
```

Note how many things it got

and check out status

```
git status
```

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

nothing to commit, working tree clean
```

we are still on fun_fact

5.4. How does git store information?

In class, I received the following question

> Why are we on the `fun_fact` branch, not `main`?

This is because of how a `branch` is implemented and how git runs.

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

```
ls .git
```

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

We will learn more about these files later, but looking at one helps us answer the question

First we will look at the `HEAD` file

```
cat .git/HEAD
```

```
ref: refs/heads/fun_fact
```

`HEAD` is a pointer to the currently checked out branch.

The other files with `HEAD` in their name are similarly pointers to other references, named corresponding to other things.

Next we switch to main, since we want to be there for the files we merged anyway

```
git checkout main
```

```
Switched to branch 'main'
Your branch is behind 'origin/main' by 2 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

Now we can look at the `HEAD` file again

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

it changed! because one the the things that `git checkout` does is update the head pointer.

So `HEAD` is a pointer to the branch, but the branch is also a pointer to a commit

```
cat .git/refs/heads/main
```

```
0c1271483e62e69b8b3fc329203617b7093413df
```

that file has only the hash of a commit

Each of us will have a unique hash, we'll learn more about why for next week

but if we use `git log`

```
git log
```

Important

`git log` starts a program that you can exit with the `q` key.

The branch pointer matches the lastcommit where the HEAD and main pointers are. (`git log` also reads those files…)

```
commit 0c1271483e62e69b8b3fc329203617b7093413df (HEAD -> main, my_branch_checkedoutb, my_branch)
Merge: 0e7c990 c7375fa
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 17 12:50:51 2024 -0400
```

Note

This is truncated output for brevity

Now again, we can use git status

```
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
```

We're behind, so the files we merged into main are not here yet

```
ls
```

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

so, we will pull

```
git pull
```

```
Updating 0c12714..991ee65
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
```

5.5. 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 way to learn this might be to 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

For example

```
cat API.md
```

```
jupyterbook file to generate api documentation
```

But our older files are as expected

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
```

> **Note**
>
> Here we talked about how those small files do not have a new line character at the end

> **Note**
>
> On Windows, students ge the warning about CRLF and LF
>
> There is setting in git that controls it. For a good explanation, consider this stack overflow question.-,How%20autocrlf%20works%3A,-core.autocrlf%3Dtrue) about how it works.
>
> GitHub also has a whole file on how to work with this.
>
> Wikipedia's history in the Newline article notes that the CRLF that Windows uses actually comes from the era of teletype machines. Unix adopted LF alone and Apple (pre OsX) used CR alone.
>
> more OSs are described in a table

5.5.1. How does `cat` work?

```
cat --help
```

```
cat: illegal option -- -
usage: cat [-belnstuv] [file ...]
```

it doesn't have a help option, but the error still gives us the beginning of its documentation. it says there are options we can provide `[-belnstuv]` and then we pass it a file `[file...]`

If we run it without an explicit file, it uses STDIN which we can see by trying it, or from its docs

```
cat
```

each time we type things into STDIN, it outputs that

```
lksdfldkfds
lksdfldkfds
^C
```

we can ust `CTRL` + `C` (`^C` means that is what I pressed) to exit.

5.6. Files, Redirects, git restore

we will work on a branch so that we can easily recover from any mistakes

```
git checkout -b organization
```

```
Switched to a new branch 'organization'
```

and check status to ensure we are where we want to be and note that no files have been changed

```
git status
```

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

Let's review what is in the README

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
```

Echo repeats things we pass into it

```
echo "its finally fall"
```

```
its finally fall
```

since the `echo` program writes to the STDOUT file, we can change it to write to another file by redirecting it to another file.

```
echo "its finally fall" > README.md
```

```

```

There is no outut of this command

but we can look at the file

```
cat README.md
```

```
its finally fall
```

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.

As always we start by checking in

```
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")
```

it gives us a reminder, `use "git restore <file>..." to discard changes in working directory`

So we do that:

```
git restore README.md
```

then check in with git again

```
git status
```

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

and check the file

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
```

back as we wanted!

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

This could be familiar from:

- `fopen` in C
- or `open` in Python

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

We can also **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 "its finally fall" >> README.md
```

```

```

Then we check the contents of the file and we see that the new content is there.

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
its finally fall
```

We can redirect other commands too:

```
git status > curgit
```

we see this created a new file

```
ls
```

```
API.md                  abstract_base_class.py  scratch.ipynb
CONTRIBUTING.md         alternative_classes.py  setup.py
LICENSE.md              curgit                  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
```

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

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

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

```
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")
```

Now we have made some changes we want, so let's commit our changes.

```
git commit -a -m "add a note"
```

```
[organization 72b85c7] add a note
 1 file changed, 1 insertion(+)
```

```
git status
```

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

```
git status
```

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
```

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
its finally fall
|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 |
```

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

Finally, we will commit the changes

```
git commit -a -m 'explain files'
```

```
[organization f17e276] explain files
 1 file changed, 19 insertions(+)
```

5.7. Summary

- stdout is a file, that is displayed on the terminal
- “moving” a file does not re-write the data to a different part of the disk, it updates its adress only
- the wildcard operator `*` allows us to use patterns with bash commands
- `touch` can accept a list of files
- bash lists are space delimited without any brackets
- the `.gitignore` file prevents files from being in your repo

5.7.1. New commands

command	explanation
`echo 'message'`	repeat ‘message’ to stdout
`>`	write redirect
`>>`	append redirect
`rm file`	remove (delete) `file`

5.7.2. New git commands

command	explanation
`git commit -a -m 'msg'`	the `-a` option adds modified files (but not untracked)

5.8. 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 and `VioletVex` is already requested as a reviewer when your experience badge (inclass) 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. Hints: Look for bash commands that we have seen before and `cp` copies a file.

5.9. Badges

Review | **Practice**

1. Update your KWL chart with the new items and any learned items.
2. Clone the course website. Append the commands used and the contents of your `fall2024/.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`)

5.10. Experience Report Evidence

Save your history with:

```
history > activity-2024-09-19.md
```

5.11. Questions After Today's Class

5.11.1. Would using > and >> locally create a merge conflict if the edited file also has a merge to it on Github?

any edit to a file could create a merge conflict, so yes one wiht a redirect can

5.11.2. I would like to learn more about adding text into files, for example if you can add text in between specific lines when writing to a file

This requires more complex commands than we will use in class time, but it can be done with **`sed`**. A tutorial on this could be a good explore badge.

5.11.3. If I forgot to restore something for a couple of days and then chose to restore it a while later would I still be able to or is there a time limit to doing so?

There is not a *time* limit, but if you did other operations it can require different commands beside `restore`

5.11.4. Can bash be used to trigger cursor events like mouse clicks?

I do not think so, but a headless browser allows you to automate browser operations, including selecting things that are most typically done by users with a mouse click, because in the browser clickable things are labeled in other ways.

This is also a good explore option.

5.11.5. What is different between using echo or nano to add to a file?

Adding to a file with `echo` and a redirect is nice for adding small changes, but not extnsive changes or changes to the middle of a file. `nano` is a full text editor. In class today, `echo` was an easy way to demonstrate redirects, because it is so simple, but redirects are more powerful with more powerful commands.

5.11.6. A question that I have is about git pull, does it run git fetch beforehand to retrieve new content from the online repo Or is git pull just a completely different pocess to bring down content?

> … precisely, `git pull` runs `git fetch` with the given parameters and then depending on configuration options or command line flags, will call either `git rebase` or `git merge` to reconcile diverging branches.
>
> —official docs

example with diagrams in last class notes

5.11.7. How do I create a community badge?

instructions in syllabus

6. Patterns in git and bash

We will continue working in the `gh-inclass` repo

```
cd gh-inclass-brownsarahm/
```

We added a bunch of files so that we can orgaize them.

```
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
```

We left off on a dedicated branch with a clean working tree.

```
git status
```

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

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/gh-inclass-brownsarahm
```

```
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
```

We put an overview in the README:

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
its finally fall
|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 |
```

6.1. Making a folder

First, we'll make a directory with `mkdir`

```
mkdir docs
```

next we will move a file there with `mv`

```
mv philosophy.md docs/
```

move takes 2 inputs: a source and destination.

we can see what happened

```
ls
```

```
API.md                  abstract_base_class.py  setup.py
CONTRIBUTING.md         alternative_classes.py  tests_alt.py
LICENSE.md              docs                    tests_helpers.py
README.md               example.md              tests_imp.py
_config.yml             helper_functions.py     tsets_abc.py
_toc.yml                important_classes.py
about.md                scratch.ipynb
```

```
ls docs/
```

```
philosophy.md
```

what this does is change the path of the file from `.../github-inclass-brownsarahm/philosophy.md` to `.../github-inclass-brownsarahm/docs/philosophy.md`

git does not quite understand though

```
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:    philosophy.md

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

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

it sees a missing file (thinks it is deleted) and a new folder

if we stage the folder, it will start tracking it

```
git add docs/
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   docs/philosophy.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:    philosophy.md
```

now it sees a new file and a deleted one

if we add everything

```
git add .
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    philosophy.md -> docs/philosophy.md
```

now it can tell we renamed

6.2. Undoing a change, only in git

let's first make change to the README

```
echo " ">>README.md
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    philosophy.md -> docs/philosophy.md

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
```

we can add the readme too

```
git add .
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
        renamed:    philosophy.md -> docs/philosophy.md
```

If we decide to not put these two changes in the same commit, we can *unstage* the file without losing our changes with `restore` and its `--staged` option

```
git restore --staged README.md
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        renamed:    philosophy.md -> docs/philosophy.md

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
```

Now only the philosophy change will be commited

Now we make that commit

```
git commit -m 'start organizing (move philosophy)'
```

```
[organization 9120d9d] start organizing (move philosophy)
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename philosophy.md => docs/philosophy.md (100%)
```

Now we can see what git knows

```
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")
```

and we will commit the README changes

W

I ch
here
one
read

```
git commi -a -m 'fill in description more (README)'
```

```
git: 'commi' is not a git command. See 'git --help'.

The most similar commands are
        commit
        column
        config
```

first I made a typo, but again, `git` tries to help

```
git commit -a -m 'fill in description more (README)'
```

```
[organization 4ceb150] fill in description more (README)
 1 file changed, 1 insertion(+)
```

now we have a clean working tree

```
git status
```

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

6.3. Undoing a commit

We will start by looking at our commit history

```
git log
```

```
commit 4ceb1500582236e98bdb141116821a5857f75a76 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:44:39 2024 -0400

    fill in description more (README)

commit 9120d9d88aa587e4ffda1ee9aa8c3dcf8f764f7e
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:44:06 2024 -0400

    start organizing (move philosophy)

commit f17e276f43e36a92dd6062cb9e2dae938870c38b
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 19 13:42:19 2024 -0400

    explain files

commit 72b85c7834afb148e1298c153a7bad423e995ce0
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 19 13:33:44 2024 -0400

    add a note

commit 991ee65fa0d0692bd097915daec156aa95eba82f (origin/main, origin/HEAD, main)
```

> **Important**
>
> this opens a program so we press `q` to exit.

recall from the output above, that commit `9120d9` above was like:

```
[organization 9120d9d] start organizing (move philosophy)
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename philosophy.md => docs/philosophy.md (100%)
```

it moved `philosophy.md` to `docs/philosophy.md` and

Now let's undo the organizing one, but keep the reame one my hash begins with `9120d9` so I pass that to `git revert` but you will pass a different hash

```
git revert 9120d9
```

git revert requires a message so vimwill open use `esc` to be sure you are in command mode then type `:wq` and press `enter`/`return` to exit vim

```
[organization a3904a0] Revert "start organizing (move philosophy)"
 1 file changed, 0 insertions(+), 0 deletions(-)
 rename docs/philosophy.md => philosophy.md (100%)
```

note that this is a new `commit, here hash` a3904a0 `and it does the **opposite** of the  one we reverted` 9120d9 `, so it moves` docs/philosphy.md `to` philosophy.md`

```
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
```

With the commit history we can see more clearly that it adds a new commit that is the opposite.

```
git log
```

```
commit a3904a0a5e7adbcbf9fe439c387fb4dbd7846c51 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:46:19 2024 -0400

    Revert "start organizing (move philosophy)"

    This reverts commit 9120d9d88aa587e4ffda1ee9aa8c3dcf8f764f7e.

commit 4ceb1500582236e98bdb141116821a5857f75a76
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:44:39 2024 -0400

    fill in description more (README)

commit 9120d9d88aa587e4ffda1ee9aa8c3dcf8f764f7e
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:44:06 2024 -0400

    start organizing (move philosophy)

commit f17e276f43e36a92dd6062cb9e2dae938870c38b
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 19 13:42:19 2024 -0400

    explain files
```

6.4. More moving files

```
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
```

we do actually want that folder so we make it again

```
mkdir docs
```

and move the file there

```
mv philosophy.md docs/
```

we can move more files more than one at a time by listing multiple paths it moves all but the last one to the last one, which must be a directory, not a file.

```
mv about.md API.md example.md docs/
```

we can see that it works

```
ls
```

```
CONTRIBUTING.md         alternative_classes.py  tests_alt.py
LICENSE.md              docs                    tests_helpers.py
README.md               helper_functions.py     tests_imp.py
_config.yml             important_classes.py    tsets_abc.py
_toc.yml                scratch.ipynb
abstract_base_class.py  setup.py
```

6.5. Moving with patterns

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 use `ls` to see it

```
ls
```

```
CONTRIBUTING.md         docs                    tests_alt.py
LICENSE.md              helper_functions.py     tests_helpers.py
README.md               important_classes.py    tests_imp.py
abstract_base_class.py  scratch.ipynb           tsets_abc.py
alternative_classes.py  setup.py
```

and in the folder:

```
ls docs/
```

```
API.md          _toc.yml        example.md
_config.yml     about.md        philosophy.md
```

6.6. Move is also rename

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
```

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.

Next we make a folder for them

```
mkdir tests
```

and move all of the test files there:

If we press `tab` multiple times it shows us what matches the pattern

```
mv tests
```

```
tests/            tests_alt.py      tests_imp.py
tests_abc.py      tests_helpers.py
```

the folder is in there, so if we were to do `mv tests* tests/` it would give us an error because the folder also matches `tests*`

If we add the `_`

```
mv tests_
```

now it’s only what we want to move:

```
tests_abc.py      tests_alt.py      tests_helpers.py  tests_imp.py
```

so we run this

```
mv tests_* tests/
```

```
ls
```

```
CONTRIBUTING.md         alternative_classes.py  scratch.ipynb
LICENSE.md              docs                    setup.py
README.md               helper_functions.py     tests
abstract_base_class.py  important_classes.py
```

```
ls tests/
```

```
tests_abc.py            tests_helpers.py
tests_alt.py            tests_imp.py
```

Finally, we wil move the rest of they python files

```
mkdir src
```

```
mv *_*.py
```

```
mv: important_classes.py is not a directory
```

without the target specified it tried to treat the last one as the destination and it did not work

```
mv *_*.py src/
```

```

```

```
ls
```

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

6.7. Hidden files and Ignoring 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

These files will not show by default

```
ls
```

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

but do with thw `-a` option

```
ls -a
```

```
.               .github         CONTRIBUTING.md docs            src
..              .gitignore      LICENSE.md      scratch.ipynb   tests
.git            .secret         README.md       setup.py
```

gitignore lets us *not* track certain files

```
nano .gitignore
```

we will ignore the secret file and all notebook files

```
.secrete
*.ipynb
```

```
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:    API.md
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    about.md
        deleted:    abstract_base_class.py
        deleted:    alternative_classes.py
        deleted:    example.md
        deleted:    helper_functions.py
        deleted:    important_classes.py
        deleted:    philosophy.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/
        src/
        tests/

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

again we will add all so that git can see we moved files

```
git add .
```

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   .gitignore
        renamed:    API.md -> docs/API.md
        renamed:    _config.yml -> docs/_config.yml
        renamed:    _toc.yml -> docs/_toc.yml
        renamed:    about.md -> docs/about.md
        renamed:    example.md -> docs/example.md
        renamed:    philosophy.md -> docs/philosophy.md
        renamed:    abstract_base_class.py -> src/abstract_base_class.py
        renamed:    alternative_classes.py -> src/alternative_classes.py
        renamed:    helper_functions.py -> src/helper_functions.py
        renamed:    important_classes.py -> src/important_classes.py
        renamed:    tsets_abc.py -> tests/tests_abc.py
        renamed:    tests_alt.py -> tests/tests_alt.py
        renamed:    tests_helpers.py -> tests/tests_helpers.py
        renamed:    tests_imp.py -> tests/tests_imp.py
```

better !

and note that the .secret file is not in the list, git is *ignoring* it.

we will make a commit now

```
git commit -m 'organized files into foleders and ignore private'
```

```
[organization d2d1fac] organized files into foleders and ignore private
 15 files changed, 2 insertions(+)
 create mode 100644 .gitignore
 rename API.md => docs/API.md (100%)
 rename _config.yml => docs/_config.yml (100%)
 rename _toc.yml => docs/_toc.yml (100%)
 rename about.md => docs/about.md (100%)
 rename example.md => docs/example.md (100%)
 rename philosophy.md => docs/philosophy.md (100%)
 rename abstract_base_class.py => src/abstract_base_class.py (100%)
 rename alternative_classes.py => src/alternative_classes.py (100%)
 rename helper_functions.py => src/helper_functions.py (100%)
 rename important_classes.py => src/important_classes.py (100%)
 rename tsets_abc.py => tests/tests_abc.py (100%)
 rename tests_alt.py => tests/tests_alt.py (100%)
 rename tests_helpers.py => tests/tests_helpers.py (100%)
 rename tests_imp.py => tests/tests_imp.py (100%)
```

```
git status
```

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

we can check the files too

```
ls -a
```

```
.               .github         CONTRIBUTING.md docs            src
..              .gitignore      LICENSE.md      scratch.ipynb   tests
.git            .secret         README.md       setup.py
```

we still have the file but git does nto see it

we can confirm by pushing

```
git push
```

```
fatal: The current branch organization has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin organization

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

but we have to pair this branch to a remote

```
git push --set-upstream origin organization
```

```
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 8 threads
Compressing objects: 100% (18/18), done.
Writing objects: 100% (20/20), 2.55 KiB | 2.55 MiB/s, done.
Total 20 (delta 8), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (8/8), completed with 1 local object.
remote:
remote: Create a pull request for 'organization' on GitHub by visiting:
remote:      https://github.com/compsys-progtools/gh-inclass-brownsarahm/pull/new/organization
remote:
To https://github.com/compsys-progtools/gh-inclass-brownsarahm.git
 * [new branch]      organization -> organization
branch 'organization' set up to track 'origin/organization'.
```

in our browser we notice the file is not there, but the .ipynb file still is

let's try editing it

```
nano scratch.ipynb
```

and checkign with git

```
git status
```

```
On branch organization
Your branch is up to date with 'origin/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:   scratch.ipynb

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

it is still tracking, because `.gitignore` does not remove files from tracking it only prevents tracking from starting

we can learn about git commands with `--help`

```
git rm --help
```

it is a program we exit with `q`

we want to remove it from git, but not delete it locally, so we use `--cached` option on `git rm`

```
git rm --cached scratch.ipynb
```

```
rm 'scratch.ipynb'
```

we can still see it

```
ls
```

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

```
git status
```

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

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        deleted:    scratch.ipynb
```

git thinks it is deleted

so we commit this

```
git commit -m 'stop tracking'
```

```
[organization 87c72ae] stop tracking
 1 file changed, 1 deletion(-)
 delete mode 100644 scratch.ipynb
```

now we can edit it more

```
echo "a change" >> scratch.ipynb
```

and check git

```
git status
```

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

nothing to commit, working tree clean
```

it dose not see!

```
echo "again change" >> scratch.ipynb
```

```
git status
```

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

nothing to commit, working tree clean
```

doesn't see it!

```
git push
```

```
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (2/2), 229 bytes | 229.00 KiB/s, done.
Total 2 (delta 1), reused 0 (delta 0), pack-reused 0 (from 0)
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
To https://github.com/compsys-progtools/gh-inclass-brownsarahm.git
   d2d1fac..87c72ae  organization -> organization
```

and we note that it is now removed online.

6.8. Copying a file

`cp` copies

```
cp README.md docs/overview.md
```

```
git status
```

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

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

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

```
git add .
```

```

```

```
git commit -m 'include readmen content'
```

```
[organization 1e2ab92] include readmen content
 1 file changed, 24 insertions(+)
 create mode 100644 docs/overview.md
```

We discussed the solution to the prepare,

- edith `.templates/experience-refletion` for your name
- add `reviwers:VioletVex` at the end of `.github/workflows/experienceinclass.yml`

6.9. 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 t
```

6.10. Badges

Review | **Practice**

3. **lab** Organize the provided messy folder in a Codepsace (details will be provided in lab time). Commit and push the changes. Answer the questions below in your kwl (this) repo in a file called `terminal_organization.md`
4. clone your `messy_repo` locally and append the `history.md` file to your `terminal_organization.md` using a redirect

```
# Terminal File moving reflection

1. How was this activity overall?
1. Did this get easier toward the end?
4. When do you think that using the terminal will be better than using your GUI file explorer?
5. What questions/challenges/ reflections do you have after this?
```

6.11. Experience Report Evidence

append your gh inclass `git log` and `history` from the abovce to a file `evidence-2024-09-24.md` on the branch in your `fall24-` repo

6.12. Questions After Today's Class

6.12.1. What are some other useful git bash commands to ensure safety in

removing, ignoring, moving, renaming, and making?

We have covered a lot of the *most* commonly used ones at this point. We will learn some more, but also looking up different people's bash or git cheatsheets is a good contribution to the site for a community badge. Comparing them extensively could be an explore badge

6.12.2. What are other special files in git?

Next week we will dig into how git works under the hood and that will see more

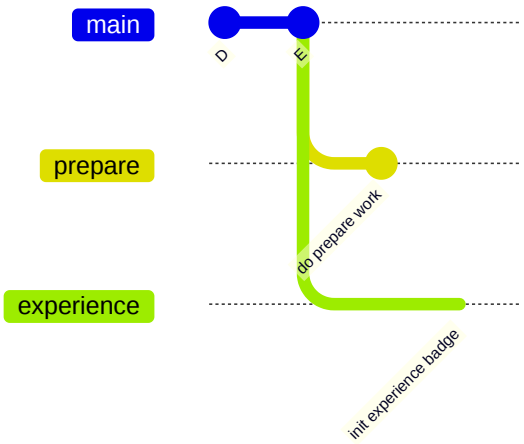
6.12.3. is .gitignore something standard from github or just a git bash thing?

it is standard to *git* no matter what host (github, bitbucket, gitlab) or terminal (mac termnial, gitbash, anaconda prompt) or shell (bash, zsh, powershell) you use.

