About this Site

Contents

Skip to main content

- 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

Welcome to the course website for Computer Systems and Programming Tools in Fall2023 with Professor Brown.

This class meets TuTh 12:30-1:45 in library 166

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
```

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.

Skip to main content

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 commit history page.

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

> **Should you download the syllabus and rely on your offline copy?**
>
> No, because the syallabus 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**

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.

Skip to main content

> **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
- A bash shell
- A web browser compatible with Jupyter Notebooks
- 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**

- Install python via Anaconda video install
- 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

Learning Outcomes

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. So, the first thing to keep in mind, always is the course learning outcomes:

By the end of the semester, students will be able to:

1. Apply common design patterns and abstractions to understand new code bases, programming tools, and components of systems.
2. Apply appropriate programming workflows using context-relevant tools that enable adherance to best practices for effective code, developer time efficiency, and collaboration.
3. Differentiate the different classes of tools used in computer science in terms of their features, roles, and how they interact and justify positions and preferences among popular tools
4. Identify how information flows across levels of abstraction.
5. Discuss implications of design choices across levels of abstraction
6. Describe the social context in which essential components of computing systems were developed and explain the impact of that context on the systems.
7. Differentiate between social conventions and technical requirements in programming contexts.

These are what I will be looking for evidence of to say that you met those or not.

Principles of Grading

Learning happens through practice and feedback. My goal as a teacher is for you to learn. The grading in this course is designed to reflect how deeply you learn the material, even if it takes you multiple attempts to truly understand a topic. The topics in this course are all topics that will come back in later courses in the Computer Science major, so it is important that you understand each of them *correctly* so that it helps in the next course.

This course is designed to encourage you to work steadily at learning the material and demonstrating your new knowledge. There are no single points of failure, where you lose points that cannot be recovered. Also, you cannot cram anything one time and then forget it. The material will build and you have to demonstrate that you retained material. You will be required to demonstrate understanding of the connections between ides from different parts of the course.

- Earning a C in this class means you have a general understanding; you will know what all the terms mean; you could follow along in a meeting where others were discussing systems concepts and use core tools for common tasks. You know where to start when looking things up.
- Earning a B means that you can apply the course concepts in other programming environments; you can solve basic common errors without looking much up.
- Earning an A means that you can use knowledge from this course to debug tricky scenarios; you can know where to start and can form good hypotheses about why uncommon errors have occurred; you can confidently figure out new complex systems.

The course is designed for you to *succeed* at a level of your choice. As you accumulate knowledge, the grading in this course is designed to be cumulative instead of based on deducting points and averaging. No matter what level of work you choose to engage in, you will be expected to revise work until it is correct. The material in this course will all come back in other 300 and 400 level CSC courses, so it is essential that you do not leave this course with misconceptions, as they will make it harder for you to learn related material later.

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

Since learning developer tools is a core learning outcome of the course, we will also use them for all aspects of administering the course. This will help you learn these tools really well and create accountability for getting enough practice with core operations, but it also creates a high stakes situation: even submitting your work requires you understanding the tools. This would not be very fair at the beginning of the semester.

For the first three weeks we will have a low stakes penalty-free zone where we will provide extra help and reminders for how to get feedback on your work. In this period, deadlines are more flexible as well. If work is submitted incorrectly, we will still see it because we will manually go look for all activities. After this zone, we will assume you *chose* to skip something if we do not see it.

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.

During the third week, you will create a course plan where you establish your goals for the course and I make sure that you all understand the requirements to complete your goals.

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. We will also give you an activity that helps us to be sure that you understand it at that time.

Learning Badges

Your grade will be based on you choosing to work with the material at different levels and participating in the class community in different ways. Each of these represents different types of badges that you can earn as you accumulate evidence of your learning and engagment.

- experience: guided in class acitivies
- review: just the basics
- practice: a little bit more indepdendent
- explore: posing your own directions of inquiry
- build: in depth- application of course topics

Skip to main content

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.

To earn a D you must complete:

- 22 experience badges
- 13 lab check outs

To earn a C you must complete:

- 22 experience badges
- 13 lab check outs
- 18 review badges

To earn a B you must complete:

- 22 experience badges
- 13 lab check outs
- your choice:
  - 18 practice badges
  - 12 review + 12 practice

For an A you must complete:

- 22 experience badges
- 13 lab check outs
- your choice:
  - 18 practice badges + 6 explore badges
  - 18 review badges + 3 build badges
  - 6 review badges + 12 practice badges + 4 explore badges + 1 build badges
  - 12 review badges + 6 practice badges+ 2 explore badges + 2 build badges

You can also mix and match to get +/-. For example (all examples below assume 22+ experience badges aand 13 lab checkouts)

- A-: 18 practice + 4 explore
- B+: 6 review + 12 practice + 4 explore
- B-: 6 review + 12 practice
- B+: 24 practice
- C+: 12 review + 6 practice

> ⚠ **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.

Skip to main content

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 or `:idk:``)
- 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

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.

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.

Skip to main content

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

Community badges are awarded for extra community participation. Both programming and learning are most effective in good healthy collaboration. Since being a good member of our class community helps you learn (and helps others learn better), some collaboration is required in other badges. Some dimensions of community participation can only be done once, for example fixing a typo on the course website, so while it's valuable, all students cannot contribute to the course community in the same way. To reward these unique contributions, you can earn a community badge.

You can see some ideas as they arise by issues labeled `community`.

Community badges can replace missed experience, review, and practice badges, upgrade a review to a practice badge, or they can be used as an alternate way to earn a + modifier on a D,C,or B (URI doesn't award A+s, sorry). Community badges are smaller, so they are not 1:1 replacements for other badges. You can earn a maximum of 14 community badges, generally one per week. Extra helpful contributions may be awarded 2 community badges, but that does not increase your limit. When you earn them, you can plan how you will use it, but they will only be officially applied to your grade at the end of the semester. They will automatically be applied in the way that gives you the maximum benefit.

Community Badge values:

- 3 community = 1 experience badge
- 4 community = 1 review
- 7 community = 1 practice.
- 3 community badges + 1 review = 1 practice.
- 10 community = add a `+` to a D,C, or B, **note that this is more efficient.**

You can earn community badges by:

- fixing small issues on the course website (during penalty free zone only)
- contributing extra terms or reviews to your team repo
- sharing articles and discussing them in the course discussions
- contributing annotated resources the course website

You will maintain a list of your contributions in your KWL repo in the `community_contributions.md` file. Every individual change to this file (representing one contribution) should be commited to a new branch and then submitted as a PR, with a review requested from @brownsarahm.

> **Note**
>
> Some participation in your group repo and a small number of discussions will be required for experience, review, and practice badges. This means that not every single contribution or peer review to your team repo will earn a community badge.

Example(nonexhaustive) uses:

- 22 experience + 17 review + 11 community = C (replace 2 experience, 1 review)
- 24 experience + 17 review + 5 community = C (replace 1 review)
- 24 experience + 18 review + 10 community = C+ (modifier)

Skip to main content

- 24 experience + 16 practice + 2 review + 10 community = B (upgrade 2 review)
- 24 experience + 10 review + 10 community + 6 practice + 3 explore + 2 build = A (replace 2 review)
- 24 experience + 14 review + 10 community + 4 practice + 3 explore + 2 build = A (upgrade 2 review to practice)
- 24 experience + 12 review + 14 community + 4 practice + 3 build =A (replace 2 practice)

These show that community badges can save you work at the end of the semester by reducing the number of practice badges or simplifying badges

🎁 Free corrections

All work must be correct and complete to earn credit. In general, this means that when your work is not correct, we will give you guiding questions and advice so that you can revise the work to be correct. Most of the time asking you questions is the best way to help you learn, but sometimes, especially for small things, showing you a correct example is the best way to help you learn.

Additionally, on rare occasions, a student can submit work that is incorrect or will have down-the-line consquences but does not demonstrate a misunderstanding. For example, in an experience badge, putting text below the `#` line instead of replacing the hint within the `< >`. Later, we will do things within the kwl repo that will rely on the title line being filled in, but it's not a big revision where the student needs to rethink about what they submitted.

In these special occasions, good effort that is not technically correct may be rewarded with a 🎁. In this case, the instructor or TA will give a suggestion, with the 🎁 emoji in the comment and leave a review as "comment" instead of "changes requested" or "approved". If the student commits the suggestion to acknowledge that they read it, the instructor will then leave an approving review. Free corrections are only available when revisions are otherwise eligible. This means that they cannot extend a deadline and they are not available on the final grading that occurs after our scheduled "exam time".

Important

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

This means that, for example, the same mistake the first time, might get a 🎁, a second will probably be a hint, and a third or fourth time might be a regular revision where we ask you to go review prior assignments to figure out what you need to fix with a broad hint instead of the specific suggestion

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**

Deadlines

There will be fixed feedback hours each week, if your work is submitted by the start of that time it will get feedback. If not, it will go to the next feedback hours.

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.

Skip to main content

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
- planned: 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 5stages:

- proposed: issue created
- in progress: issue is labeled "approved" by the instructor

- 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.

Build Badges

You may earn at most one build badge per month, with final grading in December. To earn three build badges, you must earn the first one by the end of October.

Ungrading Option

At the end of the semester, you have the option of submitting a final reflection that states what grade you think you deserve, and justifies it by summarizing what you have learned and providing evidence of that. Instructions for this option will be provided as we approach the end of the semester. The policy of no submitted content that was not generated by you still applies. If you take this option, you may be required to also take an oral exam by appointment to supplement the evidence provided in your reflection.

This option exists in recognition of the fact that grading schemes are not perfect and I am truly committed to your learning. If you think that the grading scheme described on this page is working out to you earning a different grade than you deserve and you can support that with strong evidence that you have learned, you can have the grade you deserve.

What do you think?

share your thoughts on this option in the discussions for the class and then

Academic Honesty Violation Penalty

All of your work must reflect your own thinking and understanding. The work that you submit must all 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.

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, 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 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 prepare badge, the review and prepare 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 to 2/3 of the maximum possible.

This page includes more visual versions of the information on the badge 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.

Skip to main content