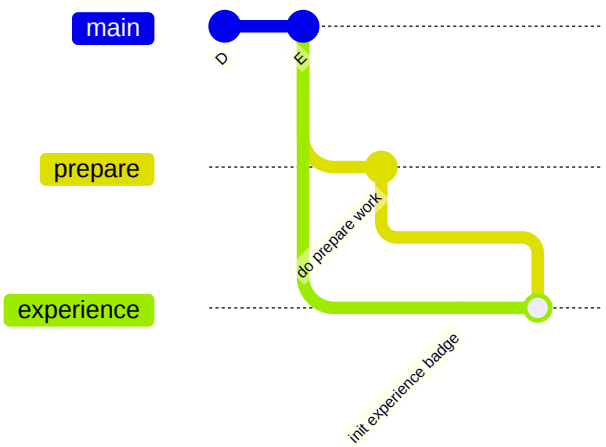
6.12.4. How do I combine pull requests

Pull requests compare two branches.

For example you might have



and have one PR from prepare to main and another from experience to main. If you change the base of the prepare one to be expereince and then merge it, then you have:



and everything will show in the experience PR

7. Why are these tools like this?

7.1. Discussion

At your tables:

- share your responses to the work in the prep work for today's topic
- discuss any similiarities 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?

7.2. 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, disucssing some principles and seeing how you have already interacted wit 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.

7.2.1. 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

7.2.2. 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
- I will not take a “great men of history” approach

This is because:

- I think that history is important context for making decisions
- Many of these “great men” are actually, in many ways, Not Great^TM
- It *is* important to remember that all of this work was done by *people*
- all people are imperfect
- 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

7.3. 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

Today we will work in your main repo

```
cd fall24-brownsarahm/
```

7.4. Philosophy in practice, using pipes

open a codespace on the main branch of your KWL repo

To check if your `gh` CLI is working:

```
gh
```

```
  Work seamlessly with GitHub from the command line.

USAGE
  gh <command> <subcommand> [flags]

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

GITHUB ACTIONS COMMANDS
  cache:       Manage GitHub Actions caches
  run:         View details about workflow runs
  workflow:    View details about GitHub Actions workflows

EXTENSION COMMANDS
  classroom:   Extension classroom

ALIAS COMMANDS
  co:          Alias for "pr checkout"

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

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

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

EXAMPLES
  gh issue create
  gh repo clone cli/cli
  gh pr checkout 321

LEARN MORE
  Use `gh <command> <subcommand> --help` for more information about a command.
  Read the manual at https://cli.github.com/manual
  Learn about exit codes using `gh help exit-codes`
```

When we use it without a subcommand, it gives us help output a list of all the commeands that we can use. If it is *not* working, you would get a `command not found` error.

Let's inpsect the action file that creates the issues for your badges.

```
cat .github/workflows/getassignment.yml
```


```
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

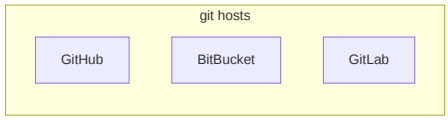
    # Install dependencies
    - name: Set up Python
      uses: actions/setup-python@v5
      with:
        python-version: 3.12

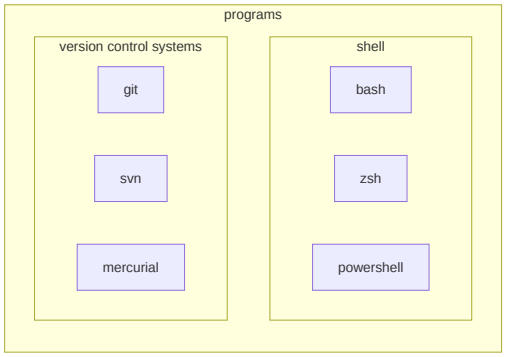
    - name: Install Utils
      run: |
        pip install git+https://github.com/compsys-progtools/
    - name: Get badge requirements
      run: |
        # prepare badge lines
        pretitle="prepare-"$(cspt getbadgedate --prepare)
        cspt getassignment --type prepare | gh issue create -
        # review badge lines
        rtitle="review-"$(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 pract
# added comment to fix
```

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

Key terms that were disambiguated today

- terminal
- shell
- git
- bash





A lot of this is familiar, but we are focused today on the three highlighed 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 some custom code.

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.

7.4.1. Installing from source

As we saw in the action, we can install python packages from source via git. let's install the new version of the code

```
pip install git+https://github.com/compsys-progtools/courseutils@main
```

```
Collecting git+https://github.com/compsys-progtools/courseutils@main
  Cloning https://github.com/compsys-progtools/courseutils (to revision main) to /private/var/folders/8g/
  Running command git clone --filter=blob:none --quiet https://github.com/compsys-progtools/courseutils /
  Resolved https://github.com/compsys-progtools/courseutils to commit 395ac8754c55a119ffa2b3670afabfe4ba0
  Preparing metadata (setup.py) ... done
Requirement already satisfied: Click in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: pandas in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.1
Requirement already satisfied: lxml in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12/
Requirement already satisfied: pyyaml in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.1
Requirement already satisfied: numpy in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3.12
Requirement already satisfied: requests in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3
Requirement already satisfied: html5lib in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3
Requirement already satisfied: six>=1.9 in /Library/Frameworks/Python.framework/Versions/3.12/lib/python3
Requirement already satisfied: webencodings in /Library/Frameworks/Python.framework/Versions/3.12/lib/pyt
Requirement already satisfied: python-dateutil>=2.8.2 in /Library/Frameworks/Python.framework/Versions/3.
Requirement already satisfied: pytz>=2020.1 in /Library/Frameworks/Python.framework/Versions/3.12/lib/pyt
Requirement already satisfied: tzdata>=2022.7 in /Library/Frameworks/Python.framework/Versions/3.12/lib/p
Requirement already satisfied: charset-normalizer<4,>=2 in /Library/Frameworks/Python.framework/Versions/
Requirement already satisfied: idna<4,>=2.5 in /Library/Frameworks/Python.framework/Versions/3.12/lib/pyt
Requirement already satisfied: urllib3<3,>=1.21.1 in /Library/Frameworks/Python.framework/Versions/3.12/l
Requirement already satisfied: certifi>=2017.4.17 in /Library/Frameworks/Python.framework/Versions/3.12/l
Building wheels for collected packages: syscourseutils
  Building wheel for syscourseutils (setup.py) ... done
  Created wheel for syscourseutils: filename=syscourseutils-1.0.6-py3-none-any.whl size=21580 sha256=9efc
  Stored in directory: /private/var/folders/8g/px8bm7bj0_j31j71yh6mfd_r0000gn/T/pip-ephem-wheel-cache-6zv
Successfully built syscourseutils
Installing collected packages: syscourseutils
  Attempting uninstall: syscourseutils
    Found existing installation: syscourseutils 1.0.6
    Uninstalling syscourseutils-1.0.6:
      Successfully uninstalled syscourseutils-1.0.6
Successfully installed syscourseutils-1.0.6
```

TO confirm it worked, we use its base command

```
cspt
```

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

Options:
  --help  Show this message and exit.

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

Let's try the one we saw in the action

```
cspt getassignment --type prepare
```

```
404: Not Found
```

With default settings, it says 404.

We know that it goes online. So, for example, if you turn your wifi off and then try it again, you will get a different error.

To learn more about this, lets use the `--help` option.
A lot of CLI tools have this option. In this program, that I made, the Python library `click` that I used to make this, provides the `help` option automatically to show the documentation or lets me as the developer add help text to options.

```
cspt getassignment --help
```

```
Usage: cspt getassignment [OPTIONS]

  get the assignment text formatted

Options:
  --type TEXT  type can be {prepare, review, or practice}; default prepare
  --date TEXT  date should be YYYY-MM-DD of the tasks you want; default most
               recently posted
  --help       Show this message and exit.
```

Now we can see that it can take some options.

If we do not provide a date, I can tell you (and I should add to the documentation), it uses today's date. Since we ran this on a day with class, before the badges were posted, the file it looked for did not exist, so we got 404.

We can use the options to get the last posted prepare work

```
cspt getassignment --type prepare --date 2024-09-26
```

```
- [ ] Think through and make some notes about what you have learned about design so far. Try to answer th

```
- What past experiences with making decisions about design of software do you have?
- what experiences studying design do you have?
- What processes, decisions, and practices come to mind when you think about designing software?
- From your experiences as a user, how you would describe the design of command line tools vs other GUI b
```
```

```
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`
```

7.4.2. Interactive Design

```
gh issue create
```

```
Creating issue in compsys-progtools/compsys-progtools-fa24-fall24-kwl-template

? Title tst
? Body <Received>
? What's next? Submit
https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-template/issues/43
```

```
gh issue list
```

```
Showing 1 of 1 open issue in compsys-progtools/compsys-progtools-fa24-fall24-kwl-template

ID   TITLE  LABELS  UPDATED
#43  tst            less than a minute ago
```

```
gh issue view 43
```

```
tst compsys-progtools/compsys-progtools-fa24-fall24-kwl-template#43
Open • brownsarahm opened about 1 minute ago • 0 comments


  No description provided


View this issue on GitHub: https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-templat
```

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-fa24-fall24-kwl-template/issues/43#issuecomment-23
```

We can look again

```
gh issue view 43
```

```
tst compsys-progtools/compsys-progtools-fa24-fall24-kwl-template#43
Open • brownsarahm opened about 2 minutes ago • 1 comment


  No description provided


brownsarahm • 0m • Newest comment

  lksjflakjfladksjlf;kwhf;kwh


View this issue on GitHub: https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-templat
```

We can also close issues

```
gh issue close 43
```

```
✓ Closed issue compsys-progtools/compsys-progtools-fa24-fall24-kwl-template#43 (tst)
```

7.5. Now with the pipe

```
cspt getassignment --type prepare --date 2024-09-26 | gh issue create --title 'duplicatee' --body-file -
```

```
Creating issue in compsys-progtools/compsys-progtools-fa24-fall24-kwl-template

https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-template/issues/44
```

we can aloso open t in browser to see more fully

```
gh issue view 44 --web
```

```
Opening github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-template/issues/44 in your browser
```

Important

If you are using GitBash on Windows see the mintty info to make `gh` work locally.

Since your repos are forks, they have two remotes, or upstream repositories, that you can pull from.

We can see with `git remote`

```
git remote
```

```
origin
upstream
```

it has a `verbose` option with the `-v` flag that shows more detail

```
git remote -v
```

```
origin  https://github.com/compsys-progtools/fall24-brownsarahm.git (fetch)
origin  https://github.com/compsys-progtools/fall24-brownsarahm.git (push)
upstream        https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-template.git (fetc
upstream        https://github.com/compsys-progtools/compsys-progtools-fa24-fall24-kwl-template.git (push
```

We can change which of those is a default

```
gh repo set-default origin
```

```
expected the "[HOST/]OWNER/REPO" format, got "origin"
```

I forgot the format and tried to use the short name, `origin` when `gh` requires it tobe in `owner/repo` format, but this error is informative, so we know what to do:

Important

Do this step!! It will fix some of your errors

````
```{code-cell} bash
:tags: ["skip-execution"]
gh repo set-default compsys-progtools/fall24-brownsarahm
```
````

```
✓ Set compsys-progtools/fall24-brownsarahm as the default repository for the current directory
```

Hack the course

> build and explore ideas

Develop new command line tools, github actions, VS Code (or other IDE) extensions, visual aids, etc that help future students in the course. This is a way for you to demonstrate your learning and contribute to open source repos that you can then link to on a portfolio website or resume.

You can participate in this at different levels (or multiple):

- build: writing, documenting at the API level, and configuring major pieces (can be collaborative)
- explore: writing example/tutorial style docs that another student (or team) build (mostly solo; open to justification for collab)
- explore: adding summary/visuals to the notes
- community: (test option): testing other students contributions and making descriptive issues or writing short reviews (~3-5 sentenccs); must not be spammy
- community: (review option): reviewing PRs submitted by a classmate (can be within a collaborative build, but must be not your own code) a single PR can have multiple reivews if sensible and not duplicative; no spam
- community: (contribution) add a glossary term to this site

see the bonus table for more information

Prepare for Next Class

1. Learn about hacktoberfest
2. check your plan for success PR for comments and reply or merge if approved
3. read about conventional commits and find some opinions about them in dev blogs, forums (eg reddit) or similar

Badges

Review | **Practice**

1. Read today's notes when they are posted. There are imporant 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 <Adherance/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.

Experience Report Evidence

Questions After Today's Class

Is there anything you can do in Github that you can't do locally?

View PRs linked to issues easily is one example. It's not a lot though

What should I do for prepare work?

see instructions

What is the `cspt` in get an assignment and what exactly is the `getassignment` command doing?

see its docs or source

Why did we type in the codepsace terminal on Git Hub and not on Bash?

the terminal in the codespace is `bash` on linux, where you all have *exactly* the same specs.

GitBash, the windows terminal program that allows you to use `bash` and `git` commands has and issue with mintty that you probably have to reinstall it to make `gh` work there, but the GitHub Codespace comes with `gh` installed.

Why did we use the pip install command?

`pip` is the package manager for python. `pip install` installs a package. We installed from the source repo

What are some examples of dev tools that we can use today

All the ones we use each class, every IDE you use, etc.

When should I use `--help` option?

Whenever you do not know the usage of a command. Not all programs provide it, but many modern ones do. Worst case it gives an error.

Why was the default repo set wrong?

That is the normal choice for github for forks, that issues go to the primary repo, not the fork. It is just not what we want for this class (we did not really want forks, but GitHub classrooms makes them now).

8. What *is* a commit?

8.1. 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.

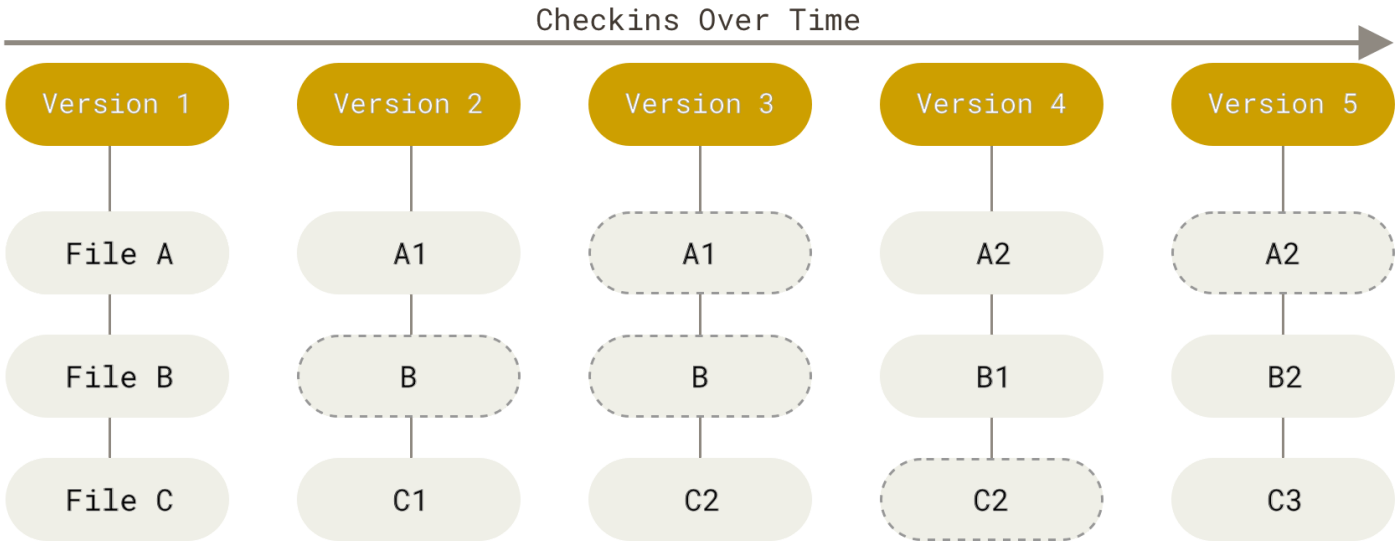
Now we need to contend with multiple definitions

These definitons 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.

8.2. 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.

8.3. 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

8.4. 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

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

We can use `git log` to view past commits

```
git log
```

```
commit 1e2ab9259651a73ad277e826d602514d28969c86 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:30:44 2024 -0400

    include readmen content

commit 87c72aeca9bd16700fc8fd8ee719136c13e83e01 (origin/organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:23:36 2024 -0400

    stop tracking

commit d2d1fac72642204bfdcebc7703d786615b8de934
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:16:10 2024 -0400

    organized files into foleders and ignore private

commit a3904a0a5e7adbcbf9fe439c387fb4dbd7846c51
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:46:19 2024 -0400

    Revert "start organizing"

    This reverts commit 9120d9d88aa587e4ffda1ee9aa8c3dcf8f764f7e.
```

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 first visible commit above, that is highlighted

`git cat-file` has different modes:

- `-p` for pretty print
- `-t` to return the type

```
git cat-file -p 1e2a
```

```
tree 7c055c5ff9309a982982db0b890bc2a02926d7e3
parent 87c72aeca9bd16700fc8fd8ee719136c13e83e01
author Sarah M Brown <brownsarahm@uri.edu> 1727199044 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1727199044 -0400

include readmen content
```

Here we see the actual parts of a commit file:

- a pointer to a tree
- a pointer to a parent commit (highlighted)
- author info with timestamp
- committer info with timestamp
- commit message

8.4.1. Commit parents help us trace back

kind of like a linked list

we can use the hash of the parent in the output above

```
git cat-file -p 87c7
```

```
tree 06895b0f89062a5d9d12b5de5a068dc253f27092
parent d2d1fac72642204bfdcebc7703d786615b8de934
author Sarah M Brown <brownsarahm@uri.edu> 1727198616 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1727198616 -040

stop tracking
```

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, but some do.

8.4.2. Commit trees are the hash of the content

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.

```
git cat-file -p 0689
```

```
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 d3980efccf4856f0c61a6a16ed40be534c5230a5    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 hashis of the *contents* and the files all do have the same contents

8.4.3. Trees point to blobs of the file content

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

```
git cat-file -p 2d23
```

```
the info on how the code can be reused
```

++{“lesson_part”:“main”}

8.5. 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.

```
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.

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

Recall, we had seen the `HEAD` pointer before

```
cat .git/HEAD
```

```
ref: refs/heads/organization
```

which stores our current branch

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

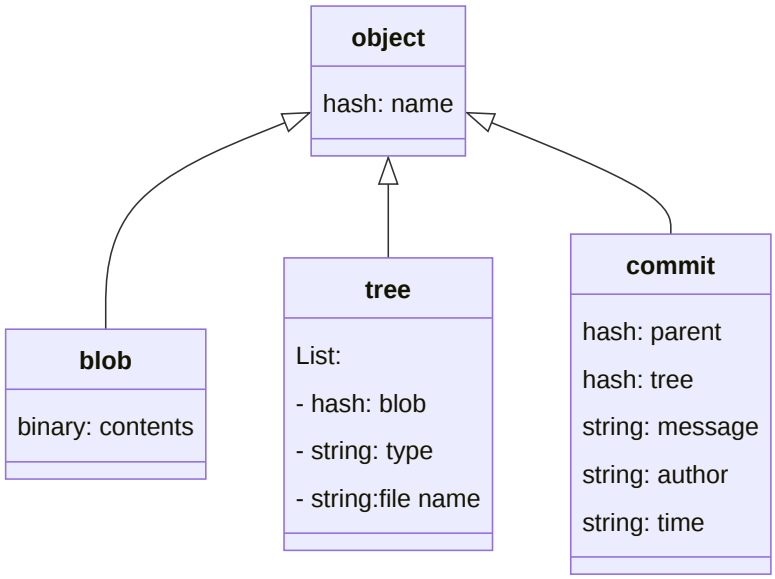
```
ls .git/objects/
```

```
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      70      91      a8      c5      e0
```

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

8.5.1. a commit is a type of git object

This is a *class diagram* for the `git object`s:




```
cat .git/objects/29/245e4b9cce937fb9e50bc3762ab19c6a7a12c3
```

```
x%?A
?0Fa?9?nt!?]? *(
??x?1??`Ld2???V?????eS/???P???1?aLL?EUT???!=????fu??~?
                                                                ??.???x?TItƤ???|)?>?'#?F hŸ?%?Cu?ٍ.??ђGb?????|Ez8`

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

```{code-cell} bash
:tags: ["skip-execution"]
git cat-file -t 2924
```
```

```
blob
```

```
git cat-file -p 2924
```

```
# Sarah Brown

tenure year: 2027
- i skied competiively in hs
<<<<<<< HEAD
- i started at uri in 2020
=======
- i went to Northeastern
>>>>>>> 62dcf61 (local second fun fact)
```

```
git log
```

```
commit 1e2ab9259651a73ad277e826d602514d28969c86 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:30:44 2024 -0400

    include readmen content

commit 87c72aeca9bd16700fc8fd8ee719136c13e83e01 (origin/organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:23:36 2024 -0400

    stop tracking

commit d2d1fac72642204bfdcebc7703d786615b8de934
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:16:10 2024 -0400

    organized files into foleders and ignore private

commit a3904a0a5e7adbcbf9fe439c387fb4dbd7846c51
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:46:19 2024 -0400

    Revert "start organizing"

    This reverts commit 9120d9d88aa587e4ffda1ee9aa8c3dcf8f764f7e.

commit 4ceb1500582236e98bdb141116821a5857f75a76
```

```
git status
```

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

nothing to commit, working tree clean
```

```
ls
```

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

8.6. Commit messages are essential

A git commit message must exist and is *always* for people, but can also be for machines.

the conventional commits standard is a format of commits

if you use this, then you can use automated tools to generate a full change log when you release code

Tooling and examples of conventional commits

8.7. Git tags point to commits and do not move

```
echo "example" >> README.md
```

```

```

```
git commit -a -m 'add more stuff'
```

```
[organization e8ab736] add more stuff
 1 file changed, 1 insertion(+)
```

```
git log
```

```
commit e8ab736a7c843e8515e484b136ffcbccfc162618 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Oct 1 13:33:22 2024 -0400

    add more stuff

commit 1e2ab9259651a73ad277e826d602514d28969c86
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:30:44 2024 -0400

    include readmen content

commit 87c72aeca9bd16700fc8fd8ee719136c13e83e01 (origin/organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:23:36 2024 -0400

    stop tracking

commit d2d1fac72642204bfdcebc7703d786615b8de934
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:16:10 2024 -0400

    organized files into foleders and ignore private

commit a3904a0a5e7adbcbf9fe439c387fb4dbd7846c51
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:46:19 2024 -0400
```

```
git checkout 1e2a
```

```
Note: switching to '1e2a'.

You are in 'detached HEAD' state. You can look around, make experimental
changes and commit them, and you can discard any commits you make in this
state without impacting any branches by switching back to a branch.

If you want to create a new branch to retain commits you create, you may
do so (now or later) by using -c with the switch command. Example:

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

Turn off this advice by setting config variable advice.detachedHead to false

HEAD is now at 1e2ab92 include readmen content
```

```
git tag 0.0.1
```

```

```

```
git tag
```

```
0.0.1
```

```
git checkout organization
```

```
Previous HEAD position was 1e2ab92 include readmen content
Switched to branch 'organization'
Your branch is ahead of 'origin/organization' by 2 commits.
  (use "git push" to publish your local commits)
```

```
git log
```

```
commit e8ab736a7c843e8515e484b136ffcbccfc162618 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Oct 1 13:33:22 2024 -0400

    add more stuff

commit 1e2ab9259651a73ad277e826d602514d28969c86 (tag: 0.0.1)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:30:44 2024 -0400

    include readmen content

commit 87c72aeca9bd16700fc8fd8ee719136c13e83e01 (origin/organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:23:36 2024 -0400

    stop tracking

commit d2d1fac72642204bfdcebc7703d786615b8de934
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:16:10 2024 -0400

    organized files into foleders and ignore private

commit a3904a0a5e7adbcbf9fe439c387fb4dbd7846c51
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 12:46:19 2024 -0400
```

```
ls .git
```

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

```
ls .git/refs/
```

```
heads   remotes tags
```

```
ls .git/refs/tags/
```

```
0.0.1
```

```
ls .git/refs/heads/
```

```
1-create-a-readme       my_branch
fun_fact                my_branch_checkedoutb
main                    organization
```

```
## Prepare for Next Class

```{include} ../_prepare/2024-10-03.md
```

8.8. 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?

8.9. Experience Report Evidence

8.10. Questions After Today's Class

8.10.1. Besides a linked list, what other data structures or algorithms does git use on its inner workings?

the main algorithm it uses is hashing.

More detail here is a good explore badge.

8.10.2. Why did running `ls .git/objects` had no files even when I have a commit history?

git compresses content into packfiles at times. This is rare, but can happen.

8.10.3. Is the only reason I would need to sign a commit is to show that I am the one dispersing it so it doesn't seem like spam or a virus? or should signing commits become a frequent practice?

the answer is yes to both questions

8.10.4. When I entered "git cat-file -p 39d5" in the command line, the result was "file with function with instructions for pip". What does this mean?

your blob object 39d5 is a fil with that contents like:
```

```
file with function with instructions for pip
```

we do not yet know its filenmae, but that is what this is.