```
%matplotlib inline
import os
from datetime import date,timedelta
import calendar
import pandas as pd
import numpy as np
import seaborn as sns
from myst_nb import glue
# style note: when I wrote this code, it was not all one cell. I merged the cells
# for display on the course website, since Python is not the main outcome of this course

# semester settings
first_day = date(2023,9,5)
last_day = date(2023,12,12)

no_class_ranges = [(date(2023,11,23),date(2023,11,26)),
                   (date(2023,11,13)),
                  (date(2023,10,10))]


meeting_days =[1,3] # datetime has 0=Monday
spring_break = (date(2023,3,11),date(2023,3,19))
penalty_free_end = date(2023, 9, 28)


def day_off(cur_date,skip_range_list):
    '''
    is the current date a day off?

    Parameters
    ----------
    cur_date : datetime.date
        date to check
    skip_range_list : list of datetime.date objects or 2-tuples of datetime.date
        dates where there is no class, either single dates or ranges specified by a tuple

    Returns
    -------
    day_is_off : bool
        True if the day is off, False if the day has class
    '''
    # default to not a day off
    day_is_off=False
    #
    for skip_range in skip_range_list:
        if type(skip_range) == tuple:
            # if any of the conditions are true that increments and it will never go down, flase=0, true=1
            day_is_off +=  skip_range[0]<=cur_date<=skip_range[1]
        else:
            day_is_off += skip_range == cur_date
    #
    return day_is_off


# enumerate weeks
meeting_days =[1,3] # spring
mtg_delta = timedelta(meeting_days[1]-meeting_days[0])
week_delta = timedelta(7)

during_sb = lambda d: spring_break[0]<d<spring_break[1]

possible = [(first_day+week_delta*w, first_day+mtg_delta+week_delta*w) for w in range(weeks)]
weekly_meetings = [[c1,c2] for c1,c2 in possible if not(day_off(c1,no_class_ranges))]
meetings = [m for w in weekly_meetings for m in w]
meetings_string = [m.isoformat() for m in meetings]
weekly_meetings


# possible = [(first_day+week_delta*w, first_day+mtg_delta+week_delta*w) for w in range(weeks)]
# weekly_meetings = [[c1,c2] for c1,c2 in possible if not(during_sb(c1))]
```

```
# build a table for the dates
badge_types = ['experience', 'review', 'practice']
target_cols = ['review_target','practice_target']
df_cols = badge_types + target_cols
badge_target_df = pd.DataFrame(index=meetings, data=[['future']*len(df_cols)]*len(meetings),
                               columns=df_cols).reset_index().rename(
                                  columns={'index': 'date'})
# set relative dates
today = date.today()
start_deadline = date.today() - timedelta(7)
complete_deadline = date.today() - timedelta(14)

# mark eligible experience badges
badge_target_df['experience'][badge_target_df['date'] <= today] = 'eligible'
# mark targets, cascading from most recent to oldest to not have to check < and >
badge_target_df['review_target'][badge_target_df['date'] <= today] = 'active'
badge_target_df['practice_target'][badge_target_df['date'] <= today] = 'active'
badge_target_df['review_target'][badge_target_df['date']
                         <= start_deadline] = 'started'
badge_target_df['practice_target'][badge_target_df['date']
                           <= start_deadline] = 'started'
badge_target_df['review_target'][badge_target_df['date']
                         <= complete_deadline] = 'completed'
badge_target_df['practice_target'][badge_target_df['date']
                           <= complete_deadline] = 'completed'
# mark enforced deadlines
badge_target_df['review'][badge_target_df['date'] <= today] = 'active'
badge_target_df['practice'][badge_target_df['date'] <= today] = 'active'
badge_target_df['review'][badge_target_df['date']
                         <= start_deadline] = 'started'
badge_target_df['practice'][badge_target_df['date']
                           <= start_deadline] = 'started'
badge_target_df['review'][badge_target_df['date']
                         <= complete_deadline] = 'completed'
badge_target_df['practice'][badge_target_df['date']
                           <= complete_deadline] = 'completed'
badge_target_df['review'][badge_target_df['date']
                         <= penalty_free_end] = 'penalty free'
badge_target_df['practice'][badge_target_df['date']
                         <= penalty_free_end] = 'penalty free'

# convert to numbers and set dates as index for heatmap compatibility
status_numbers_hm = {status:i+1 for i,status in enumerate(['future','eligible','active','penalty free','star
badge_target_df_hm = badge_target_df.replace(status_numbers_hm).set_index('date')

# set column names to shorter ones to fit better
badge_target_df_hm = badge_target_df_hm.rename(columns={'review':'review(e)','practice':'practice(e)',
                                                        'review_target':'review(t)','practice_target':'practi
# build a custom color bar
n_statuses = len(status_numbers_hm.keys())
manual_palette = [sns.color_palette("pastel", 10)[7],
                  sns.color_palette("colorblind", 10)[2],
                  sns.color_palette("muted", 10)[2],
                  sns.color_palette("colorblind", 10)[9],
                  sns.color_palette("colorblind", 10)[8],
                  sns.color_palette("colorblind", 10)[3]]
# generate the figure, with the colorbar and spacing
ax = sns.heatmap(badge_target_df_hm,cmap=manual_palette,linewidths=1)
# mote titles from bottom tot op
ax.xaxis.tick_top()
# pull the colorbar object for handling
colorbar = ax.collections[0].colorbar
# fix the location fo the labels on the colorbar
r = colorbar.vmax - colorbar.vmin
colorbar.set_ticks([colorbar.vmin + r / n_statuses * (0.5 + i) for i in range(n_statuses)])
colorbar.set_ticklabels(list(status_numbers_hm.keys()))
# add a title
today_string = today.isoformat()
glue('today',today_string,display=False)
glue('today_notdisplayed',"not today",display=False)
ax.set_title('Badge Status as of '+ today_string);
```

```
---------------------------------------------------------------------------
NameError                                 Traceback (most recent call last)
Cell In[1], line 62
     58 week_delta = timedelta(7)
     60 during_sb = lambda d: spring_break[0]<d<spring_break[1]
---> 62 possible = [(first_day+week_delta*w, first_day+mtg_delta+week_delta*w) for w in range(weeks)]
     63 weekly_meetings = [[c1,c2] for c1,c2 in possible if not(day_off(c1,no_class_ranges))]
     64 meetings = [m for w in weekly_meetings for m in w]