8.10.5. What is the reason for tags? Are they means of labeling? Or means of access?

they are for labeling and for creating fixed “versions” that can be used for a “release”. For example, in calss I use tags to create releases and then those make notifcations, but for software releases are like the times we push to users.

9. How do programmers communicate about code?

> **Tip**
>
> check if your codespace has uncommitted changes on github.com/codespaces
>
> note:
>
> - you can only have 2 active at a time(green dots)
> - you can see if any have uncommitted changes
> - you can export those changes to a branch from this page

9.1. Why Documentation

Today we will talk about documentation, there are several reasons this is important:

- **using** official documentation is the best way to get better at the tools
- understanding how documentation is designed and built will help you use it better
- **writing** and **maintaining** documentation is really important part of working on a team
- documentation building tools are a type of developer tool (and these are generally good software design)

Design is best learned from examples. Some of the best examples of software *design* come from developer tools.

- source (js version)
- source (python version)

In particular documentation tools are really good examples of:

- pattern matching
- modularity and abstraction
- automation
- the build process beyond compiling

By the end of today’s class you will be able to:

- describe different types of documentation
- generate documentation as html

Plus we will reinforce things we have already seen:

- paths
- good file naming

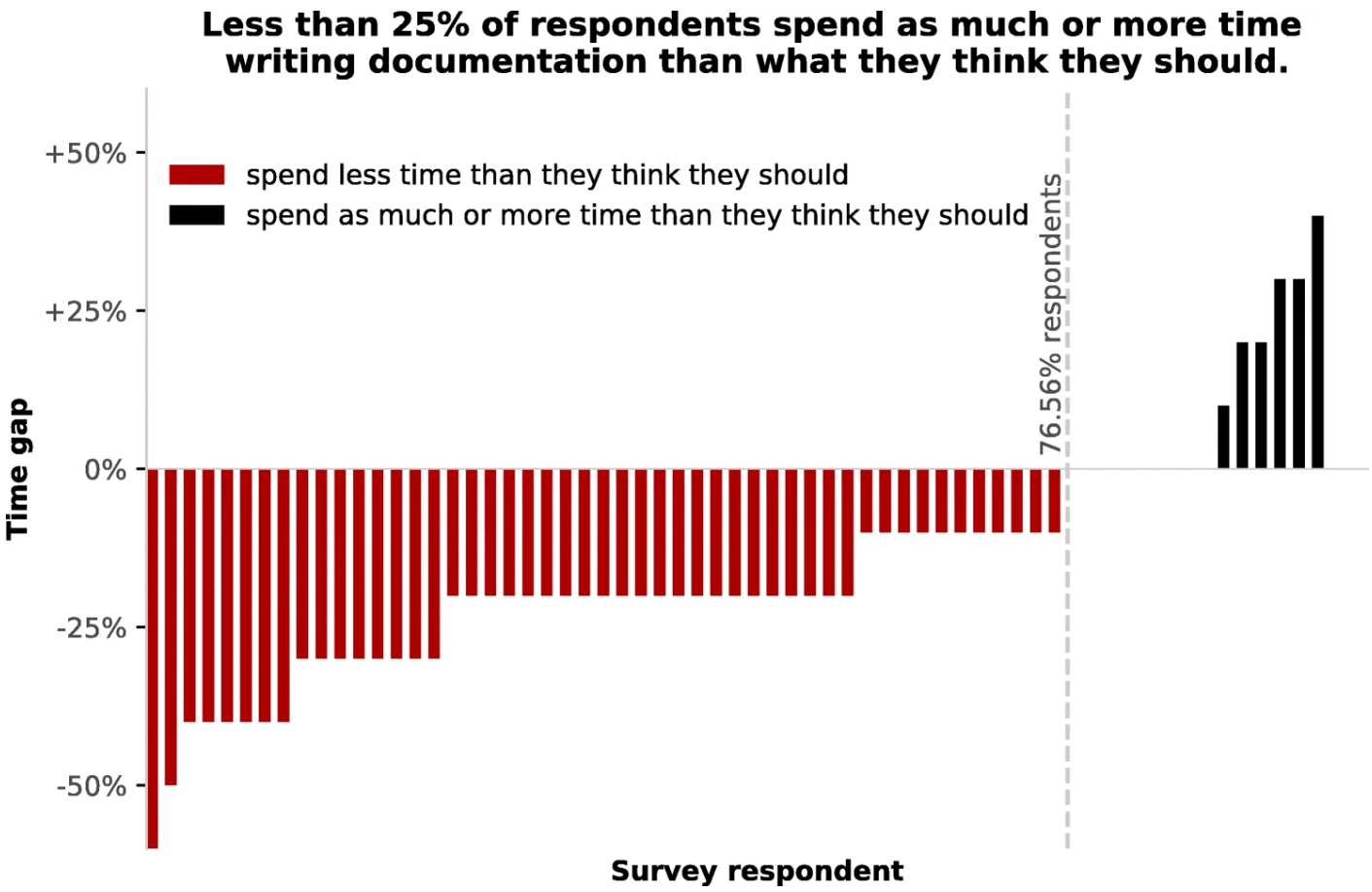
9.2. What is documentation

documentation types table

from ethnography of docuemtnation data science

9.2.1. Why is documentation so important?

we should probably spend more time on it



via source

9.3. So, *how* do we do it?

Different types of documentation live in different places and we use tools to maintain them.

As developers, we rely on code to do things that are easy for computers and hard for people.

Documenation Tools

write the docs

linux kernel uses sphinx and here is why and how it works

9.4. Jupyterbook

Jupyterbook wraps sphinx and uses markdown instead of restructured text. The project authors note in the documenation that it "can be thought of as an *opinionated distribution of Sphinx*". We're goign to use this.

navigate to your folder for this course (mine is `inclass/systems`)

We can confirm that `jupyter-book` is installed by checking the version.

```
jupyter-book --version
```

```
Jupyter Book      : 1.0.2
External ToC      : 1.0.1
MyST-Parser       : 2.0.0
MyST-NB           : 1.1.1
Sphinx Book Theme : 1.1.3
Jupyter-Cache     : 1.0.0
NbClient          : 0.10.0
```

Let's look at the status of this folder to set a baseline before we run new commands.

```
ls
```

```
fall24-brownsarahm      gh-inclass-brownsarahm
```

We will run a command to create a jupyterbook from a template, the command has 3 parts:

- `jupyter-book` is a program (the thing we installed)
- `create` is a subcommand (one action that program can do)
- `tiny-book` is an argument (a mandatory input to that action)

```
jupyter-book create tiny-book
```

```
================================================================================

Your book template can be found at

    tiny-book/

================================================================================
```

but it does an unrelated thing, we can use `ls` to check again to see more of what happened beyond the annoucnement above

```
ls
```

we have a new folder!

```
fall24-brownsarahm      tiny-book
gh-inclass-brownsarahm
```

You can make it with any name:

```
jupyter-book create example
```

beacuse the name is an argument or input

```
================================================================================

Your book template can be found at

    example/

================================================================================
```

Each one makes a directory, we can see by listing

```
ls
```

```
example                 gh-inclass-brownsarahm
fall24-brownsarahm      tiny-book
```

And we can delete the second one since we do not actually want it.

```
rm example/
```

```
rm: example/: is a directory
```

we get an error because it is not well defined to delete a directory, and potentially risky, so `rm` is written to throw an error

Instead, we have to tell it two additional things:

- to delete recusively `r`
- to force it to do something risky with `f`

note we can stack single character options together with a single `-`

```
rm -rf example/
```

```
ls
```

```
fall24-brownsarahm      tiny-book
gh-inclass-brownsarahm
```

as we want

9.5. Structure of a Jupyter book

```
cd tiny-book/
```

We will explore the output by looking at the files

```
ls
```

```
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
markdown-notebooks.md
```

A jupyter book has two required files (`_config.yml` and `_toc.yml`), some for content, and some helpers that are common but not required.

- config defaults
- toc file formatting rules
- the `*.md` files are content
- the `.bib` file is bibiolography information
- The other files are optional, but common. Requirements.txt is the format for pip to install python depndencies. There are different standards in other languages for how

Note

the extention (`.yml`) is yaml, which stands for “YAML Ain’t Markup Language”. It consists of key, value pairs and is deigned to be a human-friendly way to encode data for use in any programming language.

9.5.1. Dev tools mean we do not have to write bibliographies manually

bibliographies are generated with bibtex which takes structured information from the references in a bibtex file with help from sphinxcontrib-bibtex

For general reference, reference managers like zotero and mendeley can track all of your sources and output the references in bibtex format that you can use anywhere or sync with tools like MS Word or Google Docs.

this is not a git repo it is only a folder we have made locally

```
git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```

```
cat references.bib
```

```
@inproceedings{holdgraf_evidence_2014,
        address = {Brisbane, Australia, Australia},
        title = {Evidence for {Predictive} {Coding} in {Human} {Auditory} {Cortex}},
        booktitle = {International {Conference} on {Cognitive} {Neuroscience}},
        publisher = {Frontiers in Neuroscience},
        author = {Holdgraf, Christopher Ramsay and de Heer, Wendy and Pasley, Brian N. and Knight, Robert
        year = {2014}
}

@article{holdgraf_rapid_2016,
        title = {Rapid tuning shifts in human auditory cortex enhance speech intelligibility},
        volume = {7},
        issn = {2041-1723},
        url = {http://www.nature.com/doifinder/10.1038/ncomms13654},
        doi = {10.1038/ncomms13654},
        number = {May},
        journal = {Nature Communications},
        author = {Holdgraf, Christopher Ramsay and de Heer, Wendy and Pasley, Brian N. and Rieger, Jochem
        year = {2016},
        pages = {13654},
        file = {Holdgraf et al. - 2016 - Rapid tuning shifts in human auditory cortex enhance speech inte
}

@inproceedings{holdgraf_portable_2017,
        title = {Portable learning environments for hands-on computational instruction using container-ar
        volume = {Part F1287},
        isbn = {978-1-4503-5272-7},
        doi = {10.1145/3093338.3093370},
        abstract = {© 2017 ACM. There is an increasing interest in learning outside of the traditional cl
        booktitle = {{ACM} {International} {Conference} {Proceeding} {Series}},
        author = {Holdgraf, Christopher Ramsay and Culich, A. and Rokem, A. and Deniz, F. and Alegro, M.
        year = {2017},
        keywords = {Teaching, Bootcamps, Cloud computing, Data science, Docker, Pedagogy}
}

@article{holdgraf_encoding_2017,
        title = {Encoding and decoding models in cognitive electrophysiology},
        volume = {11},
        issn = {16625137},
        doi = {10.3389/fnsys.2017.00061},
        abstract = {© 2017 Holdgraf, Rieger, Micheli, Martin, Knight and Theunissen. Cognitive neuroscier
        journal = {Frontiers in Systems Neuroscience},
        author = {Holdgraf, Christopher Ramsay and Rieger, J.W. and Micheli, C. and Martin, S. and Knight
        year = {2017},
        keywords = {Decoding models, Encoding models, Electrocorticography (ECoG), Electrophysiology/evok
}

@book{ruby,
  title     = {The Ruby Programming Language},
  author    = {Flanagan, David and Matsumoto, Yukihiro},
  year      = {2008},
  publisher = {O'Reilly Media}
}
```

9.6. YAML files let us set parameters in a file

The table of contents file describe how to put the other files in order.

```
cat _toc.yml
```

```
# Table of contents
# Learn more at https://jupyterbook.org/customize/toc.html
```

The first two lines are comments, the pound sign `#` is a comment in bash and YAML. Here, the developers chose, in the template to put information about how to set up this file right in the template file. This is developers helping their users!

```
format: jb-book
root: intro
```

The first next two lines are key value pairs that tell the high level settings

```
chapters:
- file: markdown
- file: notebooks
- file: markdown-notebooks
```

the end of the file shows a list of files that will be treated as chapters. This is also the syntax for a list in YAML, the list is named `chapters` and each item, starting with a `-` has a single key, `file`

Where have we seen a YAML list?

You can create a community badge that uses as the location of the “contribution” a link to the snipped to a set of lines in your main fall24 repo that is a YAML list. This is valid as long as it is created before lab time on Monday October 7.

Config tells `jupyter-book` how to run, these are literally the options and settings it needs to make a site (or other document).

```
cat _config.yml
```

```
# Book settings
# Learn more at https://jupyterbook.org/customize/config.html

title: My sample book
author: The Jupyter Book Community
logo: logo.png

# Force re-execution of notebooks on each build.
# See https://jupyterbook.org/content/execute.html
execute:
  execute_notebooks: force

# Define the name of the latex output file for PDF builds
latex:
  latex_documents:
    targetname: book.tex

# Add a bibtex file so that we can create citations
bibtex_bibfiles:
  - references.bib

# Information about where the book exists on the web
repository:
  url: https://github.com/executablebooks/jupyter-book  # Online location of your book
  path_to_book: docs  # Optional path to your book, relative to the repository root
  branch: master  # Which branch of the repository should be used when creating links (optional)

# Add GitHub buttons to your book
# See https://jupyterbook.org/customize/config.html#add-a-link-to-your-repository
html:
  use_issues_button: true
  use_repository_button: true
```

Here the develoeprs left many links to more information and included default values for several settings that people might want to change easily.

9.6.1. Retiring racist language

Historically the default branch in git was called master. `jupyter-book` has that as default

- derived from a master/slave analogy which is not even how git works, but was adopted terminology from other projects
- GitHub no longer does
- the broader community is changing as well
- git allows you to make your default not be master
- literally the person who chose the names “master” and “origin” regrets that choice the name main is a more accurate and not harmful term and the current convention.

9.7. Files can help us install dependencies too

The one last file tells us what dependencies we have

```
cat requirements.txt
```

If your book generates with error messages run `pip install -r requirements.txt`

```
jupyter-book
matplotlib
numpy
```

Thse are two python libraries that are required to run the code in the notebook files, plus `jupyter-book` itself.

```
ls
```

```
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
markdown-notebooks.md
```

9.8. Building Documentation

We can transform from raw source to an output by **building** the book

```
jupyter-book build .
```

```
Running Jupyter-Book v1.0.2
Source Folder: /Users/brownsarahm/Documents/inclass/systems/tiny-book
Config Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_config.yml
Output Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html
Running Sphinx v7.4.7
loading translations [en]... done
making output directory... done
[etoc] Changing master_doc to 'intro'
checking bibtex cache... out of date
parsing bibtex file /Users/brownsarahm/Documents/inclass/systems/tiny-book/references.bib... parsed 5 ent
myst v2.0.0: MdParserConfig(commonmark_only=False, gfm_only=False, enable_extensions={'linkify', 'dollarm
myst-nb v1.1.1: NbParserConfig(custom_formats={}, metadata_key='mystnb', cell_metadata_key='mystnb', kerr
Using jupyter-cache at: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/.jupyter_cache
sphinx-multitoc-numbering v0.1.3: Loaded
building [mo]: targets for 0 po files that are out of date
writing output...
building [html]: targets for 4 source files that are out of date
updating environment: [new config] 4 added, 0 changed, 0 removed
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executing notebook using lo
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executed notebook in 4.13 s
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executing notebook using local CW
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executed notebook in 2.45 seconds

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
copying assets...
copying static files... done
copying extra files... done
copying assets: done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
copying images... [100%] _build/jupyter_execute/a31e63b1f6ca34376ef17d2b6c277648c6b47bb0a75c5165999735167
dumping search index in English (code: en)... done
dumping object inventory... done
[etoc] missing index.html written as redirect to 'intro.html'
build succeeded.

The HTML pages are in _build/html.

================================================================================

Finished generating HTML for book.
Your book's HTML pages are here:
    _build/html/
You can look at your book by opening this file in a browser:
    _build/html/index.html
Or paste this line directly into your browser bar:
    file:///Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html/index.html

================================================================================
```

9.9. Documentation engines use patterns

These are a structure that a lot of programming languages and other contexts

See examples in other langauges in their documentation:

- rust
- `fmt` package in go

- java
- the liquid templating language is used for shopify and used in a few other technologies

I can do a quick python example by starting a python interpretter.

```
'ldksjfsdfj {my_var} sflksjflsdk'.format(my_var ='hello')
```

```
'ldksjfsdfj hello sflksjflsdk'
```

We can see it added a folder

```
ls
```

```
_build                  markdown-notebooks.md
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
```

the `_build` fodler is new

and we can look at the top of the contents (you can inspect fully in your repo, but this output would be long with `cat` and the point here is *not* to know the details of HTML, but to trust that it can be added formulaically)

```
head _build/html/index.html
```

```
<meta http-equiv="Refresh" content="0; url=intro.html" />
```

we can see how many files it made