NameError: name 'weeks' is not defined
```

Prepare work and Experience Badges Process

> **Warning**
>
> This was changed substantively on 2023-09-08

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)

sequenceDiagram participant P as prepare Sep 12 participant E as experience Sep 12 participant M as main note over P: complete prepare work
 between feb Sep 7 and Sep12 note over E: run experience badge workflow
 at the end of class Sep12 P ->> E: local merge or PR you that
 does not need approval note over E: fill in experience reflection critical Badge review by instructor or TA E ->> M: Experience badge PR option if edits requested note over E: make requested edits option when approved note over M: merge badge PR end
In the end the commit sequence for this will look like the following:

gitGraph commit commit checkout main branch prepare-2023-09-12 checkout prepare-2023-09-12 commit id: "gitunderstanding.md" branch experience-2023-09-12 checkout experience-2023-09-12 commit id: "initexp" merge prepare-2023-09-12 commit id: "fillinexp" commit id: "revisions" tag:"approved" checkout main merge experience-2023-09-12
Where the “approved” tag represents and approving reivew on the PR.

Review and Practice Badge

Legend:

flowchart TD badgestatus[[Badge Status]] passive[/ something that has to occur
 not done by student /] student[Something for you to do] style badgestatus fill:#2cf decisionnode{Decision/if} sta[action a] stb[action b] decisionnode --> |condition a|sta decisionnode --> |condition b|stb subgraph phase[Phase] st[step in phase] end
This is the general process for review and practice badges

flowchart TD %% subgraph work[Steps to complete] subgraph posting[Dr Brown will post the Badge] direction TB write[/Dr Brown finalizes tasks after class/] post[/Dr. Brown pushes to github/] link[/notes are posted with badge steps/] posted[[Posted: on badge date]] write -->post post -->link post --o posted end subgraph planning[Plan the badge] direction TB create[/Dr Brown runs your workflow/] decide{Do you need this badge?} close[close the issue] branch[create branch] planned[[Planned: on badge date]] create -->decide decide -->|no| close decide -->|yes| branch create --o planned end subgraph work[Work on the badge] direction TB

Skip to main content

within two weeks
 of posted date]] wait[/wait for feedback/] start --> commit commit -->moretasks commit --o started moretasks -->ccommit ccommit -->reqreview reqreview --> wait reqreview --o completed end subgraph review[Revise your completed badges] direction TB prreview[Read review feedback] approvedq{what type of review} merge[Merge the PR] edit[complete requested edits] earned[[Earned
 due by final grading]] discuss[reply to comments] prreview -->approvedq approvedq -->|changes requested|edit edit -->|last date to edit: May 1| prreview approvedq -->|comment|discuss discuss -->prreview approvedq -->|approved|merge merge --o earned end posting ==> planning planning ==> work work ==> review %% styling style earned fill:#2cf style completed fill:#2cf style started fill:#2cf style posted fill:#2cf style planned fill:#2cf

Explore Badges

flowchart TD subgraph proposal[Propose the Topic and Product] issue[create an issue] proposed[[Proposed]] reqproposalreview[Assign it to Dr. Brown] waitp[/wait for feedback/] proceedcheck{Did Dr. Brown apply a proceed label?} branch[start a branch] progress[[In Progress]] iterate[reply to comments and revise] issue --> reqproposalreview reqproposalreview --> waitp reqproposalreview --> proposed waitp --> proceedcheck proceedcheck -->|no| iterate proceedcheck -->|yes| branch branch --> progress iterate -->waitp end subgraph work[Work on the badge] direction TB moretasks[complete the work] ccommit[commit work to the branch] reqreview[request a review] wait[/wait for feedback/] complete[[Complete]] moretasks -->ccommit ccommit -->reqreview reqreview --o complete reqreview --> wait end subgraph review[Revise your work] direction TB prreview[Read review feedback] approvedq{what type of review} revision[[In revision]] merge[Merge the PR] edit[complete requested edits] earned[[Earned
 due by final grading]] prreview -->approvedq approvedq -->|changes requested|edit edit --> prreview edit --o revision approvedq -->|approved| merge merge --o earned end proposal ==> work work ==> review %% styling style proposed fill:#2cf style progress fill:#2cf style complete fill:#2cf style revision fill:#2cf style earned fill:#2cf

Build Badges

flowchart TD subgraph proposal[Propose the Topic and Product] issue[create an issue] proposed[[Proposed]] reqproposalreview[Assign it] waitp[/wait for feedback/] proceedcheck{Did Dr. Brown apply a proceed label?} branch[start a branch] progress[[In Progress]] iterate[reply to comments and revise] issue --> reqproposalreview reqproposalreview --> waitp reqproposalreview --> proposed waitp --> proceedcheck proceedcheck -->|no| iterate proceedcheck -->|yes| branch branch --> progress iterate -->waitp end subgraph work[Work on the badge] direction TB commit[commit work to the branch] moretasks[complete the work] draftpr[Open a draft PR and
 request a review] ccommit[incorporate feedback] reqreview[request a review] wait[/wait for feedback/] complete[[Complete]] commit -->moretasks commit -->draftpr draftpr -->ccommit moretasks -->reqreview ccommit -->reqreview reqreview --> complete reqreview --> wait end subgraph review[Revise your work] direction TB prreview[Read review feedback] approvedq{what type of review} revision[[In revision]] merge[Merge the PR] edit[complete requested edits] earned[[Earned
 due by final grading]] prreview -->approvedq approvedq -->|changes requested|edit edit --> prreview edit -->revision approvedq -->|approved| merge merge --o earned end proposal ==> work work ==> review %% styling style proposed fill:#2cf style progress fill:#2cf style complete fill:#2cf style revision fill:#2cf style earned fill:#2cf

Community Badges

These are the instructions from your `community_contributions.md` file: 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

Skip to main content

Important

You want one contribution per PR for tracking

flowchart TD contribute[Make a contribution
 typically not in your KWL] link[Add a link to your
 contribution to your
 communiyt_contribution.md
 in your KWL repo] pr[create a PR for that link] rev[request a review
 from @brownsarahm] contribute --> link link --> pr --> rev rev --> approved[Dr. Brown approves] --> merge[Merge the PR] merge --o earned

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.

▶ Show code cell source

Grade cutoffs for total influence are:

▶ Show code cell source

letter	threshold
D	44
D+	62
C-	80
C	98
C+	116
B-	134
B	152
B+	170
A-	188
A	206

The total influence of each badge is as follows:

▶ Show code cell source

	badge_type	badge	complexity	weight	influence
0	learning	experience	2.0	1.000000	2.000000
1	learning	review	3.0	1.000000	3.000000
2	learning	practice	6.0	1.000000	6.000000
3	learning	explore	9.0	1.000000	9.000000
4	learning	build	36.0	1.000000	36.000000
0	community	plus	1.0	1.800000	1.800000
1	community	experience_makeup	1.0	0.666667	0.666667
2	community	review_makeup	1.0	0.750000	0.750000
3	community	review_upgrade	1.0	1.000000	1.000000
4	community	practice_makeup	1.0	0.857143	0.857143

▶ Show code cell source

▶ Show code cell source

Important

the labels on the horizontal axis are just example names, they do not have any meaning, I just have not figured out what I want to replace them with that might have meaning and need some sort of unique identifier there for the plot to work.

Warning

Officially what is on the Grading page is what applies if this page is in conflict with that.

The total influence of a badge on your grade is the product of the badge's weight and its complexity. All learning badges have a weight of 1, but have varying complexity. All community badges have a complexity of 1, but the weight of a community badge can vary depending on what learning badges you earn.

There are also some hard thresholds on learning badges. You must have:

- 22 experience badges to earn a D or above
- at least 18 additional badges total across review and practice to earn above C or above

Only community badges can make exceptions to these thresholds. So if you are missing learning badges required to get to a threshold, your community badges will fill in for those. If you meet all of the thresholds, the community badges will be applied with more weight to give you a step up (eg C to C+ or B+ to A-).

Community badges have the most weight if you are on track for a grade between D and B+

If you are on track for an A, community badges can be used to fill in for learning badges, so 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.

More precisely the order of application for community badges:

- to make up missing experience badges
- to make up for missing review or practice badge to reach a total of 18 between these two types
- to upgrade review to practice to meet a threshold
- to give a step up (highest weight)

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

Skip to main content

We'll spend the first two classes 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

Tentative Schedule

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()
```