```
ls _build/html/
```

```
_images                 markdown-notebooks.html
_sources                markdown.html
_sphinx_design_static   notebooks.html
_static                 objects.inv
genindex.html           search.html
index.html              searchindex.js
intro.html
```

there is

- one html file for each source file from the toc
- a fodler for images
- the source code
- some files that help the search

- an index page
- and a `_static` folder for the css to style the page

9.10. Prepare for Next Class

1. review the notes on what is a commit. In gitdef.md on the branch for this issue, try to describe git in the four ways we described a commit. **the point here is to think about what you know for git and practice remembering it, not "get the right answer"; this is prepare work, we only check that it is complete, not correct**
2. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file below. We will come back to these notes in class later, but it is best to record over a time period instead of trying to remember at that time. Store your notes in your fall24 repo in idethoughts.md on a dedicated `ide_prep` branch. **This is prep for after a few weeks from now, not for October 8; keep this branch open until it is specifically asked for**

9.11. Badges

Review | **Practice**

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book, by manually adding those files. **do not add the whole template to your repo**, make the content you have already so it can *build* into html. Set it so that the `_build` directory is not under version control.
2. Add `docs-review.md` to your KWL repo and explain the most important things to know about documentation in your own words using other programming concepts you have learned so far. Include in a markdown (same as HTML `<!-- comment -->`) comment the list of CSC courses you have taken for context while we give you feedback.

9.12. Experience Report Evidence

9.13. Questions After Today's Class

Important

Windows users should see the Windows Help & Notes page for help

9.13.1. What are some similar tools to jupyter-book?

Note

We are using `jupyter-book` which is different from `jupyter notebook`

Jupyter-book is similar to `sphinx` because it uses it. There are a lot of other Documenation Tools.

mystmd is a newer project that is a lot like jupyter (made by the same people) designed for publishing like with curvenote or vercel

Another similar tool is quarto

and in some ways pandoc is also similar

> **Important**
>
> using any of these tools is a valid build idea

9.13.2. Is it technically a Platform as a service?

No, it is just a program tht transforms our content from one to another.

9.13.3. does jupyter-book have to be referenced anywhere when used to create something?

No you do not have to say that you used it to create your content, but you can. For example, in commit 947fdff I add that to the footer of this site.

10. What *is* git?

Last week, we learned about what a commit is and then we took a break from how git works, to talk more about how developers communicate about code

Today we will learn what git is more formally.

Tip

We will go in and out of topics at times, in order to provides what is called *spaced repetition*, repeating material or key concepts with breaks in between.

Using git correctly is a really important goal of this course because git is an opportunity for you to demonstrate a wide range of both practical and conceptual understanding.

So, I have elected to interleave other topics with git to give core git ideas some time to simmer and give you time to practice them before we build on them with more depth at git.

Also, we are both learning git and *using* git as a motivating example of other key important topics. ``

Why so much git?

Today, we are going to learn *what* git is and later we will learn more details of how it is implemented.

Remember we are spending so much time with git for two reasons:

1. it is an important developer tool
2. it demonstrates important conceptual ideas that occur in other areas of CS

git book is the official reference on git.

this includes other spoken languages as well if that is helpful for you.

git definition

From here, we have the full definition of git

git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it.

We do not start from that point, because these documents were written for target audience of working developers who are familar with other, old version control systems and learning an *additional* one.

Most of you, however, have probably not used another version control system.

Git is a File system

Content-addressable filesystem means a key-value data store.

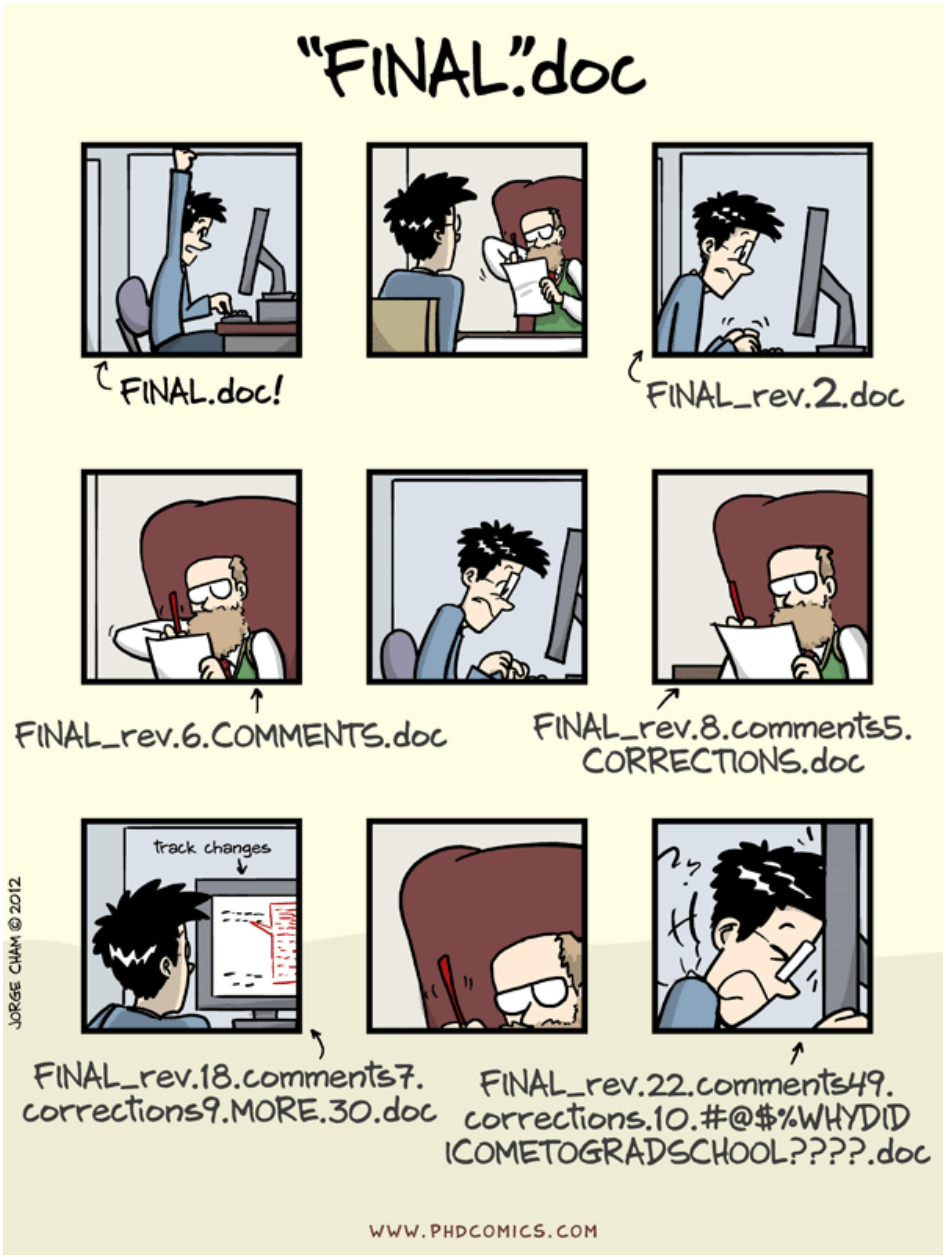
some examples of key-value pairs that you have seen in computer science broadly, and in this course specficially

- python dictionaries
- pointers (address,content)
- parameter, passed values
- yaml files

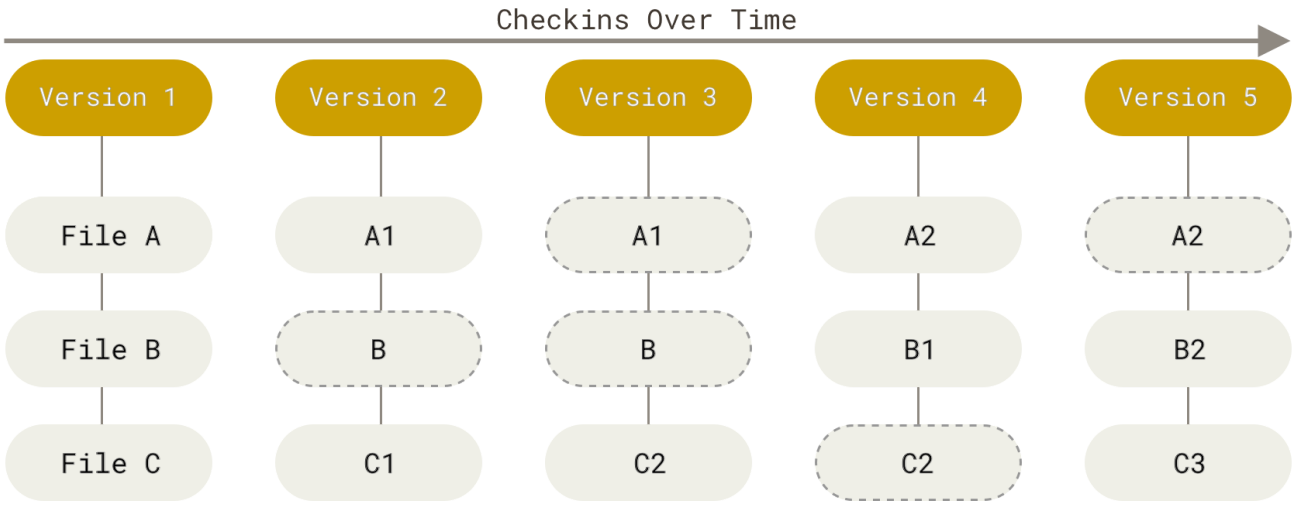
What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

Git is a Version Control System

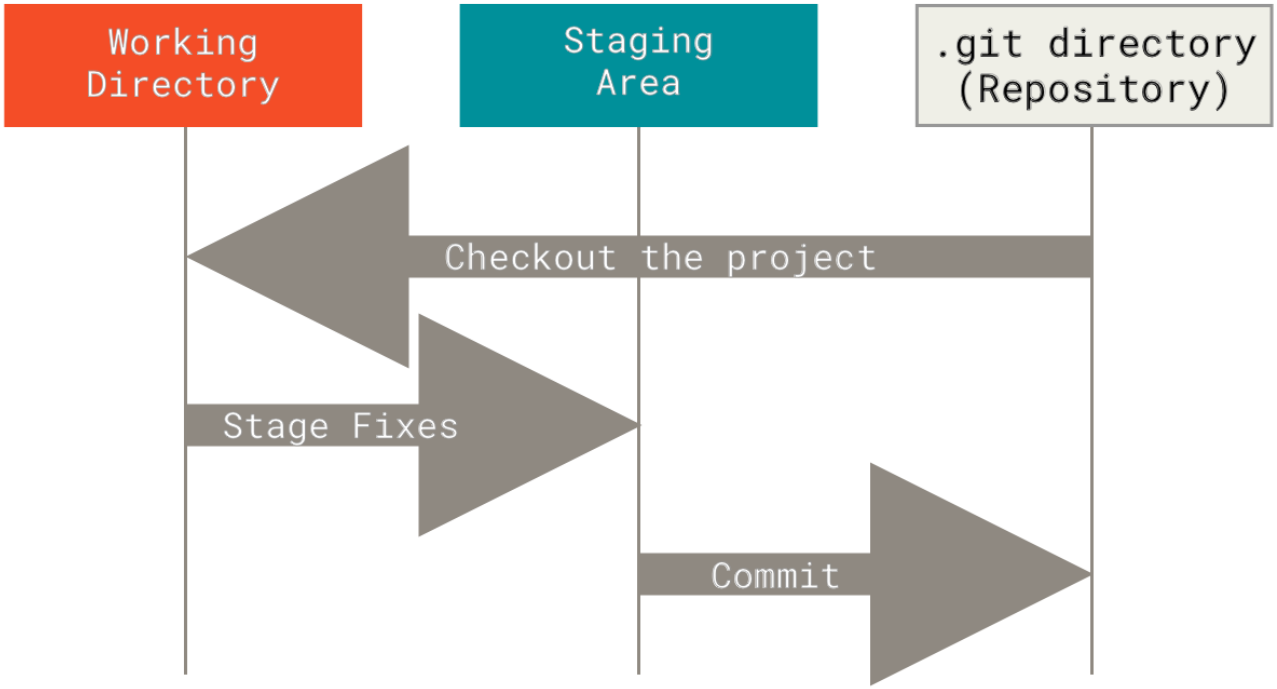
In the before times



git stores **snapshots** of your work each time you commit.



it uses 3 stages:



These three stages are the in relation to your working directory, and potentially remotes.

So in broader context, the git visual cheatsheet is a more complete picture and has commands overlayed with the concept.

Git has two sets of commands

- Porcelain: the user friendly VCS
- Plumbing: the internal workings- a toolkit for a VCS

We have so far used git as a version control system. A version control system, in general, will have operations like commit, push, pull, clone. These may work differently under the hood or be called different things, but those are what something needs to have in order to keep track of different versions.

The plumbing commands reveal the *way* that git performs version control operations. This means, they implement the git file system operations for the git version control system.

You can think of the plumbing vs porcelain commands like public/private methods. As a user, you only need the public methods (porcelain commands) but those use the private ones to get things done (plumbing commands). We will use the plumbing commands over the next few classes to examine what git *really* does when we call the porcelain commands that we will typically use.

Git is distributed

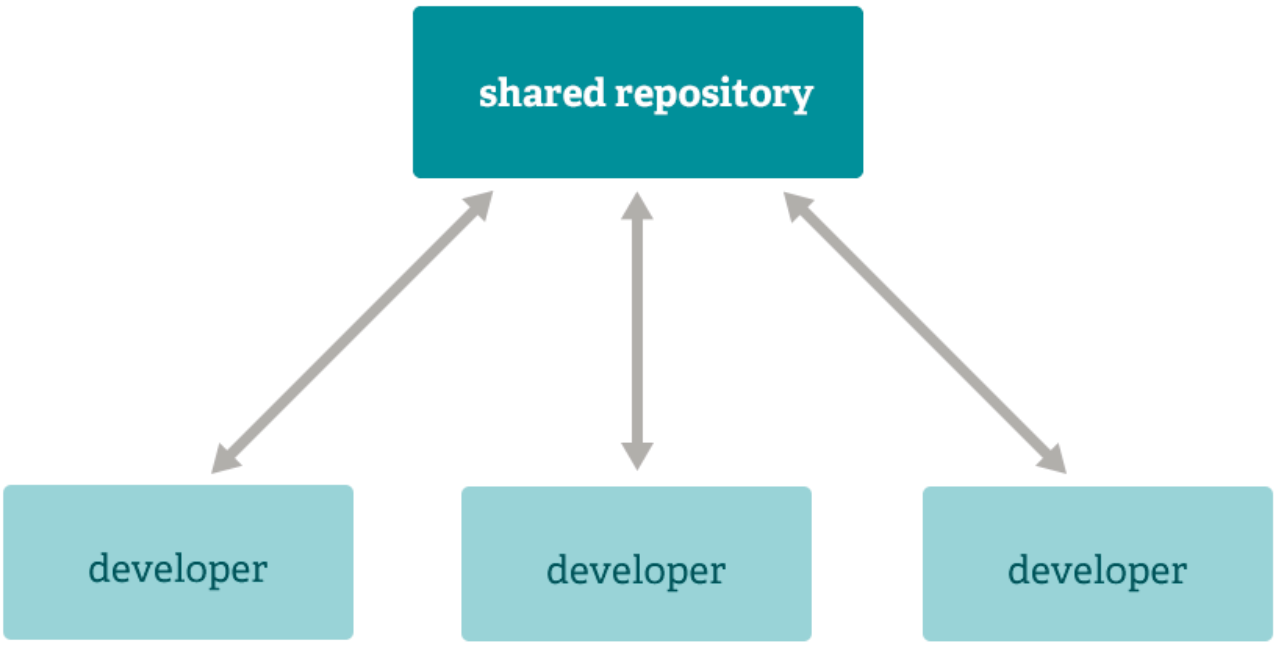
What does that mean?

Git runs locally. It can run in many places, and has commands to help sync across remotes, but git does not require one copy of the repository to be the “official” copy and the others to be subordinate. git just sees repositories.

For human reasons, we like to have one “official” copy and treat the others as local copies, but that is a social choice, not a technological requirement of git. Even though we will typically use it with an offical copy and other copies, having a tool tht does not care, makes the tool more flexible and allows us to create workflows, or networks of copies that have any relationship we want.

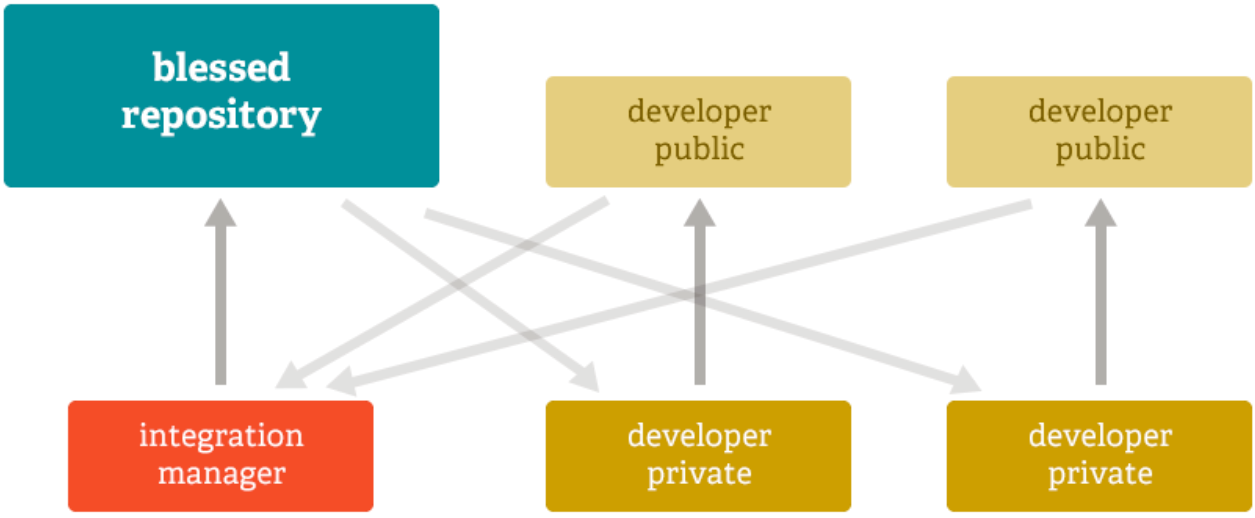
It’s about the workflows, or the ways we socially *use* the tool.

Subversion Workflow

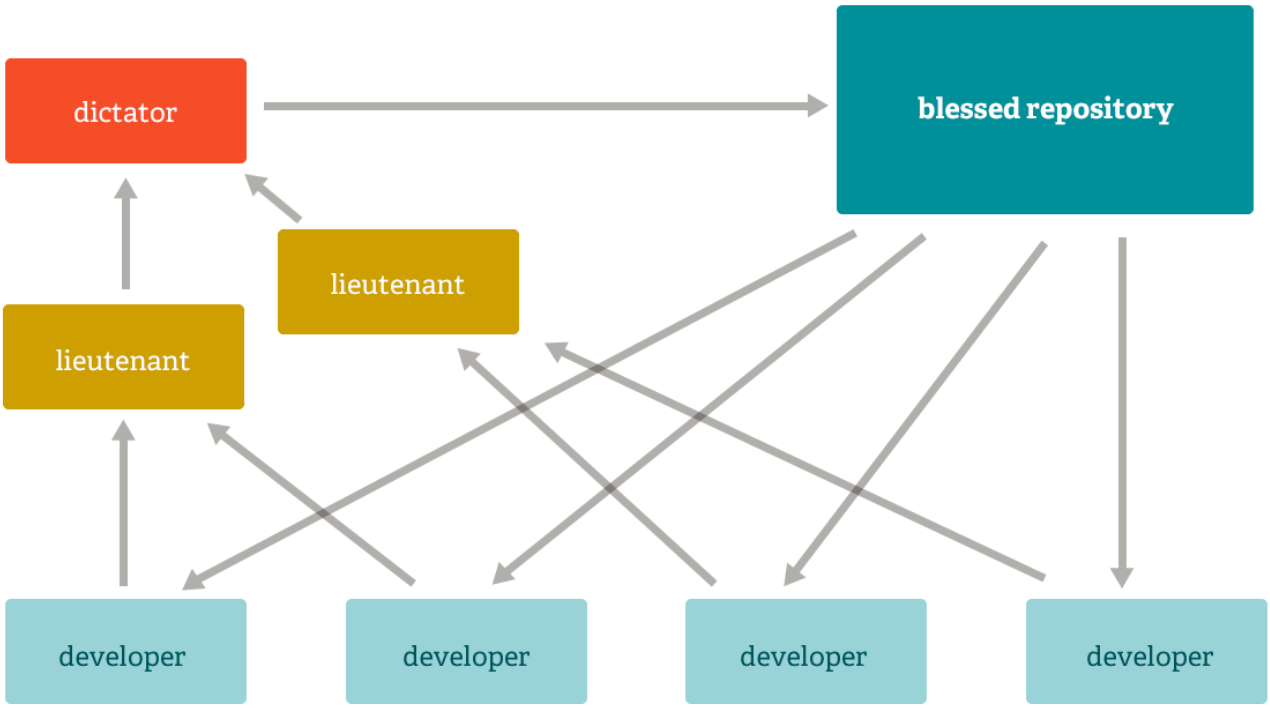


subversion is an older VCS

Integration Manager



dictator and lieutenants



Starting a git repo locally

We made this folder, but we have not used any git operations on it yet, it is actually not a git repo, which we *could* tell from the output above, but let's use git to inspect and get another hint.

Now we will return to the `tiny-book` folder we made last class. This is a jupyterbook, meaning the command `jupyter-book build .` works in the directory.

```
ls
```

```
_build                  markdown-notebooks.md
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
```

we see all fo the files

```
ls -a
```

```
.                       logo.png
..                      markdown-notebooks.md
_build                  markdown.md
_config.yml             notebooks.ipynb
_toc.yml                references.bib
intro.md                requirements.txt
```

but no .git!

and if we try a git command

```
git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```

This tells us the `.git` directory is missing form the current path and all parent directories.

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/tiny-book
```

nor is any other fold in this list, since it checks the parents.

To make it a git repo, we use `git init` with the path we want to initialize, which currently is `.`

```
git init .
```

```
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/brownsarahm/Documents/inclass/systems/tiny-book/.git/
```

we'll change our default branch to main

```
git branch -m main
```

and check in with git now

```
git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committ
        _build/
        _config.yml
        _toc.yml
        intro.md
        logo.png
        markdown-notebooks.md
        markdown.md
        notebooks.ipynb
        references.bib
        requirements.txt

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

this time it works and we see a two important things:

- there are no previous commits
- all of the files are untracked

Retiring racist language

Historically the default branch was called master.

- derived from a master/slave analogy which is not even how git works, but was adopted terminology from other projects
- GitHub no longer does
- the broader community is changing as well
- git allows you to make your default not be master
- literally the person who chose the names "master" and "origin" regrets that choice the name main is a more accurate and not harmful term and the current convention.

10.1. Handling Built files

The built site files are compeltey redundant, content wise, to the original markdown files.

the build folder is compeltely redundant, we can delete it

```
rm -rf _build/
```

confirm it is gone

```
ls
```

```
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
markdown-notebooks.md
```

then re-build to replace it

```
jupyter-book build .
```

```
Running Jupyter-Book v1.0.2
Source Folder: /Users/brownsarahm/Documents/inclass/systems/tiny-book
Config Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_config.yml
Output Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html
Running Sphinx v7.4.7
loading translations [en]... done
making output directory... done
[etoc] Changing master_doc to 'intro'
checking bibtex cache... out of date
parsing bibtex file /Users/brownsarahm/Documents/inclass/systems/tiny-book/references.bib... parsed 5 ent
myst v2.0.0: MdParserConfig(commonmark_only=False, gfm_only=False, enable_extensions={'substitution', 'ta
myst-nb v1.1.1: NbParserConfig(custom_formats={}, metadata_key='mystnb', cell_metadata_key='mystnb', kerr
Using jupyter-cache at: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/.jupyter_cache
sphinx-multitoc-numbering v0.1.3: Loaded
building [mo]: targets for 0 po files that are out of date
writing output...
building [html]: targets for 4 source files that are out of date
updating environment: [new config] 4 added, 0 changed, 0 removed
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executing notebook using lo
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executed notebook in 1.28 s
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executing notebook using local CW
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executed notebook in 2.10 seconds

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
copying assets...
copying static files... done
copying extra files... done
copying assets: done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
copying images... [100%] _build/jupyter_execute/a31e63b1f6ca34376ef17d2b6c277648c6b47bb0a75c5165999735167
dumping search index in English (code: en)... done
dumping object inventory... done
[etoc] missing index.html written as redirect to 'intro.html'
build succeeded.

The HTML pages are in _build/html.

================================================================================

Finished generating HTML for book.
Your book's HTML pages are here:
    _build/html/
You can look at your book by opening this file in a browser:
    _build/html/index.html
Or paste this line directly into your browser bar:
    file:///Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html/index.html

================================================================================
```

and see it is back, or view the file in our browser

```
ls
```

```
_build                  markdown-notebooks.md
_config.yml             markdown.md
_toc.yml                notebooks.ipynb
intro.md                references.bib
logo.png                requirements.txt
```

it contains *derived* content, not content that a person manually authored

```
ls _build/
```

```
html            jupyter_execute
```

```
ls _build/html/
```

```
_images                 markdown-notebooks.html
_sources                markdown.html
_sphinx_design_static   notebooks.html
_static                 objects.inv
genindex.html           search.html
index.html              searchindex.js
intro.html
```

since they are fully redundant,

We do not want to keep track of changes for the built files since they are generated from the source files. It's redundant and makes it less clear where someone should update content.

Git helps us with this with the .gitignore

```
echo "_build/" > .gitignore
```

now it does not show!

```
git status
```

```
On branch main

No commits yet

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore
        _config.yml
        _toc.yml
        intro.md
        logo.png
        markdown-notebooks.md
        markdown.md
        notebooks.ipynb
        references.bib
        requirements.txt

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

Now, we can add and commit our book!

```
git add .
```

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   .gitignore
        new file:   _config.yml
        new file:   _toc.yml
        new file:   intro.md
        new file:   logo.png
        new file:   markdown-notebooks.md
        new file:   markdown.md
        new file:   notebooks.ipynb
        new file:   references.bib
        new file:   requirements.txt
```

```
git commit -m 'tempatle'
```

```
[main (root-commit) eb9ef8e] tempatle
 10 files changed, 342 insertions(+)
 create mode 100644 .gitignore
 create mode 100644 _config.yml
 create mode 100644 _toc.yml
 create mode 100644 intro.md
 create mode 100644 logo.png
 create mode 100644 markdown-notebooks.md
 create mode 100644 markdown.md
 create mode 100644 notebooks.ipynb
 create mode 100644 references.bib
 create mode 100644 requirements.txt
```

10.2. How do I push a repo that I made locally to GitHub?

Right now, we do not have any remotes, so if we try to push it will fail. Next we will see how to fix that.

First let's confirm

```
git push
```

```
fatal: No configured push destination.
Either specify the URL from the command-line or configure a remote repository using

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

it fails, but it tells us how to fix it. This is why inspection is so powerful in developer tools, that is where we developers give one another hints.

Right now, we do not have any remotes

For today, we will create an empty github repo shared with me, by accepting the assignment linked in prismia or ask a TA/instructor if you are making up class.

More generally, you can create a repo

That default page for an empty repo if you do not initiate it with any files will give you the instructions for what remote to add.

Now we add the remote

```
git remote add origin https://github.com/compsys-progtools/tiny-book-brownsarahm-1.git
```

Then we can push

```
git push
```

```
fatal: The current branch main has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin main

To have this happen automatically for branches without a tracking
upstream, see 'push.autoSetupRemote' in 'git help config'.
```

once we link the branch

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

```
To https://github.com/compsys-progtools/tiny-book-brownsarahm-1.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/compsys-progtools/tiny-book-brownsarahm-1.git'
hint: Updates were rejected because the remote contains work that you do not
hint: have locally. This is usually caused by another repository pushing to
hint: the same ref. If you want to integrate the remote changes, use
hint: 'git pull' before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

except github classroom added a commit, so we have diverged histories

we can fix this by pulling

```
git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
Unpacking objects: 100% (5/5), 1.70 KiB | 348.00 KiB/s, done.
From https://github.com/compsys-progtools/tiny-book-brownsarahm-1
 * [new branch]      feedback   -> origin/feedback
 * [new branch]      main       -> origin/main
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

    git branch --set-upstream-to=origin/<branch> main
```

which requires the branch to be better linked

```
git branch --set-upstream-to=origin/main main
```

```
branch 'main' set up to track 'origin/main'.
```

```
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.
```

and we need the option for how to work. I want our local commits to be the most recent so I will `rebase`

```
git pull --rebase
```

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

We can see that in the log it is as we expected.

```
git log
```

```
commit 60bd457a4c3523eae25be36ac685207d605c9676 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Oct 8 12:51:52 2024 -0400

    tempatle

commit d781535217b324d9cb2ce6cb45ae565b54ee786f (origin/main)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Oct 8 16:54:12 2024 +0000

    Setting up GitHub Classroom Feedback

commit 72bcbb8cbd2769d21aad3c23c8fbe477d0260ced (origin/feedback)
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Oct 8 16:54:12 2024 +0000

    GitHub Classroom Feedback
```

Note the commits made more recently *in time* appear older in the commit history.

Now we can push!

```
git push
```

```
Enumerating objects: 13, done.
Counting objects: 100% (13/13), done.
Delta compression using up to 8 threads
Compressing objects: 100% (10/10), done.
Writing objects: 100% (12/12), 16.33 KiB | 8.17 MiB/s, done.
Total 12 (delta 0), reused 0 (delta 0), pack-reused 0 (from 0)
To https://github.com/compsys-progtools/tiny-book-brownsarahm-1.git
   d781535..60bd457  main -> main
```

10.3. How does git do all these things?

We can use the bash command `find` to search the file *system* note that this does not search the contents of the files, just the names.

```
find .git/objects/ -type f
```

```
.git/objects//69/fa449dd96e2405945b2e4cff2fd0ab8b102097
.git/objects//0e/2e3b27f61b5908c4bb75a1ca680ee4053aa992
.git/objects//60/bd457a4c3523eae25be36ac685207d605c9676
.git/objects//5f/534f8051f6a94d40e57e58242ef0113fae4fd1
.git/objects//5f/2d0ff4056de4c043c720fd80db5ff75502d286
.git/objects//d7/81535217b324d9cb2ce6cb45ae565b54ee786f
.git/objects//eb/9ef8efc08fa2067507e7af41fcc904a9a11cb5
.git/objects//fd/b7176c429a73d5335e127b27d530b8aaa07c7d
.git/objects//29/a422c19251aeaeb907175e9b3219a9bed6c616
.git/objects//74/d5c7101ed8c8c1a6f87e31debd9445df1f0e71
.git/objects//7e/821e45db31376729c73f3616fb24db2b655a95
.git/objects//72/bcbb8cbd2769d21aad3c23c8fbe477d0260ced
.git/objects//06/d56f40c838b64eb048a63e036125964a069a3a
.git/objects//a0/57a320dcd595f3f0e0d250c3af4a5653596914
.git/objects//e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects//fa/eea606145667f54d220a0c17ffe8d22db07146
.git/objects//c2/48258b7f940fe3a133a2f317b574821da8bf7b
.git/objects//f8/cdc73cb2be06824f521837366ec95b73d55ef8
.git/objects//78/3ec6aa5afe2f0a66087d01a112f543e1ed287e
```

This is a lot of files! It's more than we have in our working directory.

This is a consequence of git taking snap shots and tracking both the actual contents of our working directory **and** our commit messages and other meta data about each commit.

10.4. Git Variables

the program `git` does not run continously the entire time you are using it for a project. It runs quick commands each time you tell it to, it's goal is to manage files, so this makes sense. This also means that important information that `git` needs is also saved in files.

We can see the files that it has by listing the directory:

```
ls .git
```

```
AUTO_MERGE      ORIG_HEAD       index           refs
COMMIT_EDITMSG  config          info
FETCH_HEAD      description     logs
HEAD            hooks           objects
```

the files in all caps are like gits variables.

Lets look at the one called `HEAD` we have interacted with `HEAD` before when resolving merge conflicts.

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

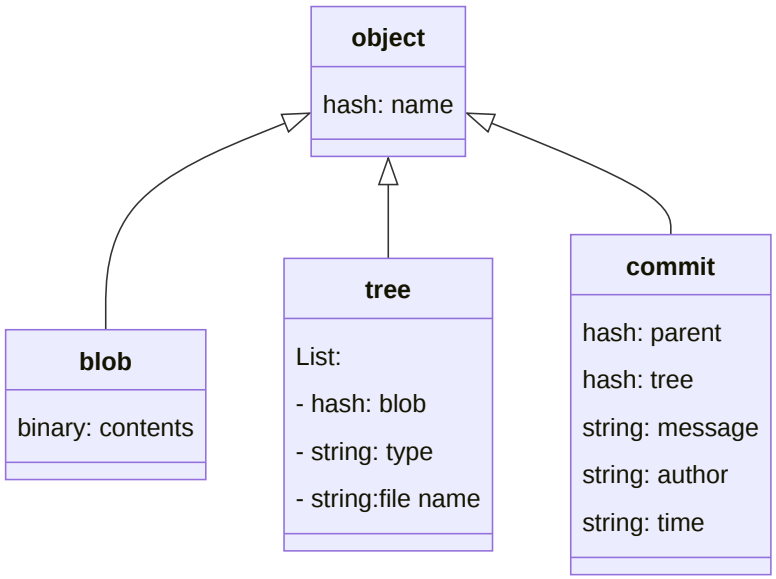
`HEAD` is a pointer to the currently checked out branch.

The other files with `HEAD` in their name are similarly pointers to other references, named corresponding to other things.

10.5. Git Objects

There are 3 types:

- blob objects: the content of your files (data)
- tree objects: stores file names and groups files together (organization)
- Commit Objects: stores information about the sha values of the snapshots



10.6. Prepare for Next Class

1. Take a few minutes to think what you know about hashing and numbers. Create hash_num_prep.md with two sections: `## Hashing` with a few bullet points summarzing key points about hashing, and `## Numbers` with what types of number representations you know.
2. start commenting/expressing interest on build/explore ideas. Next lab will be focused on that.

- build template
- explore instructions

10.7. Badges

Review | **Practice**

1. Read about different workflows in git and describe which one you prefer to work with and why in favorite_git_workflow.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. In commit_contents.md, redirect the content of your most recent commit, its tree, and the contents of one blob. Edit the file or use `echo` to put markdown headings between the different objects. Add a title `# Complete Commit` to the file

> and at the bottom of the file add `## Reflection` subheading with some notes on how, if at all this excercise helps you understand what a commit is.

10.8. Experience Report Evidence

10.9. Questions After Today's Class

10.9.1. Can you clarify the difference between working in local repos, github repos, and github codespaces and how they all integrate?

- a local repo is a folder on your computer
- a github repo is the copy on the github.com server that the website interface interacts with
- a github codespace is a virtual machine on an Azure server that you can use VScode with
- a codespace is a *local* copy as far as git is concerned, since you have local access
- both types of local repos can be pushed and pulled to github.com

10.9.2. how were the older verion controls and what similar features did it have to git that remains present in git present day?

this is a good explore badge topic.

10.9.3. why is there a difference between Porcealin and Plumbing commands when they both interact with the file system and are dependent on its contents?

> is it the operatiions that are performed by the commands? Or is it more to do with the level that they interact with the file system.

the porcelain commands essentially call the plumbing comamnds. In normal day to day use you only need the porcelain commands, plumbing are the low level details.

10.9.4. Can the list of commit (objects) ever become long enough to cause issues?

Technically, yes, practically no. the badges for today will have you investigate this idea.

11. What is a commit number?

```
cd gh-inclass-brownsarahm/
```

```
git  log
```

```
commit e8ab736a7c843e8515e484b136ffcbccfc162618 (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Oct 1 13:33:22 2024 -0400

    add more stuff

commit 1e2ab9259651a73ad277e826d602514d28969c86 (tag: 0.0.1)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:30:44 2024 -0400

    include readmen content

commit 87c72aeca9bd16700fc8fd8ee719136c13e83e01 (origin/organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 24 13:23:36 2024 -0400

    stop tracking
```

11.1. What is a hash?

a hash is:

- a fixed size value that can be used to represent data of arbitrary sizes
- the *output* of a hashing function
- often fixed to a hash table

Common examples of hashing are lookup tables and encryption with a cyrptographic hash.

A hashing function could be really simple, to read off a hash table, or it can be more complex.

For example:

Hash	content
0	Success
1	Failure

If we want to represent the status of a program running it has two possible outcomes: success or failure. We can use the following hash table and a function that takes in the content and returns the corresponding hash. Then we could pass around the 0 and 1 as a single bit of information that corresponds to the outcomes.

This lookup table hash works here.

In a more complex scenario, imagine trying to hash all of the new terms you learn in class.

A table would be hard for this, because until you have seen them all, you do not know how many there will be. A more effective way to hash this, is to derive a *hashing function* that is a general strategy.

A *cyrptographic* hash is additionally:

- unique
- not reversible
- similar inputs hash to very different values so they appear uncorrelated

Now lets go through each of these properties

11.1.1. Cryptographic Hashes are unique

This means that two different values we put in should give different results.

For this property alone, a simple function could work:

```
def basic_unique(input):
    return input
```

but this is not a hash because its length would not be constant and not a cryptographic has because it is easily reversible.

11.1.2. Cryptographic Hashes are not reversible

This means that given the hash (output), we cannot compute the message(input).

Any function that gives the same output for two (or more) values meets this criteria.

for example modulus:

```
13%3
```

```
1
```

```
10%3
```

```
1
```

It can be any function that gives the same output for two (or more) values.

but this is not a cryptographic hash

11.1.3. Similar inputs lead to seemingly uncorrelated outpus

We can see this using git:

```
echo "it's finally cool" | git hash-object --stdin
```

We can break down this command:

- git hash-object would take the content you handed to it and merely return the unique key
- `--stdin` option tells git hash-object to get the content to be processed from stdin instead of a file
- the `|` is called a pipe (what we saw before was a redirect) it pipes a *process* output into the next command
- `echo` would write to stdout, withthe pip it passes that to std in of the `git-hash`

it returns the hash to std out, so ha the next command could use it.

```
c70cede405d6534b30debeb70ec84f527788eac3
```

and then we change it just a little bit:

```
echo "it's finlaly cool" | git hash-object --stdin
```

```
bef51664b1a5a1b4b6b9cf5749dd161d5b1808ed
```

and the hash is completely different

```
echo "its finlaly cool" | git hash-object --stdin
```

```
c6489cb5bd5d20dcfce9709868b535fe3eeae59f
```

```
git hash-object --help
```

11.1.4. Hashes are fixed length

So, no matter the size of the input, we get back the same length.

This is good for memory allocation reasons.

We could again write a function that only does this simply:

```
def fixed_len(input):
    '''
    pad or trim
    '''
    len_target=100
    str_in = str(input)
    if len(str_in)< len_target:
        return str_in.ljust(len_target,'-')
    else:
        return str_in[:len_target]
```

11.2. How can hashes be used?

Hashes can then be used for a lot of purposes:

- message integrity (when sending a message, the unhashed message and its hash are both sent; the message is real if the sent message can be hashed to produce the same has)
- password verification (password selected by the user is hashed and the hash is stored; when attempting to login, the input is hashed and the hashes are compared)
- file or data identifier (eg in git)

11.2.1. Hashing in passwords

Passowrds can be encrypted and the encrypted information is stored, then when you submit a candidate password it can compare the hash of the submitted password to the hash that was stored. Since the hashing function is nonreversible, they cannot see the password.

password breach

blog post

Some sites are negligent and store passwords unencrypted, if your browser warns you about such a site, proceed with caution and definitely do not reuse a password you ever use. (you *should never* reuse passwords, but especially do not if there is a warning)

An attacker who gets one of those databases, could build a lookup table. For example, “password” is a bad password because it has been hashed in basically every algorithm and then the value of it can be reversed. Choosing an uncommon password makes it less likely that your password exists in a lookup table.

For example, in SHA-1 the hashing algorithm that git uses

```
echo "password" | git hash-object --stdin
```

```
f3097ab13082b70f67202aab7dd9d1b35b7ceac2
```

11.3. Hashing in Git

In git we hash both the content directly to store it in the database (.git) directory and the commit information.

Recall, when we were working in our toy repo we created an empty repository and then added content directly, we all got the same hash, but when we used git commit our commits had different hashes because we have different names and made the commits at different seconds. We also saw that *two* entries were created in the `.git` directory for the commit.

Git was originally designed to use SHA-1.

Then the SHA-1 collision attack was discovered

Git switched to hardened HSA-1 in response to a collision.

> In that case it adjusts the SHA-1 computation to result in a safe hash. This means that it will compute the regular SHA-1 hash for files without a collision attack, but produce a special hash for files with a collision attack, where both files will

> have a different unpredictable hash. from.

they will change again soon

git uses the SHA hash primarily for uniuqeness, not privacy

It does provide some *security* assurances, because we can check the content against the hash to make sure it is what it matches.

This is a Secure Hashing Algorithm that is derived from cryptography. Because it is secure, no set of mathematical options can directly decrypt an SHA-1 hash. It is designed so that any possible content that we put in it returns a unique key. It uses a combination of bit level operations on the content to produce the unique values.

This means it can produce 2^{160} different hashes. Which makes the probability of a collision very low.

> The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is `p = n(n-1)/2) * (1/2^160))$. $2^{80}`) is 1.2 x 1024 or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.
>
> —A SHORT NOTE ABOUT SHA-1 in the Git Documentation

11.3.1. Workign with git hashes

> **Important**
>
> We had discussed this bit before, while answering other questions, so I skipped it today, but added it here for completeness.

Mostly, a shorter version of the commit is sufficient to be unique, so we can use those to refer to commits by just a few characters:

- minimum 4
- must be unique

by using some options on `git log` to make it easier to read a set of them

```
git log --abbrev-commit --pretty=oneline
```

```
e8ab736 (HEAD -> organization) add more stuff
1e2ab92 (tag: 0.0.1) include readmen content
87c72ae (origin/organization) stop tracking
d2d1fac organized files into foleders and ignore private
a3904a0 Revert "start organizing"
4ceb150 fill in decritoin more
9120d9d start organizing
f17e276 explain files
72b85c7 add a note
991ee65 (origin/main, origin/HEAD, main) Merge pull request #5 from compsys-progtools/organizing_ac
1ee9c9f (origin/organizing_ac) add files for organizing activity
0c12714 (my_branch_checkedoutb, my_branch) Merge pull request #4 from compsys-progtools/1-create-a-readme
c7375fa (origin/1-create-a-readme, 1-create-a-readme) create readme closes #1
0e7c990 start about file closes #3
0373342 Initial commit
```

For most project 7 characters is enough and by default, git will give you 7 digits if you use `--abbrev-commit` and git will automatically use more if needed.

11.4. How are hashes computed?

Hashes are computed on the binary representation of the data, most treat that as numbers and then perform mathematical operations.

[you can see the psuedocode to SHA-1]

> **Important**
>
> this is an example we skipped in class, but I wanted to give some more info

we'll work a small example, Python can convert characters to ints:

```
ord('s')
```

```
115
```

and ints to binary

```
bin(ord('s'))
```

```
'0b1110011'
```

with some extra at the start, so if we just want the binary string

```
str(bin(ord('s')))[2:]
```

```
'1110011'
```

We can use this to make a function that performs an xor based hash.

```
def xor_hash(message,n):
    '''
    a simple hashing algorithm based on xor that can take a message of
    any length and return a version of it with a specific bit length n
    '''
    bitstring = ''.join([str(bin(ord(c)))[2:] for c in message])

    # zero pad to make it a multiple of n
    extra = len(bitstring)%n
    num_pad = (n-extra)*int(extra >0)
    padded_bitstring = bitstring + num_pad*'0'

    # split into length n chunks
    split_bitstring = [padded_bitstring[i:i+n] for i in range(0, len(padded_bitstring), n)]

    # now make them back to integers so we can xor
    bin_int_strings = [int("0b" + bits,2) for bits in split_bitstring]

    # now xor
    out_hash_int = 0
    for n_int in bin_int_strings:
        out_hash_int = out_hash_int^n_int

    # we would need to left zero pad the binary to keep it correct
    bin_str = str(bin(out_hash_int))[2:]
    extra_out = len(bin_str)%n
    num_pad_out = (n-extra_out)*int(extra_out >0)
    padded_out =  num_pad_out*'0' + bin_str
    return padded_out
```

You can see this visualized on python tutor

We can test that we get something the right length

```
xor_hash('hello',8)
```

```
'00001101'
```

for different lengths of input

```
xor_hash('it is almost break',8)
```

```
'00110000'
```

This is a sort of meta function, so we could make simpler, more specific functions for each length or we can use it with different ones to see

```
xor_hash('it is almost break',13)
```

```
'0010111001001'
```

This is not cryptographic though because similar inputs have correlated outputs.

11.5. How do we get to the alphanumeric hashes we see?

Let’s build this up with some review

11.5.1. What is a Number ?

a mathematical object used to count, measure and label

11.6. What is a number system?

While numbers represent **quantities** that conceptually, exist all over, the numbers themselves are a cultural artifact. For example, we all have a way to represent a single item, but that can look very different.

for example I could express the value of a single item in different ways:

- 1
- I

11.6.1. Hindu Arabic Number system

In modern, western cultures, our number system is called the hindu-arabic system, it consists of a set of **numerals**: 0,1,2,3,4,5,6,7,8,9 and is a **place based** system with **base 10**.

11.6.1.1. Where does the name come from?

- invented by Hindu mathematicians in India 600 or earlier
- called “Arabic” numerals in the West because Arab merchants introduced them to Europeans
- slow adoption

11.6.1.2. Numerals

are the visual characters used to represent quantities

11.6.1.3. Place based

We use a **place based** system. That means that the position or place of the symbol changes its meaning. So 1, 10, and 100 are all different values. This system is also a decimal system, or base 10. So we refer to the places and the ones (10^0), the tens (10^1), the hundreds(10^2), etc for all powers of 10.

Number systems can use different characters, use different strategies for representing larger quantities, or both.

Not all systems are place based, for example Roman numerals.

11.6.2. Roman Numerals

is a number system that uses different numerals and is not a place based system

There are symbols for specific values: 1=I, V=5, X=10, L =50, C = 100, D=500, M = 1000.

In this system the subsequent symbols are either added or subtracted, with no (nonidentity) multipliers based on position.

Instead if the symbol to right is the same or smaller, add the two together, if the left symbol is smaller, subtract it from the one on the right.

For example

roman numerals	translation	value (hindu-arabic)
III	1+1+1	3
IV	-1 + 5	4
VI	5 +1	4
XLIX	-10 + 50 -1 + 10	49

11.7. Comparing Bases

11.7.1. Decimal

To represent larger numbers than we have digits on we have a base (10) and then.

$$10 = 10 * 1 + 1 * 0$$

$$22 = 10 * 2 + 1 * 2$$

we have the ones (10^0) place, tens (10^1) place, hundreds (10^2) place etc.

11.7.2. Binary

Binary is any base two system, and it can be represented using any two distinct numerals.

Binary number systems have origins in ancient cultures:

- Egypt (fractions) 1200 BC
- China 9th century BC
- India 2nd century BC

In computer science we use binary because mechanical computers began using relays (open/closed) to implement logical (boolean; based on true/false, also binary) operations and then digital computers use on and off in their circuits.

We typically represent binary using the same hindu-arabic symbols that we use for other numbers, but only the 0 and 1(the first two). We also keep it as a place-based number system so the places are the ones(2^0), twos (2^1), fours (2^2), eights (2^3), etc for all powers of 2.

so in binary, the number of characters in the word `binary` is `110`.

$$10 => 1 * 2^1 + 0 * 2^0 = 1 * 2 + 0 * 1 = 2$$

so 10 in binary is 2 in decimal

11.7.3. Octal

Is base 8.