Skip to main content

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
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...

Skip to main content

date	question	keyword	conceptual	practical	social	activity
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()
```

Skip to main content

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

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.

Skip to main content

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 URI Policies

Anti-Bias Statement:

We respect the rights and dignity of each individual and group. We reject prejudice and intolerance, and we work to understand differences. We believe that equity and inclusion are critical components for campus community members to thrive. If you are a target or a witness of a bias incident, you are encouraged to submit a report to the URI Bias 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

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

- Using material, directly or paraphrasing, from published sources (print or electronic) without appropriate citation
- Claiming disproportionate credit for work not done independently
- Unauthorized possession or access to exams
- Unauthorized communication during exams
- Unauthorized use of another's work or preparing work for another student
- Taking an exam for another student
- Altering or attempting to alter grades
- The use of notes or electronic devices to gain an unauthorized advantage during exams
- Fabricating or falsifying facts, data or references
- Facilitating or aiding another's academic dishonesty
- Submitting the same paper for more than one course without prior approval from the instructors

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, you do not need to notify me in advance. You can follow the makeup procedures

Excused Absences

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

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.

Office Hours & Communication

> **Warning**
>
> Due to Indigenous People's Day:
>
> - Amoy's Tuesday office hours on 10/10 are cancelled
> - Dr. Brown's Monday office hours for 10/9 are moved to Tues 10/10

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 few classes they will be made by BrightSpace too, but that will stop

> **Sign up to watch**
>
> Watch the repo and then create a file called `community.md` in your kwl repo and add a link to this section, like:
>
> ```
> - [watched the repo as per announcements](https://introcompsys.github.io/spring2023/syllabus/communi
> ```

Help Hours

Day	Time	Location	Host
Monday	11-1	Zoom	Marcin
Monday	4-5	Zoom	Dr. Brown
Tuesday	11:15-12:30	Tyler 140	Amoy
Wednesday	1-2	Zoom	Marcin
Thursday	11:15-12:30	Tyler 140	Amoy
Friday	10-11	Zoom	Marcin
Friday	3:45-5	Tyler 140	Amoy
Friday	4-5	Tyler 134	Dr. Brown (most weeks)

Online office hours locations are linked on the GitHub Organization Page

Important

You can only see them if you are a “member” to join, use the “Whole Class Discussion” link in prismia.

Tips

For assignment help

- **send in advance, leave time for a response** 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.

Using issues

- use issues for content directly related to assignments. If you push your code to the repository and then open an issue, I can see your code and your question at the same time and download it to run it if I need to debug it
- use issues 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)

...

You can submit a pull request for the typo above, but be sure to check the pull request tab of the repo before submitting to see if it has already been submitted.

For E-mail

Skip to main content

- Please include `[CSC392]` 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

1.1. Introductions

- Dr. Sarah Brown
- Please address me as Dr. Brown or Professor Brown,

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

I look forward to getting to know you all better.

1.2. 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, not impact your final grade
- this helps both me and you know how you are doing

1.2.1. Some Background

- What programming environments do you have?
- What programming environments are you most comfortable with?
- what programming tools are you familiar with?

This information will help me prepare

1.3. This course will be different

- no Brightspace

Skip to main content

- High expectations, with a lot of flexibility to work in a way that works for you

1.3.1. 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

1.3.2. 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

1.4. 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

1.5. 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 can not add value for a company

learning deeply means you can actually use them effectively and add value beyond an AI

1.6. 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

1.7. "I never use what I learned in college"

- very common saying
- it's actually a sign of deep learning
- when we have expertise, we do not even notice when we apply it

Skip to main content

Note

We may not think that we use the fact that we know the order of letters in the English language very often. Most of us learned the alphabet with a song, but we do not sing that on a dialy basis.

However, we do fill out forms where we have to, for example, find the state we live in, in a dropdown and knowing the alphabetical order of the states helps us find ours faster.

When you know things really well, you apply them without noticing.

1.8. How does this work?

1.8.1. In class:

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

1.8.2. Outside of class:

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

1.9. Getting started

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

Accept the assignment to create your repo

1. click on .gihub, then workflows, then track.yml
2. click the 3 dots menu and select delete
3. commit directly to main with the default message
4. back to the main code tab
5. Click the pencil to edit the readme
6. Add your name
7. Add a descriptive commit message
8. Choose create a branch and open a pull request

10. Click on the list of commits, now you have one more!

1.10. What is this course about?

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

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

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

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