We use hindu-arabic symbols, 0,1,2,3,4,5,6,7 (the first eight). Then nine is represented as 11.

In computer science, this numbering system was popular in 6 bit and 12 bit computers, but is has origins before that.

Native Americans using the Yuki Language (based in what is now California)used an octal system because they count using the spaces between fingers and speakers of the Pamean languages in Mexico count on knuckles in a closed fist. Europeans debated using decimal vs octal in the 1600-1800s for various reasons because 8 is better for math mostly. It is also found in Chinese texts dating to 1000BC.

Some examples:

- $$10 \rightarrow 1 * 8^1 + 0 * 8^0 = 1 * 8 + 0 * 1 = 8$$
- $$401 \rightarrow 4 * 8^2 + 0 * 8^1 + 1 * 8^0 = 4 * 64 + 1 * 0 = 257$$

In computer science we use octal a lot because it reduces every 3 bits of a number in binary to a single character. So for a large number, in binary say `101110001100` we can change to `5614` which is easier to read, for a person.

`101110001100` $\rightarrow$ `101 110 001 100` $\rightarrow$ `5614`

11.7.4. Hexadecimal

base 16, common in CS because its 4 bits. we use 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

This is commonly used for representing colors

Color Picker: 

Some examples:

$$A4 \rightarrow (9+1) * 16^1 + 4 * 16^0 = 10 * 16 + 4 * 1 = 164$$

$$E40 \rightarrow 14 * 16^2 + 4 * 16^1 + 0 * 16^0 = 14 * 256 + 4 * 16 + 0 * 1 = 3648$$

This is how the git hash is 160 bits, or 20 bytes (one byte is 8 bits) but we represent it as 40 characters. 160/4=40.

```
python
```

```
Python 3.12.4 (v3.12.4:8e8a4baf65, Jun  6 2024, 17:33:18) [Clang 13.0.0 (clang-1300.0.29.30)] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> len('f3097ab13082b70f67202aab7dd9d1b35b7ceac2')
40
>>> exit()
```

11.8. Prepare for Next Class

1. (before lab on Tuesday ideally) start commenting/expressing interest on build/explore ideas.
   - build template
   - explore instructions
2. Create a file `gitcommandsbreakdown.md` and for each command in the template below break down what steps it must do based on what we have learned so far about git objects. I started the first one as an example. Next class, we will make a commit using plumbing commands, so thinking about what you already know about commits will prepare you to learn this material.

```
# What git commands do

## `git status`

- check the branch of the HEAD pointer
- compare the HEAD pointer to the FETCH_HEAD, if different trace back through parent commits to find out
- compare the snapshot at the HEAD pointer's commit to the current working directory
- if staging is not empty, compare that to the working directory

## `git commit`

-

## `git add`

-
```

11.9. Badges

Review | **Practice**

1. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
2. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include

links to your sources and be sure that the sources are trustworthy.
3. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts.md with a title `# Git counts`. How many files would have to exist to reach that number of objects assuming every fiile was edited in each of two commits? *If you get stuck, outline what you know and then request a review.*

11.10. Experience Report Evidence

11.11. Questions After Today's Class

11.11.1. Questions that are good explore badge topics:

- How will quantum computing affect cryptography?

11.11.2. How can hashes be used in programming tasks?

often for storing or for encrypting.

11.11.3. is hashing used in creating IP Addresses?

Not in the addressing, but the content that is sent is sometimes hashed.

11.11.4. How do you convert Roman numerals into their corresponding decimal values?

take what each symbol represents and add or subtract based on their order. the notes should be more clear than in class was.

12. How does git *make* a commit?

Today we will dig into how git really works. This will be a deep dive and provide a lot of details about how git creates a commit. It will reinforce important **concepts**, which is of practical use when fixing things give you some ideas about how you might fix things when things go wrong.

Later, we will build on this more on the practical side, but these **concepts** are very important for making sense of the more practical aspects of fixing things in git.

This deep dive in git is to help you build a correct, flexbile understanding of git so that you can use it independently and efficiently. The plumbing commands do not need to be a part of your daily use of git, but they are the way that we can dig in and see what *actually* happens when git creates a commit.

this is also to serve as an example method you could apply in understanding another complex system

Inspecting a system's components is a really good way to understand it and correctly understanding it will impact your ability to ask good questions and even look up the right thing to do when you need to fix things.

Also, looking at the parts of git is a good way to reinforce specific design patterns that are common in CS in a practical way. This means that today we will also:

- review and practice with the bash commands we have seen so far
- see a practical example of hashing
- reinforce through examples what a pointer does

Navigate to your folder for class

```
ls
```

```
fall24-brownsarahm      tiny-book
gh-inclass-brownsarahm
```

it should have your other repos

and, *not* be a git repo, we can confirm with:

```
git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```

In contrast, if we are in a repo

```
cd fall24-brownsarahm/
```

```
git status
```

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

nothing to commit, working tree clean
```

the command works

If yours worked, go up one level to be out of a repo:

```
cd ../
```

```
git status
```

```
fatal: not a git repository (or any of the parent directories): .git
```

> **Important**
>
> Today there were a lot of prismia comprehension check questions.
>
> If you struggled with them, you should be sure to check that you now understand the correct answer

12.1. Creating a repo from scratch

We can create an empty repo from scratch using `git init <path>`

Last time we used an existing directory like `git init .` because we were working in the directory that already existed

Today we will create a new directory called `test` and initialize it as a repo at the same time:

```
git init test
```

```
hint: Using 'master' as the name for the initial branch. This default branch name
hint: is subject to change. To configure the initial branch name to use in all
hint: of your new repositories, which will suppress this warning, call:
hint:
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/brownsarahm/Documents/inclass/systems/test/.git/
```

I get this message again, see context from previous class

We can see what it did by first looking at the working directory

```
ls
```

```
fall24-brownsarahm      test
gh-inclass-brownsarahm  tiny-book
```

it made a new folder named as we said

and we can go into that directory

```
cd test/
```

it is empty as expected:

```
ls
```

```

```

and then rename the branch

```
git branch -m main
```

To clarify we will look at the status

```
git status
```

Notice that there are no commits, and no origin.

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

there are no remotes at all

```
git remote
```

we do have the .git hidden directory though

```
ls -a
```

```
.       ..      .git
```

```
ls  .git/
```

```
HEAD            description     info            refs
config          hooks           objects
```

Notice that there are no commits, and no origin.

```
ls ../gh-inclass-brownsarahm/.git
```

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

in my inclass repo I have other files because as we do various git operations, we create other files for example the `COMMIT_EDITMSG` is the file that opens in vim when we forget the `-m` option

12.2. Searching the file system

We can use the bash command `find` to search the file *system* note that this does not search the **contents** of the files, just the names.

```
find .git/objects/
```

```
.git/objects/
.git/objects//pack
.git/objects//info
```

we have a few items in that directory and the directory itself.

We can limit by type, to only files with the `-type` option set to `f`

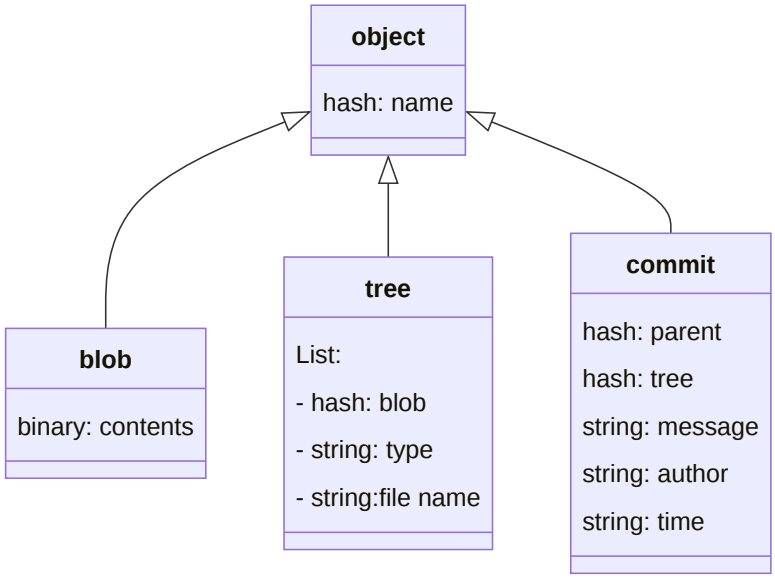
```
find .git/objects/ -type f
```

And we have no results. We have no objects yet. Because this is an empty repo

12.3. Git Objects

Remember our 3 types of objects

- blob objects: the content of your files (data)
- tree objects: stores file names and groups files together (organization)
- Commit Objects: stores information about the sha values of the snapshots



12.3.1. How to create an object

All git objects are files stored with the name that is the hash of the content in the file

Remember git is a content-addressable file systsem… so it uses key- value pairs.

Let's create our first git object. git uses hashes as the key. We give the hashing function some content, it applies the algorithm and returns us the hash as the reference to that object. We can also write to our .git directory with this.

The `git hash-object` command works on files, but we do not have any files yet. We can create a file, but we do not have to. Rememember, **everything** is a file.

When we use things like `echo` it writes to the stdout file.

```
echo "test content"
```

```
test content
```

which shows on our terminal. We can us a pipe to connect the stdout of on command to the stdin of the next.

```
echo "test content" | git hash-object --stdin
```

We can break down this command:

- git hash-object would take the content you handed to it and merely return the unique key
- `--stdin` option tells git hash-object to get the content to be processed from stdin instead of a file
- the `|` is called a pipe (what we saw before was a redirect) it pipes a *process* output into the next command
- `echo` would write to stdout, withthe pip it passes that to std in of the `git-hash`

we get back the hash:

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

Warning

if you have an odd number of quotes `"` or `'` or more open brackets `([{` than corresponding close brackets `}])` bash will wait fo you to finish. Whenever your regular prompt is replaced by `>` it is waiting for you to finish the current command.

and then it runs once we close with `"`

```
kjsjslf
sdf
jjsdkf
```

No

pipe
we'
use
Pipe
com

Now let's run it again with a slight modification. `-w` option tells the command to also write that object to the database

```
echo "test content" | git hash-object -w --stdin
```

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we can check if it wrote to the repository.

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we see a file that it was supposed to have!

We do not have any files or commits though still.

```
git status
```

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

```
ls
```

12.3.2. Viewing git objects

We can try with `cat`

```
cat .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

```
xK??OR04f(I-.QH??+I?+?K?
```

This is binary output that we cannot understand. Fortunately, git provides a utility. We can use `cat-file` to use the object by referencing at least 4 characters that are unique from the full hash, not the file name. (`7046` will not work, but `d670` will)

`cat-file` requires *an* option `-p` is for pretty print

```
git cat-file -p d670
```

```
test content
```

where we see the content we put in to the hashing function

and `-t` is for type

```
git cat-file -t d670
```

```
blob
```

> **Hint**
>
> You can always use the `--help` on any `git` command to learn more about its options

These options cannot be stacked

```
git cat-file -pt d670
```

```
error: options '-t' and '-p' cannot be used together
```

the developers did not implement it to work, but it also does not really make sense to output both of them, because they each provide a different type of content. Using both asks for 2 different types of content which would then not be usable in a basic pipe so it would be bad design in terms of the unix philosophy to do that.

12.3.3. Hashing a file

let's create a file

```
echo "version 1" > test.txt
```

first we will check in with git:

```
git status
```

```
On branch main

No commits yet

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

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

it can see the file, but it is untracked.

and now, we store it, by hashing it

```
git hash-object -w test.txt
```

```
83baae61804e65cc73a7201a7252750c76066a30
```

this returns the hash, this is still a hash that is the same for all of us.

and again with git status

```
git status
```

```
On branch main

No commits yet

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

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

still untracked

but the file does exist in the git objects:

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

we can check the type of this new object:

```
git cat-file -t 83baa
```

```
blob
```

also a blob

```
git cat-file -p 83baa
```

```
version 1
```

Now this is the status of our repo.

d67046
+ "test content"
+(blob)

83baae
+ Version 1
+(blob)

Notice, however, that we only have one file in the working directory.

```
ls
```

```
test.txt
```

it is the one test.txt, the first blob we made had no file in the working directory associated to it.

the workign directory and the git repo are not strictly the same thing, and can be different like this. Mostly they will stay in closer relationship that we currently have unless we use plumbling commands, but it is good to build a solid understanding of how the `.git` directory relates to your working directory.

So far, even though we have hashed the object, git still thinks the file is untracked, because it is not in the tree and there are no commits that point to that part of the tree.

12.4. Updating the Index

Now, we can add our file as it is to the index.

```
git update-index --add --cacheinfo 100644 \
> 83baae61804e65cc73a7201a7252750c76066a30
```

```
error: option 'cacheinfo' expects <mode>,<sha1>,<path>
```

- the `\` lets us wrap onto a second line, the `>` above is the new prompt for the second line
- this the plumbing command `git update-index` updates (or in this case creates an index, the staging area of our repository)
- the `--add` option is because the file doesn't yet exist in our staging area (we don't even have a staging area set up yet)
- `--cacheinfo` because the file we're adding isn't in your directory but is in the database.
- in this case, we're specifying a mode of 100644, which means it's a normal file.
- then the hash object we want to add to the index (the content) in our case, we want the hash of the first version of the file, not the most recent one.
- finally the file name of that content

I forgot the file name above, so I used my up arrow key to get the command back:

```
git update-index --add --cacheinfo 100644 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

note that the `\` I originally typed and the `>` from the prompt are not there.

this command had no output, so let check git status.

Again, we check in with status

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt
```

We have the files staged as expected

Now the file is staged.

Let's edit it further.

```
echo "version 2" >> test.txt
```

We can look at the content to ensure it as expected

```
cat test.txt
```

```
version 1
version 2
```

So the file has two lines

Now check status again.

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

We added the first version of the file to the staging area, so that version is ready to commit but we have changed the version in our working directory relative to the version from the hash object that we put in the staging area so we *also* have changes not staged.

we can see we do not yet have a new object for the new version of the file

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

We can hash and store this version too.

```
git hash-object -w test.txt
```

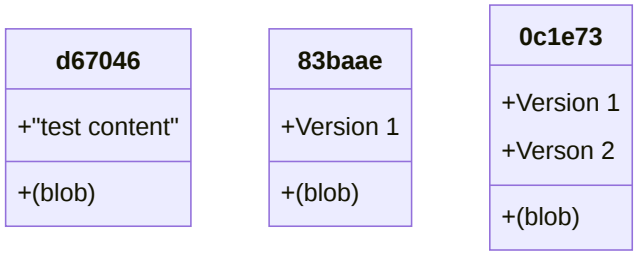
```
0c1e7391ca4e59584f8b773ecdbbb9467eba1547
```

We can then look again at our list of objects.

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

our new object is there with the first two



So now our repo has 3 items, all blobs

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

hashing the object does not impact the index, which is what git status uses

12.4.1. Preparing to Commit

When we work with porcelain commands, we use add then commit. We have staged the file, which we know is what happens when we add. What else has to happen to make a commit.

We know that commits are comprised of:

- a message
- author and times stamp info
- a pointer to a tree
- a pointer to the parent (except the first commit)

We do not have any of these items yet.

Let's make a tree next.

Now we can write a tree from the index,

```
git write-tree
```

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

and we get a hash

Lets examine the tree, first check the type

```
git cat-file -t d832
```

```
tree
```

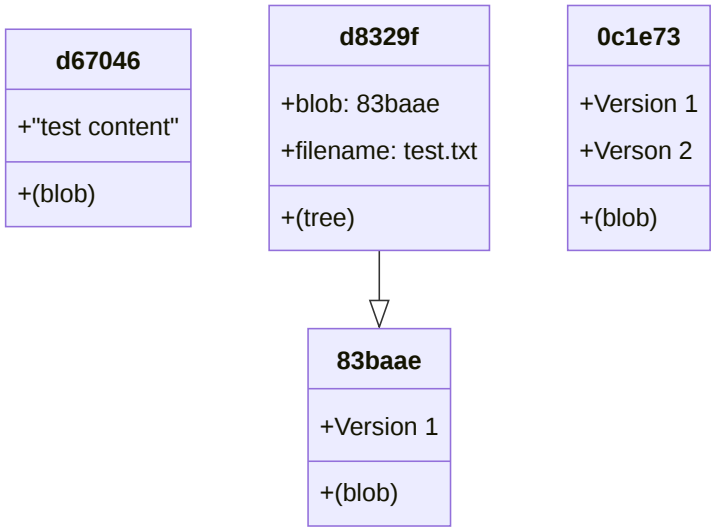
it is as expected

and now we can look at its contents

```
git cat-file -p d832
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

Now this is the status of our repo:



two blobs that are unlinked to anything and one blob that is included in a tree

Again, we will check in with git via `git status`

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

Nothing has changed, making the tree does not yet make the commit

This only keeps track of the objects, there are also still the HEAD that we have not dealt with and the index.

12.4.2. Creating a commit manually

We can echo a commit message through a pipe into the commit-tree plumbing function to commit a particular hashed object.

the `git commit-tree` command requires a message via stdin and the tree hash. We will use stdin and a pipe for the message

```
echo "first commit" | git commit-tree d832
```

```
097898b3f2a5a10f2adb96931da78666ff858002
```

and we get back a hash. But notice that this hash is unique for each of us. Because the commit has information about the time stamp and our user.

we can look at this object too:

```
git cat-file -p 0978
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1729186238 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1729186238 -0400

first commit
```

The above hash is the one I got during class, but I did this previously I had a different hash (`d450567fec96cbd8dd514313db9bcb96ad7664b0`) even though I have the same name and e-mail because the time changed.

We can also look at its type

```
git cat-file -t 0978
```

```
commit
```

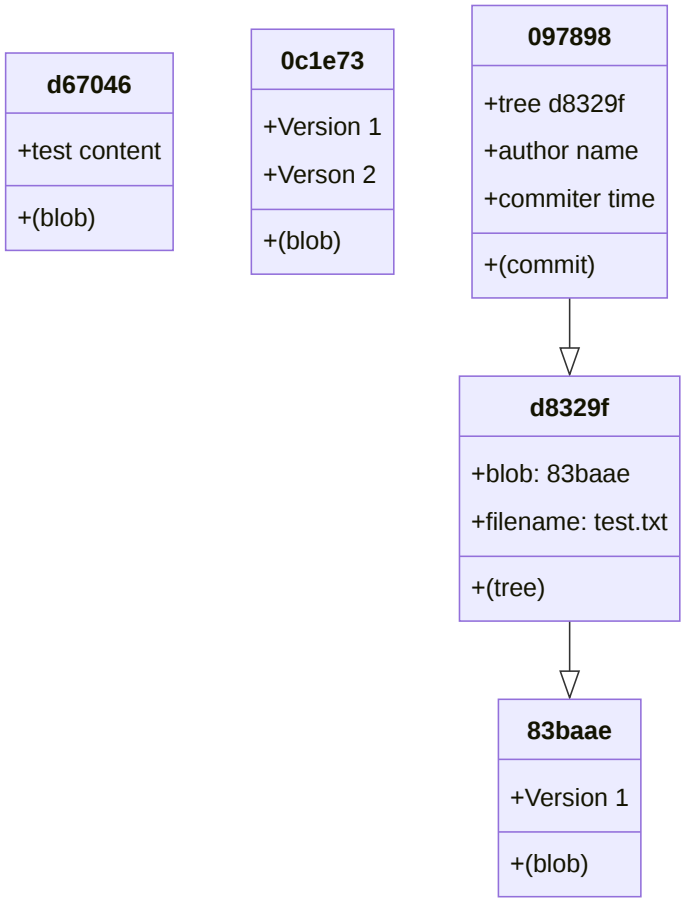
Now we check the final list of objects that we have for today

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//09/7898b3f2a5a10f2adb96931da78666ff858002
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

You should also have 5 objects like me, 3 blobs, one tree, one commit. Four of your objects (blobs & tree) will have the same has, but your commit will have a different hash. I highlghted my commit, that is the one that you should **not** have.

Visually, this is what our repo looks like:



12.5. What does git status do?

compares the working directory to the current state of the active branch

- we cansee the working directory with: `ls`
- we can see the active branch in the `HEAD` file
- what is its status?

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

we see it is “on main” this is because we set the branch to main , but since we have not written there, we have to do it directly. Notice that when we use the porcelain command for commit, it does this automatically; the porcelain commands do many

things.

Notice, git says we have no commits yet even though we have written a commit.

In our case because we made the commit manually, we did not update the branch.

This is because the main branch does not *point* to any commit.

We can check that a commit type object exists

```
git cat-file -t 0978
```

```
commit
```

We can verify by looking at the `HEAD` file

```
cat .git/HEAD
```

```
ref: refs/heads/main
```

and then viewing that file

```
cat .git/refs/heads/main
```

```
cat: .git/refs/heads/main: No such file or directory
```

which does not even exist!

we can also look at the folder where branch pointer files typicall live

```
ls .git/refs/heads/
```

nothing exists there yet!

```

```

12.6. Git References

We can make that file manually

```
echo 097898b3f2a5a10f2adb96931da78666ff858002> .git/refs/heads/main
```

> **Important**
>
> This file needs to have the **full** hash, not only a few digits.

and back to git status

```
git status
```

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

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

now it is as we expect!

12.7. Prepare for Next Class

2. Review the GitHub Action files in your KWL repo and make note of what if any syntax in there is unfamilar. (note that link will not work on the rendered website, but will work on badge issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue. (to see another application of paths)
4. Check out the github action marketplace to see other actions that are available and try to get a casual level of understanding of the *types* of things that people use actions for.

12.8. Badges

Review | **Practice**

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plumbing and porcelain. The table should have two columns, one for each type of command (plubming and porcelain). Each row should have one git porcelain command and at least one of the corresponding git plumbing command(s). Include two rows: `add` and `commit`.

12.9. Experience Report Evidence

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//09/7898b3f2a5a10f2adb96931da78666ff858002
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

> **Important**
>
> You need to have a test repo that matches this for Lab next week

Generate your evidence with the following in your test repo

```
find .git/objects/ -type f > testobj.md
```

then append the contents of your commit object to that file.

Move the `testobj.md` to your kwl repo in the experiences folder.

12.10. Questions After Today's Class

12.10.1. How can certain parts of commits be accessed and pulled for use?

We saw that we can look at the output of the `git cat-file` command.

```
git cat-file -p 0978
```

This tells us the tree (which would tell us the blobs) and any future commits would have a parent too. Let's focus on the tree and try to extract it.

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1729186238 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1729186238 -0400

first commit
```

We can pipe this output into another command to take a subset of it.

This could be a custom written tool (see python click library for example) or you can use `sed` with a regex like expression. This is a really detailed task that is a good use of a generative AI tool. I knew *what* I wanted to do, but could not remember if it was `sed` or `awk` that was the right tool and the regex syntax is not something I use enough to retain. So I asked an LLM "sed or awk to extract the parent of a commit from the git cat-file output" and it gave me:

```
git cat-file -p 0978 | sed -n 's/^tree \(.*\)/\1/p'
```

which I ran to confirm that it worked

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

12.10.2. how to remove objects after running git commit -tree?

they are files you can delete them with `rm`

12.10.3. Different between tree and blob?

blobs contain the *content* of the file, a tree contains a *listing* of the files in a snapshot (or single folder within a snapshot) that associates the blob objects to their file names.

12.10.4. Because of the nature of how git works off of files, if I were to change the ref head for a branch to point to main's current pointer- would it then contain the same contents?

Yes the branch is just a pointer to a particular commit, as PR compares the branches by showing the changes to the files between the snapshots at those two commits.

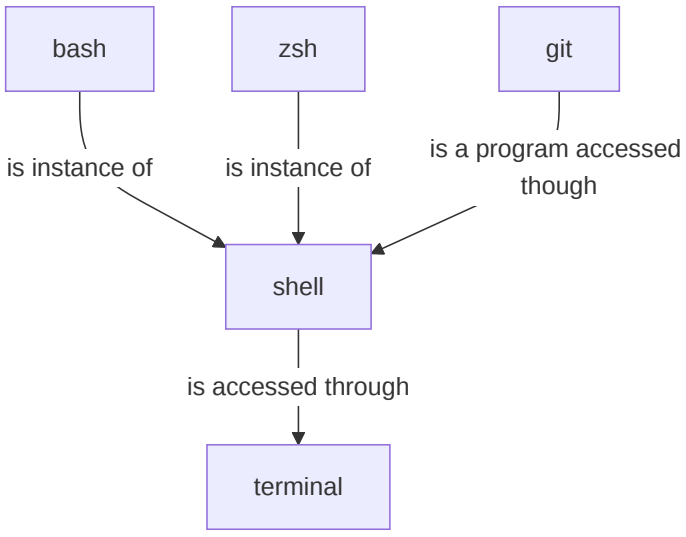
13. How can I automate things with bash?

So far we have used bash commands to navigate our file system as a way to learn about the file system itself. To do this we used commands like:

- `mv`
- `cd`
- `pwd`
- `ls`
- `find`
- `rm`

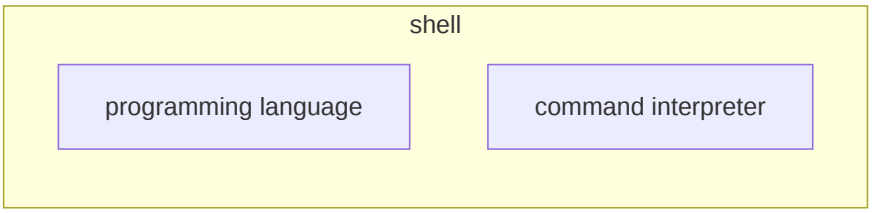
do you remember what each of those does?, no need to respond, just think through it for yourself

Bash is a unix shell for the GNU operating system and it has been adopted in other contexts as well. It is the default shell on Ubuntu linux as well for example (and many others). This is why we teach it.



A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined.

Read the official definition of `bash` and a shell in the bash manual



13.1. Inspecting an example script

Today we will start by inspecting the github action file, first we'll navigate to the folder.

> **Community badge**
>
> read in and annotate using myst/jupyter tools the file of the action file with the excerpts that show the different parts

```
cd fall24-brownsarahm/
```

```

```

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

```
Opening github.com/compsys-progtools/fall24-brownsarahm in your browser.
```

13.2. Variables in Bash

From the action files what do you think the syntax for a variable in bash is? Give an example by creating a variable called `MYVAR` with the value `my_val`

We can create variables

```
MY_NAME='sarah'
```

notice that there are **no spaces** around the `=`. spaces in bash separate commands and options, so they cannot be in variable declarations.

and use them with a `$` before the variable name.

```
echo $MY_NAME
```

```
sarah
```

This variable is local, in memory, to the current terminal window, so if we open a separate window and try `echo $NAME` it will not work. We can also see that it does not create any file changes.

A common mistake is to put a space around the `=` sign, this is actually considered **good style** in many other languages.

```
MY_VAR = 'sarah'
```

```
-bash: MY_VAR: command not found
```

In bash, however, this creates an error. When there is a space after `MY_VAR`, `bash` tried to interpet `MY_VAR` as a bash command, but then it does not find it, so it gives an error.

```
echo $MY_NAME
```

```
sarah
```

13.3. Environment Variables

Environment variables are global.

A common one is the `PATH`

```
echo $PATH
```

```
/Library/Frameworks/Python.framework/Versions/3.12/bin:/usr/local/bin:/System/Cryptexes/App/usr/bin:/usr/
```

The `$` is essential syntax for recalling variables in bash. If we forget it, it treats it as a literal

```
echo PATH
```

```
PATH
```

so we get the variable **name** out instead of the variable **value**

the variables do still work if we chnge our path:

```
cd ../gh-inclass-brownsarahm/
```

```
echo $MY_NAME
```

```
sarah
```

You can store environment variables to be set each time you start a terminal in your profile.

- On MacOS this file is: `~/.bash_profile`
- on linux it is `~/.baschrc`

```
cat ~/.bash_profile
```

```
drsmb="/Users/brownsarahm/Documents/web/drsmb-co.github.io"

alias pip=pip3
alias python=python3

export PS1="\u@\W $ "


# Setting PATH for Python 3.12
# The original version is saved in .bash_profile.pysave
PATH="/Library/Frameworks/Python.framework/Versions/3.12/bin:${PATH}"
export PATH
```

A common one you might want to set is:

- `PS1` the primary prompt line. Content you can use is documented in the gnu docs
- `alias` lets you set another name for a program, for exmaple I use `pip` to call `pip3`

13.4. GH Actions as an example of scripts

Check out the github action marketplace to see other actions that are available and try to get a casual level of understanding of the *types* of things that people use actions for.

13.5. Bash Loops

We can also make loops like

```
for loopvar in list
do
# loop body
echo $loopvar
done
```

So, for example:

```
for name in Sarah Aiden Jad
> do
> echo $name
> done
```

Notes:

- The `>` is not typed, it is what happens since the interpreter knows that after we write the first line, the command is not complete.
- a list in bash is items with spaces, no brackets here `Sarah Aiden Jad`
-

```
Sarah
Aiden
Jad
```

and it outputs each name as expected

When we get the command back with the up arrow key, it puts it all on one line, because it was one command. The `;` (semicolon) separates the “lines”

we can change the loop variable to be anything, but good practice is something semantic, unlike the following:

```
for HI in Sarah Aiden Jad; do echo $HI; done
```

```
Sarah
Aiden
Jad
```

and we can add more to the list

```
for HI in Sarah Aiden Jad Jordan Johsua; do echo $HI; done
```

```
Sarah
Aiden
Jad
Jordan
Johsua
```

13.6. Nesting commands

Note: bash provides a `date` command that returns the date

```
date
```

```
Tue Oct 22 13:14:32 EDT 2024
```

```
ls
```

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

We can run a command to generate the list by putting it inside `$()` to run that command first. Think kind of like PEMDAS and the `$` in bash is for variables.

```
for file in  $(ls)
> do
> echo $file
> done
```

the `$()` tells bash to run that command first and then hold its output as a variable for use elsewhere

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

13.7. Conditionals in bash

We can also do conditional statements

```
if test -f docs
> then
> echo "file"
> fi
```

the key parts of this:

- `test` checks if a file or directory exists
- the `-f` option makes it check if the item is a *file*
- what to do if the condition is met goes after a `then` keyword
- the `fi` (backwards `if`) closes the if statment

```

```

this outputs nothing, because `docs` is a folder, but if we change the file it checks:

```
if test -f README.md; then echo "file"; fi
```

```
file
```

We can put the if inside of the loop, but this is getting long so typing it all without errors is hard.

13.8. Script files

We can put our script into a file

```
nano filecheck.sh
```

```

```

```
cat filecheck.sh
```

```
for file in $(ls)
do
  if test -f $file
  then
    echo $file "is a file"
  fi
done
```

and run it with `bash <filename>`

```
bash filecheck.sh
```

```
CONTRIBUTING.md is a file
LICENSE.md is a file
README.md is a file
filecheck.sh is a file
scratch.ipynb is a file
setup.py is a file
```

the extension can be anything

```
mv filecheck.sh filecheck
```

```
bash filecheck
```

```
CONTRIBUTING.md is a file
LICENSE.md is a file
README.md is a file
filecheck is a file
scratch.ipynb is a file
setup.py is a file
```

and it still works

but the extension is good practice, so we will put it back

```
mv filecheck filecheck.sh
```

13.9. `gh` CLI operations

When you are working sometimes it is helpful to be able to manipulate (or create) issues, pull requests or even releases from the command line.

This is how I post announcements. I work on the notes (a markdown file) in vs code and then I use the vscode terminal to commit, push, create a tag, and create a release. I can post the notes and notify you all that they are posted without leaving VScode at all; this makes it much simpler/faster than it would be using Brightspace

We can also search and filter them by piping the output to `grep` which searches the **contents** of a file (including stdin). We previously searched the file **names** with `find`. So `find` searches the paths that exist and `grep` actually reads the contents of the files, it does so faster than many other languages would be.

```
gh issue list -s all
```

```
Showing 3 of 3 issues in compsys-progtools/gh-inclass-brownsarahm that match your search

ID  TITLE                     LABELS  UPDATED
#3  Create an about file              about 1 month ago
#2  Create a Add a classmate          about 1 month ago
#1  Create a README                   about 1 month ago
```

let's create one more issue

```
gh issue create
```

```
Creating issue in compsys-progtools/gh-inclass-brownsarahm

? Title question
? Body <Received>
? What's next? Submit
https://github.com/compsys-progtools/gh-inclass-brownsarahm/issues/6
```

no body, title "question"

```
gh issue list -s all
```

```
Showing 4 of 4 issues in compsys-progtools/gh-inclass-brownsarahm that match your search

ID  TITLE                     LABELS  UPDATED
#6  question                          less than a minute ago
#3  Create an about file              about 1 month ago
#2  Create a Add a classmate          about 1 month ago
#1  Create a README                   about 1 month ago
```

`grep` can be used with pattern matching as well

Learning more about `grep` is a good explore badge topic

```
gh issue list -s all | grep "Create"
```

```
3       CLOSED  Create an about file            2024-09-12T17:21:42Z
2       OPEN    Create a Add a classmate                2024-09-12T17:18:02Z
1       CLOSED  Create a README         2024-09-17T16:50:52Z
```

We can use `awk` to separate out only the number from the output

```
gh issue list -s all | awk '{print $1}'
```

```
6
3
2
1
```

we can also use multiple pipes

```
gh issue list -s all | grep "Create" | awk '{print $1}'
```

```
3
2
1
```

and we can put htis in a script

```
for issue in $(gh issue list -s all | grep "Create" | awk '{print $1}')
do
gh issue view $issue >> combined_issues.md
done
```

then run that script:

```
bash combine.sh
```

and loook at the output

```
cat combined_issues.md
```

````
title:  Create an about file
state:  CLOSED
author: github-actions
labels:
comments:       0
assignees:
projects:
milestone:
number: 3
--
Add a file `about.md` that has your name and expected graduation in it
follow along to close this issue with  commit.
title:  Create a Add a classmate
state:  OPEN
author: github-actions
labels:
comments:       0
assignees:
projects:
milestone:
number: 2
--
owner:
- [ ] give a class mate access to the repo
- [ ] assign this issue to them

classmate:
- [ ] add `classmate.md` with your name and expected graduation on  a bracn `classmate`
- [ ] open a PR that will close this issue
```
title:  Create a README
state:  CLOSED
author: github-actions
labels:
comments:       0
assignees:      brownsarahm
projects:
milestone:
number: 1
--
A README on GitHub, is ideally a markdown file, github will render that nicely

### Steps
- [x] Add self as assignee to this issue
- [x] check off another item
- [x] use edit to see how they look in plain text mode, the check this off there

create the file and close this issue with your commit message

contents:
```
# GitHub Practice

Name: <your name here>
```
````

13.10. Prepare for Next Class

1. think about what you know about networking
2. make sure you have putty if using windows
3. get the big ideas of hpc, by reading this IBM intro page and some hypothetical people who would attend an HPC carpentry workshop. Make a list of key terms as an issue comment
````

13.11. Badges

Review | **Practice**

1. Update your KWL Chart learned column with what you've learned
2. write a bash script to make it so that `cat`ing the the files in your `gh_inclass` repo does not make the prompt move

13.12. Experience Report Evidence

13.13. Questions After Today's Class

13.13.1. Explore topics from questions

- `zsh` vs `bash`

13.13.2. Where can I find the documentation for all bash syntax?

the bash manual is the official reference

13.13.3. How do we know when to use $ in bash when using varaibles?

Whenever you want to use the value of a variable, not declare it.

13.13.4. How can I look at the bash profile on windows? As the commands for Linux and MacOS didn't work when we looked at /.bash_profile.

I think by default it does not rely on one, but you can create one, you should create the file `.bash_profile`

13.13.5. Why do we use the dollar sign ($) in the command line, instead of other symbols like # or *?

To the best of my knowledge it is just a syntactical choice that was made by the developers, but trying to find the actual history of this from reputable sources could also be an explore idea

13.13.6. What is the use for `#!/bin/bash` Why is it not needed all the time?

`#!/bin/bash` tells the interpretter which shell to use. Since we ran our file like `bash <filename>` we did not need it.

We *would* need it for running like `./<filename>`

the `#!` is calked the shebang. more in the bash docs

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-specifc branch to `main`.
3. when it is approved, merge into main.

> **Ti**
>
> You
> on y

Minimum Rows

> **Warning**
>
> To be updated

Required Files

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

date	file	type
2024-09-10	brain.md	/_practice
2024-09-12	gitoffline.md	/_practice
2024-09-12	gitoffline.md	/_review
2024-09-17	branches.md	/_review
2024-09-17	branches-forks.md	/_practice
2024-09-19	software.md	/_prepare
2024-09-19	terminal_review.md	/_review
2024-09-24	terminal_organization_adv.md	/_practice
2024-09-26	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	/_practice
2024-10-01	commit-def.md	/_review
2024-10-08	gitdef.md	/_prepare
2024-10-08	idethoughts.md	/_prepare
2024-10-08	workflows.md	/_practice
2024-10-08	workflows.md	/_practice
2024-10-08	favorite_git_workflow.md	/_review
2024-10-10	hash_num_prep.md	/_prepare
2024-10-10	gitcounts_scenarios.md	/_practice
2024-10-10	gitcounts.md	/_review
2024-10-17	gitplumbingdetail.md	/_practice
2024-10-17	gitislike.md	/_practice
2024-10-17	gitplumbingreview.md	/_review

Team Repo

> **Warning**
>
> We will not use this in spring 2024

Contributions

Your team repo is a place to build up a glossary of key terms and a “cookbook” of “recipes” of common things you might want to do on the shell, bash commands, git commands and others.

For the glossary, follow the jupyterbook syntax.

For the cookbook, use standard markdown.

to denote code inline `use single backticks`

```
to denote code inline `use single backticks`
```

to make a code block use 3 back ticks

````
```
to make a code block use 3 back ticks
```
````

To nest blocks use increasing numbers of back ticks.

To make a link, `[show the text in squarebrackets](url/in/parenthesis)`

Collaboration

You will be in a “team” that is your built in collaboration group to practice using Git Collaboratively.
There will be assignments that are to be completed in that repo as well. These activities will be marked accordingly. You will take turns and each of you is required to do the initialization step on a recurring basis.

This is also where you can ask questions and draft definitions to things.

Peer Review

If there are minor errors/typos, suggest corrections inline.

In your summary comments answer the following:

- Is the contribution clear and concise? Identify any aspect of the writing that tripped you up as a reader.

- Are the statements in the contribution verifiable (either testable or cited source)? If so, how do you know they are correct?
- Does the contribution offer complete information? That is, does it rely on specific outside knowledge or could another CS student not taking our class understand it?
- Identify one strength in the contribution, and identify one aspect that could be strengthened further.

Choose an action:

- If the suggestions necessary before merging, select **request changes**.
- If it is good enough to merge, mark it **approved** and open a new issue for the broader suggestions.
- If you are unsure, post as a **comment** and invite other group members to join the discussion.

Review Badges

Review After Class

After each class, you will need to review the day's material. This includes reviewing prismia chat to see any questions you got wrong and reading the notes. Review activities will help you to reinforce what we do in class and guide you to practice with the most essential skills of this class, they represent the minimum bar for C level work.

2024-09-05

related notes

Activities:

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)

2024-09-12

related notes

Activities:

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.

2024-09-12

related notes

Activities:

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.

2024-09-17

related notes

Activities:

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 regulary 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.

2024-09-19

related notes

Activities:

1. Update your KWL chart with the new items and any learned items.
2. Clone the course website. Append the commands used and the contents of your `fall2024/.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`)

2024-09-24

related notes

Activities:

3. **lab** Organize the provided messy folder in a Codepsace (details will be provided in lab time). Commit and push the changes. Answer the questions below in your kwl (this) repo in a file called `terminal_organization.md`
4. clone your `messy_repo` locally and append the `history.md` file to your `terminal_organization.md` using a redirect

```
# Terminal File moving reflection

1. How was this activity overall?
1. Did this get easier toward the end?
4. When do you think that using the terminal will be better than using your GUI file explorer?
5. What questions/challenges/ reflections do you have after this?
```

2024-09-26

related notes

Activities:

1. Read today's notes when they are posted. There are imporant 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 <Adherance/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.

2024-10-01

related notes

Activities:

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?

2024-10-01

related notes

Activities:

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?

2024-10-03

related notes

Activities:

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book, by manually adding those files. **do not add the whole template to your repo**, make the content you have already so it can *build* into html. Set it so that the `_build` directory is not under version control.
2. Add `docs-review.md` to your KWL repo and explain the most important things to know about documentation in your own words using other programming concepts you have learned so far. Include in a markdown (same as HTML `<!-- comment -->`) comment the list of CSC courses you have taken for context while we give you feedback.

2024-10-08

related notes

Activities:

1. Read about different workflows in git and describe which one you prefer to work with and why in favorite_git_workflow.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. In commit_contents.md, redirect the content of your most recent commit, its tree, and the contents of one blob. Edit the file or use `echo` to put markdown headings between the different objects. Add a title `# Complete Commit` to the file and at the bottom of the file add `## Reflection` subheading with some notes on how, if at all this excercise helps you understand what a commit is.

2024-10-10

related notes

Activities:

1. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
2. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include links to your sources and be sure that the sources are trustworthy.
3. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts.md with a title `# Git counts`. How many files would have to exist to reach that number of objects assuming every fiile was edited in each of two commits? *If you get stuck, outline what you know and then request a review.*

2024-10-17

related notes

Activities:

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plumbing and porcelain. The table should have two columns, one for each type of command (plubming and porcelain). Each row should have one git porcelain command and at least one of the corresponding git plumbing command(s). Include two rows: `add` and `commit`.

2024-10-22

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. write a bash script to make it so that `cat`ing the the files in your `gh_inclass` repo does not make the prompt move

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[1], but we may ask you questions or leave comments if appropriate, in that event you should reply and then we will approve.

2024-09-10

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 faciltate rendering later (what we will learn will connect a lot of ideas, mapping out where you start, sets you up for success)

2024-09-17

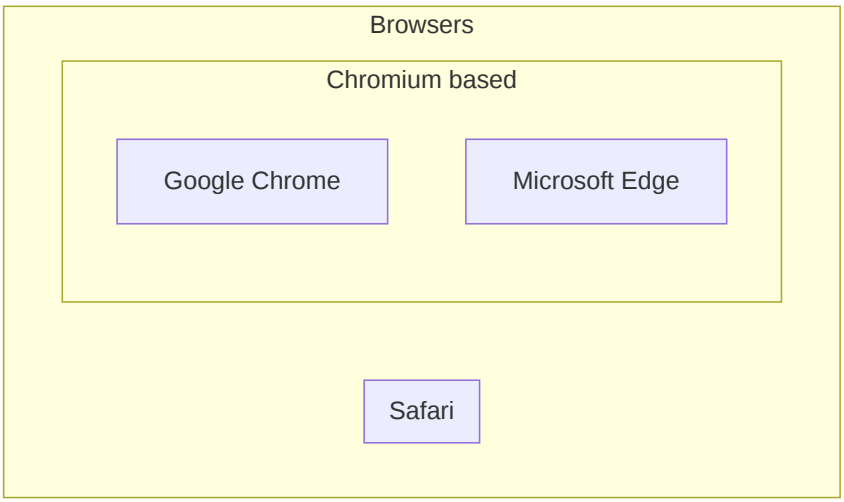
related notes

Activities:

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.

Example

Example "venn diagram " with mermaid subgraphs



2024-09-17

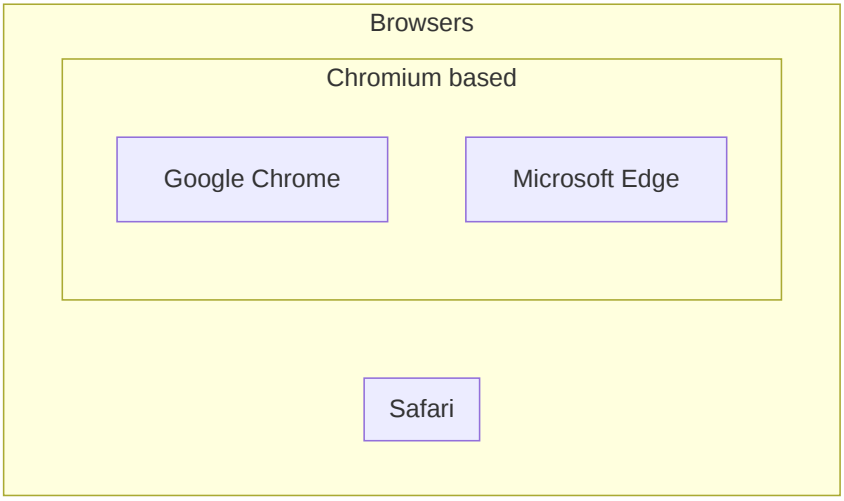
related notes

Activities:

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.

Example

Example “venn diagram “ with mermaid subgraphs



2024-09-19

related notes

Activities:

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

2024-09-24

related notes

Activities:

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 and `VioletVex` is already requested as a reviewer when your experience badge (inclass) 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. Hints: Look for bash commands that we have seen before and `cp` copies a file.

2024-09-26

related notes

Activities:

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 t
```

2024-10-01

related notes

Activities:

1. Learn about hacktoberfest
2. check your plan for success PR for comments and reply or merge if approved
3. read about conventional commits and find some opinions about them in dev blogs, forums (eg reddit) or similar

2024-10-03

related notes

Activities:

1. If on windows, you may need to reinstall gitbash or follow other steps from the gh docs mintty page for the following steps to work locally
2. install jupyterbook *this is **different** from* `jupyter lab` *or* `jupyter notebook` *that 310 uses*
3. Make sure that the `gh` CLI tool works by using it to create an issue called test on your kwl repo with `gh issue create`. If on Windows try reinstalling with mintty
4. Find 3 examples of documentation for libraries, frameworks, or developer tools that you have used and make a post on the class discussion board

2024-10-03

related notes

Activities:

1. If on windows, you may need to reinstall gitbash or follow other steps from the gh docs mintty page for the following steps to work locally
2. install jupyterbook *this is **different** from* `jupyter lab` *or* `jupyter notebook` *that 310 uses*
3. Make sure that the `gh` CLI tool works by using it to create an issue called test on your kwl repo with `gh issue create`. If on Windows try reinstalling with mintty
4. Find 3 examples of documentation for libraries, frameworks, or developer tools that you have used and make a post on the class discussion board

2024-10-08

related notes

Activities:

1. review the notes on what is a commit. In gitdef.md on the branch for this issue, try to describe git in the four ways we described a commit. **the point here is to think about what you know for git and practice remembering it, not "get the right answer"; this is prepare work, we only check that it is complete, not correct**
2. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file below. We will come back to these notes in class later, but it is best to record over a time period instead of trying to remember at that time. Store your notes in your fall24 repo in idethoughts.md on a dedicated `ide_prep` branch. **This is prep for after a few weeks from now, not for October 8; keep this branch open until it is specifically asked for**

2024-10-10

related notes

Activities:

1. Take a few minutes to think what you know about hashing and numbers. Create hash_num_prep.md with two sections: `## Hashing` with a few bullet points summarzing key points about hashing, and `## Numbers` with what types of number representations you know.
2. start commenting/expressing interest on build/explore ideas. Next lab will be focused on that.

- build template
- explore instructions

2024-10-17

related notes

Activities:

1. (before lab on Tuesday ideally) start commenting/expressing interest on build/explore ideas.
    - build template
    - explore instructions

2. Create a file `gitcommandsbreakdown.md` and for each command in the template below break down what steps it must do based on what we have learned so far about git objects. I started the first one as an example. Next class, we will make a commit using plumbing commands, so thinking about what you already know about commits will prepare you to learn this material.

```
# What git commands do

## `git status`

- check the branch of the HEAD pointer
- compare the HEAD pointer to the FETCH_HEAD, if different trace back through parent commits to find out
- compare the snapshot at the HEAD pointer's commit to the current working directory
- if staging is not empty, compare that to the working directory

## `git commit`

-

## `git add`

-
```

2024-10-22

related notes

Activities:

2. Review the GitHub Action files in your KWL repo and make note of what if any syntax in there is unfamilar. (note that link will not work on the rendered website, but will work on badge issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue. (to see another application of paths)
4. Check out the github action marketplace to see other actions that are available and try to get a casual level of understanding of the *types* of things that people use actions for.

2024-10-24

related notes

Activities:

1. think about what you know about networking
2. make sure you have putty if using windows
3. get the big ideas of hpc, by reading this IBM intro page and some hypothetical people who would attend an HPC carpentry workshop. Make a list of key terms as an issue comment

---

[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.

2024-09-05

related notes

Activities:

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. Create a profile readme and include a screenshot of your profile
5. Fill in the first two columns of your KWL chart (content of the PR; named to match the badge name)

2024-09-12

related notes

Activities:

Any steps in a badge marked **lab** are steps that we are going to focus in on during 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 comment.
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_tips.md. Compare what happens

based on what you can see on GitHub and what you can see with git status. Write a scenario with examples of how a person might make mistakes with git add and commit and what to look for to get unstuck.

2024-09-12

related notes

Activities:

Any steps in a badge marked **lab** are steps that we are going to focus in on during 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 comment.
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_tips.md. Compare what happens based on what you can see on GitHub and what you can see with git status. Write a scenario with examples of how a person might make mistakes with git add and commit and what to look for to get unstuck.

2024-09-17

related notes

Activities:

1. Create a merge conflict in your KWL repo on the branch for this issue and resolve it using your favorite IDE, then create one and resolve it on GitHub in browser (this requires the merge conflict to occur on a PR). Describe how you created it, show the files, and describe how your IDE helps or does not help in merge_conflict_comparison.md. Give advice for when you think someone should resolve a merge conflict in GitHub vs using an IDE. (if you do not regulary use an, IDE, try VSCode) *You can put content in the file for this step for the purpose of making the merge conflicts for this exercise.*
2. Learn about GitHub forks and more about git branches(you can also use other resources)
3. In branches-forks.md in your KWL repo, compare and contrast branches and forks; be specific about their relationship. You may use mermaid diagrams if that helps you think through or communicate the ideas. If you use other resources, include them in your file as markdown links.

2024-09-19

related notes

Activities:

1. Update your KWL chart with any learned items.
2. Get set up so that you can contribute to the course website repo from your local system. Note: you can pull from the `compsys-progtools/fall2024` repo, but you do not not have push permission, so there is more to do than clone. Append the commands used and the contents of your `fll2024/.git/config` to a git-remote-practice.md. Then, using a text editor (or IDE), wrap each log with three backticks to make them fenced code blocks and add headings to the sections.
3. learn about options for how git can display commit history. Try out a few different options. Choose two, write them both to a file (from the command line, not copy&paste), gitlog-compare.md. Then, using a text editor (or IDE), wrap each log with three backticks to make them fenced code blocks and then add text to the file describing a use case where that format in particular would be helpful.
4.

- Hint for working offline
  Read about forks and working with remotes.

2024-09-24

related notes

Activities:

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on. For this one in particular, I am going to give you the messy repo in lab.

3. **lab** Organize the provided messy folder (details will be provided in lab time). Commit and push the changes. Clone that repo locally.
4. Organize a folder on your computer (good candidate may be desktop or downloads folder), using only a terminal to make new directories, move files, check what's inside them, etc. Answer reflection questions in a new file, terminal_organization_adv.md in your kwl repo. Tip: Start with a file explorer open, but then try to close it, and use only command line tools to explore and make your choices. If you get stuck, look up additional commands to do acomplish your goals.

```
# Terminal File moving reflection
1. How was this activity overall?
1. Did this get easier toward the end?
2. How was it different working on your own computer compared to the Codespace form?
3. Did you have to look up how to do anything we had not done in class?
4. When do you think that using the terminal will be better than using your GUI file explorer?
5. What questions/challenges/ reflections do you have after this?
6. Append all of the commands you used in lab below. (not from your local computer's history, from the co
```

2024-09-26

related notes

Activities:

1. Read today's notes when they are posted. There are imporant tips and explanation to be sure you did and ideas for explore/build badges.
2. Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Add a `## Unix Philosophy` section to your software.md about how that project adheres to and deviates from the unix philosophy. Be specific, using links to specific lines of code or specific sections in the documentation that support your claims. Provide at least one example of both adhering and deviating from the philosophy and three total examples (that is 2 examples for one side and one for the other). You can see what badge `software.md` was previously assigned in and the original instructions on the KWL file list.

2024-10-01

related notes

Activities:

1. Explore the tools for conventional commits and then pick one to try out. Work on the branch for this badge and use one of the tools that helps making conventional commits (eg in VSCode or a CLI for it)for a series of commits adding "features" and "bug fixes" telling the story of a code project in a file called commit-story.md. For each edit, add short phrases like 'new feature 1', or 'next bug fix' to the single file each time, but use conventional commits for each commit. In total make at least 5 different types of changes (types per conventional commits standard) including 2 breaking changes and at least 10 total commits to the file.
2. learn about options for how git can display commit history. Try out a few different options. Choose two, write them both to a file, gitlog-compare.md. Using a text editor, wrap each log with three backticks to make them "code blocks" and then add text to the file describing a use case where that format in particular would be helpful. **do this after the above so that your git log examples include your conventional commits**

2024-10-01

related notes

Activities:

1. Explore the tools for conventional commits and then pick one to try out. Work on the branch for this badge and use one of the tools that helps making conventional commits (eg in VSCode or a CLI for it)for a series of commits adding "features" and "bug fixes" telling the story of a code project in a file called commit-story.md. For each edit, add short phrases like 'new feature 1', or 'next bug fix' to the single file each time, but use conventional commits for each commit. In total make at least 5 different types of changes (types per conventional commits standard) including 2 breaking changes and at least 10 total commits to the file.
2. learn about options for how git can display commit history. Try out a few different options. Choose two, write them both to a file, gitlog-compare.md. Using a text editor, wrap each log with three backticks to make them "code blocks" and then add text to the file describing a use case where that format in particular would be helpful. **do this after the above so that your git log examples include your conventional commits**

2024-10-03

related notes

Activities:

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book. Set it so that the `_build` directory is not under version control.
2. Learn about the documentation ecosystem in another language that you know using at least one official source and additional sources as you find helpful. In `docs-practice.md` include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibtex based bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources (or a reference manager).

2024-10-08

related notes

Activities:

1. Read about different workflows in git and add responses to the below in a workflows.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. Add the hash of the content of your completed workflows.md file and put that in the comment of your badge PR for this badge. *Try to do this from your local CLI, but full credit even if you use the website interface*

```
## Workflow Reflection

1. Why is it important that git can be used with different workflows?
1. Which workflow do you think you would like to work with best and why?
1. Describe a scenario that might make it better for the whole team to use a workflow other than the one
```

2024-10-10

related notes

Activities:

3. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
4. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include links to your sources and be sure that the sources are trustworthy.
5. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts_scenarios.md with a title `# Git counts`.

Describe 3 scenarios that would get you to that number of objects in terms of what types of objects would exist. For example, what is the maximum number of commits you could have without exceeding that number? How could you get to that number of objects in the fewest number of commits? What might be a typical way to get there? Assume normal git use with porcelain commands, not atypical cases with plubming commands. *If you get stuck, outline what you know and then request a review.*

6. Read about the Learn more about the SHA-1 collision attach
7. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below gittransition.md

gittransition

```
# transition questions

1. Why make the switch? (in detail, not just *an attack*)
2. What impact will the switch have on how git works?
3. Which developers will have the most work to do because of the switch?
```

2024-10-17

related notes

Activities:

1. Read more details about git internals to review what we did in class in greater detail. Make a file gitplumbingdetail.md and create a a table or mermaid diagram that shows the relationship between at least **three** porcelain commands and their corresponding plumbing commands (generally more than one each).
2. Create gitislike.md and explain main git operations we have seen (add, commit, push) in your own words in a way that will either help you remember or how you would explain it to someone else at a high level. This might be analogies or explanations using other programming concepts or concepts from a hobby.

2024-10-22

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. prevent `badges.json` and `badges.yml` from being tracked by git in your fall24 repo
3. write a bash script that prevents `jupyter-book` from giving warnings about files not having a heading by using the file name as a temporary title. (see the badge from 10/3 where you should have converted your repo, or do that one first, this counts as an extension on that if you have not done it)

Hint: Use `sed`'s insert option and head as needed.

Explore idea

modify your script to use a small llm from ollama to automatically insert a sensible title by summarizing the file. **using a commercial chat interface does not qualify, but using an llm locally does**

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 detialed 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” @brownsarahm.

In your issue, describe the question you want to answer or topic to explore and the format you want to use. There is no real template for this, it can be as short as one sentence, but there may be follow up questions.

If you propose something too big, you might be advised to consider a build badge instead. If you propose something too small, you will get ideas as options for how to expand it and you pick which ones.

Where to put the work?

- If you extend a more practice exercise, you can add to the markdown file that the exercise instructs you to create.
- If its a question of your own, add a new file to your KWL repo.
- If you do the work elsewhere, log it like a community badge but in a file called `external_explore_badges.md`

> **Important**
>
> Either way, there must be a separate issue for this work that is also linked to your PR

What should the work look like?

It should look like a blog post, written tutorial, graphic novel, or visual aid with caption. It will likely contain some code excerpts the way the class notes do. Style-wise it can be casual, like how you may talk through a concept with a friend or a more formal, academic tone. What is important is that it clearly demonstrates that you understand the material.

The exact length can vary, but these must go beyond what we do in class in scope

Explore Badge Ideas:

- Extend a more practice:
  - for a more practice that asks you to describe potential uses for a tool, try it out, find or write code excerpts and examine them
  - for a more practice that asks you to try something, try some other options and compare and contrast them. eg “try git in your favorite IDE” -> “try git in three different IDEs, compare and contrast, and make recommendations for novice developers”
- For a topic that left you still a little confused or their was one part that you wanted to know more about. Details your journey from confusion or shallow understanding to a full understanding. This file would include the sources that you used to gather a deeper understanding. eg:
  - Describe how cryptography evolved and what caused it to evolve (i.e. SHA-1 being decrypted)
  - Learn a lot more about a specific number system
  - compare another git host
  - try a different type of version control
- Create a visual aid/memory aid to help remember a topic. Draw inspriation 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 bulid

```
## < Project Tite >

<!-- 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 summrized
- if you are researching something, a report should be 2-4 pages, plus unlimited references in the 2 column ACM format.

This guidance is generative, not limiting, it is to give ideas, but not restrict what you *can* do.

Updates and work in Progress

These can be whatever form is appropriate to your specific project. Your proposal should indicate what form those will take.

Summary Report

This summary report will be added to your kwl repo as a new file `build_report_title.md` where `title` is the (title or a shortened version) from the proposal. Use the template below for the summary report.

```
# <your project title> Summary Report

## Abstract
 <!-- a one paragraph "abstract" type overview of what your project consists of.  This should be written

## Reflection
 <!-- a one paragraph reflection that summarizes challenges faced and what you learned doing your project

## Artifacts

<!-- links to other materials required for assessing the project.  This can be a public facing web resour
```

Collaborative Build rules/procedures

- Each student must submit a proposal PR for tracking purposes. The proposal may be shared text for most sections but the deliverables should indicate what each student will do (or be unique in each proposal).
- the proposal must indicate that it is a pair project, if iteration is required, I will put those comments on both repos but the students should discuss and reply/edit in collaboration
- the project must include code reviews as a part of the workflow links to the PRs on the project repo where the code reviews were completed should be included in the reflection
- each student must complete their own reflection. The abstract can be written together and shared, but the reflection must be unique.

Build Ideas

Your Profile (CV/Resume) Website

Use a static site generator, like one of the below.

Astro

- https://astro.build
- https://docs.astro.build/en/getting-started/
- requires npm installation

General ideas to write a proposal for

- make a vs code extension for this class or another URI CS course
- port the courseutils to rust. crate clap is like the python click package I used to develop the course utils
- buld 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 `@brownsarahm`.

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 <inssert 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=vsls-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 assemlby language before it is executed. must be compiled and re-executed everytime you make a change.

detached head

a state of a git repo where the head pointer is set to a commit without a branch also pointing to the commit

directory

a collection of files typically created for organizational purposes

divergent

git branches that have diverged means that there are different commits that have same parent; there are multipe ways that git could fix this, so you have to tell it what strategy to use

fixed point number

the concept that the decimal point does not move in the number. Cannot represent as wide of a range of values as a floating point number.

floating point number

the concept that the decimal can move within the number (ex. scientific notation; you move the decimal based on the exponent on the 10). can represent more numbers than a fixed point number.

git

a version control tool; it's a fully open source and always free tool, that can be hosted by anyone or used without a host, locally only.

GitHub

a hosting service for git repositories

.gitignore

a file in a git repo that will not add the files that are included in this .gitignore file. Used to prevent files from being unnecessarily committed.

git objects

FIXME something (a file, directory) that is used in git; has a hash associated with it

git Plumbing commands

low level git commands that allow the user to access the inner workings of git.

git Workflow

a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner

HEAD

a file in the .git directory that indicates what is currently checked out (think of the current branch)

merge

putting two branches together so that you can access files in another branch that are not available in yours

merge conflict

when two branches to be merged edit the same lines and git cannot automatically merge the changes

mermaid

mermaid syntax allows user to create precise, detailed diagrams in markdown files.

hash function

the actual function that does the hashing of the input (a key, an object, etc.)

hashing

transforming an input of arbitrary length to a unique fixed length output (the output is called a hash; used in hash tables and when git hashes commits).

integrated development environment

also known as an IDE, puts together all of the tools a developer would need to produce code (source code editor, debugger, ability to run code) into one application so that everything can be done in one place. can also have extra features such as showing your file tree and connecting to git and/or github.

interpreted code

code that is directly executed from a high level language. more expensive computationally because it cannot be optimized and therefore can be slower.

issue

provides the ability to easily track ideas, feedback, tasks, or bugs. branches can be created for specific issues. an issue is open when it is created. pull requests have the ability to close issues. see more in the docs

Linker

a program that links together the object files and libraries to output an executable file.

option

also known as a flag, a parameter to a command line program that change its behavior, different from an argument

path

the “location” of a file or folder(directory) in a computer

pointer

a variable that stores the address of another variable

pull (changes from a repository)

download changes from a remote repository and update the local repository with these changes.

pull request

allow other users to review and request changes on branches. after a pull request recieves approval you can merge the changed content to the main branch.

PR

short for pull request

push (changes to a repository)

to put whatever you were working on from your local machine onto a remote copy of the repository in a version control system.

relative path

the path defined **relative** to another file or the current working directory; may start with a name, includes a single file name or may start with `./`

release

a distribution of your code, related to a git tag

remote

a copy of the repository hosted on a server

repository

a project folder with tracking information in it in the form of a .git directory in it

ROM (Read-Only Memory)

Memory that only gets read by the CPU and is used for instructions

SHA 1

the hashing function that git uses to hash its functions (found to have very serious collisions (two different inputs have same hashes), so a lot of software is switching to SHA 256)

sh

abbr. see shell

shell

a command line interface; allows for access to an operating system

ssh

allows computers to safely connect to networks (such as when we used an ssh key to clone our github repos)

templating

templating is the idea of changing the input or output of a system. For instance, the Jupyter book, instead of outputting the markdown files as markdown files, displays them as HTML pages (with the contents of the markdown file).

terminal

a program that makes shell visible for us and allows for interactions with it

tree objects

type of git object in git that helps store multiple files with their hashes (similar to directories in a file system)

yml

see YAML

YAML

a file specification that stores key-value pairs. It is commonly used for configurations and settings.

zsh

zsh or z shell is built on top of the bash shell and contains new features

General Tips and Resources

This section is for materials that are not specific to this course, but are likely useful. They are not generally required readings or installs, but are options or advice I provide frequently.

on email

- how to e-mail professors

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 cod eand 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

A new bo
programı
Program
available
that linke

Preparing for class will be activities that helpy ou bring your prior knowledge to class in the most helpful way, help me mee

You will be making a lot of documentation of bits, in your own words. You will be directed to try things and make notes. This based on a recommended practices from working devs to [keep a notebook]](https://blog.nelhage.com/2010/05/software-and-lab-notebooks/) or keep a blog and notebook.

Learning in class

> **Important**
>
> My goal is to use class time so that you can be successful with *minimal frustration* while working outside of class time.

Programming requires both practical skills and abstract concepts. During class time, we will cover the practical aspects and introduce the basic concepts. You will get to see the basic practical details and real examples of debugging during class sessions. Learning to debug something you've never encountered before and setting up your programming environment, for example, are *high frustration* activities, when you're learning, because you don't know what you don't know. On the other hand, diving deeper into options and more complex applications of what you have already seen in class, while challenging, is something I'm confident that you can all be successful at with minimal frustration once you've seen basic ideas in class. My goal is that you can repeat the patterns and processes we use in class outside of class to complete assignments, while acknowledging that you will definitely have to look things up and read documentation outside of class.

Each class will open with some time to review what was covered in the last session before adding new material.

To get the most out of class sessions, you should have a laptop with you. During class you should be following along with Dr. Brown. You'll answer questions on Prismia chat, and when appropriate you should try running necessary code to answer those questions. If you encounter errors, share them via Prismia chat so that we can see and help you.

After class

After class, you should practice with the concepts introduced.
This means reviewing the notes: both yours from class and the annotated notes posted to the course website.
When you review the notes, you should be adding comments on tricky aspects of the code and narrative text between code 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 GitHubt 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)

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

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.

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 distributing your code to make it easier for users.

Environments are a tool for dependency management. We will cover thigns 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 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

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
`pwd`	print working directory
`cd <path>`	change directory to path
`mkdir <name>`	make a directory called name
`ls`	list, show the files
`touch`	create an empty file
`echo 'message'`	repeat ‘message’ to stdout
`>`	write redirect
`>>`	append redirect
`rm file`	remove (delete) `file`
`cat`	concatenate a file to standard out (show the file contents)

git commands

command	explanation
`status`	describe what relationship between the working directory and git
`clone <url>`	make a new folder locally and download the repo into it from url, set up a remote to url
`add  <file>`	add file to staging area
`commit -m 'message'`	commit using the message in quotes
`push`	send to the remote
`git log`	show list of commit history
`git branch`	list branches in the repo
`git branch new_name`	create a `new_name` branch
`git checkout -b new_Name`	create a `new_name` branch and switch to it
`git pull`	apply or fetch and apply changes from a remote branch to a local branch
`git commit -a -m 'msg'`	the `-a` option adds modified files (but not untracked)

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

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 of the repository name for each of your assignments in class.

File structure

I recommend the following organization structure for the course:

```
CSC3392
  |- kwl-
  |- gh-inclass
  |- semYYYY
  |- ...
```

This is one top level folder will all materials in it. A folder inside that for in class notes, and one folder per repository.

Please **do not** include all of your notes or your other assignments all inside your portflio, 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 acount. 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
What is a CPU, and What Does It Do?	1	Article	Easy to read article that explains CPUs and their use. Also touches on “buses” and GPUs.
Processors Explained for Beginners	1	Video	Video that explains what CPUs are and how they work and are assembled.
The Central Processing Unit	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` (cariage 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\`