1.10.1. How it fits into your CS degree

1.11. In your degree

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

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

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

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

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

1.12. 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.

1.12.1. Programming is Collaborative

There are two very common types of collaboration

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

We are going to build your skill in the *code review* model. This means you need to collaborate, but collaboration in school tends to be more stressful than it needs to. If students have different goals or motivation levels it can create conflict. So **you will have no group graded work** but you will get the chance to work on something together in a low stakes way.

You will have a "home team" that you work with throughout the semester to build a glossary and a "cookbook" of systems recipes.

Skip to main content

Your contributions and your **peer reviews** will be assessed individually for your grade, but you need a team to be able to practice these collaborative aspects.

> **Important**
>
> Remember to fill out the team formation survey

1.12.2. Class forum

This community repository "assignment" will add you to a "team" with the whole class. It allows us to share things on GitHub, for the whole class, but not the whole internet.

> **Important**
>
> When you click that link join the existing team, do not make a new one

1.12.3. Get Credit for Today's class

1. Run your Experience Reflection (inclass) action on your kwl repo
2. Complete the file.

1.13. Prepare for next class/lab

1. review notes after they are posted, both rendered and the raw markdown include links to each in your badge PR
2. 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

1.14. Review

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart (on a branch for this badge).
3. review git and github vocabulary (include link in your badge PR)
4. Post an introduction to your classmates on our discussion forum

1.15. Practice

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart (on a branch for this badge).
3. review git and github vocabulary be sure to edit a file and make an issue or PR (include link in your badge PR)
4. Post an introduction to your classmates on our discussion forum

Skip to main content

2.1. What does it mean to study Computer Systems?

"Systems" in computing often refers to all the parts that help make the "more exciting" algorithmic parts work. Systems is like the magic that helps you get things done in practice, so that you can shift your attention elsewhere. In intro courses, we typically give you an environment to hide all the problems that could occur at the systems level.

Most of us have had a bug, where we found a solution to get by, without really understanding **why** the solution fixed it or why that bug happened.

Debugging often requires understanding, in practice, of how the programming language works, how it translates that to hardware, and how the hardware works.

the first "bug" was an **actual** moth

These programmers had to know how to take apart the physical computer in order to find the insect.

our computers are a lot different, but we still need systems understanding to be efficient.

> **Important**
>
> In this course, we will take the time to understand all of this stuff. This means that we will use a different set of strategies to study it than we normally see in computer science.

This is a 300 level course, so you are one step closer to being a *professional* instead of **only** a *student*.

This is still a college course, so we will be taking time to understand the theory and the *why* not only the *what* that a bootcamp or on the job training might provide.

However, instead of using a text book that is designed explicitly and primarily for a school context, we will use **primary sources**.

In our context, that means using three main types of sources:

- official reference docs
- direct research results
- first hand accounts by professional developers

Back to what a system is …

From ACM Transactions on Computer Systems

> ACM Transactions on Computer Systems (TOCS) presents research and development results on the design, specification, realization, behavior, and use of computer systems. The term "computer systems" is interpreted broadly and includes systems architectures, operating systems, distributed systems, and computer networks. Articles that appear in TOCS will tend either to present new techniques and concepts or to report on experiences and experiments with actual systems. Insights useful to system designers, builders, and users will be emphasized.

We are going to be studying aspects *of* computer systems, but to really understand them, we also have to think about why they are the way they are. We will therefore study in a broad way.

Skip to main content

2.2. Mental Models and Learning

- When we know something well, it is easier to do, we can do it multple ways,
- it is easy to explain to others and we can explain it multiple ways.
- we can do the task almost automatically and combine and create things in new ways.

This is true for all sorts of things.

> **Important**
>
> We will practice and reinforce things a lot

a mental model is how you think about a concept and your way of relating it.

Novices have sparse mental models, experts have connected mental models.

2.3. Why do we need this for computer systems?

2.3.1. Systems are designed by programmers

> **Warning**
>
> this section is a little different than what I said in class, but it is still important and related.

Computer Science is not a natural science like biology or physics where we try to understand some aspect of the world that we live in. Computer Science as a discipline, like algorithms, mostly derives from Math.

So, when we study computer science, while parts of it are limited by physics[1], most of it is essentially an imaginary world that is made by people. Understanding how people think, both generally, and common patterns within the community of programmers[2] understand how things work and why they are the way they are. The why can also make it easier to remember, or, it can help you know what things you can find alternatives for, or even where you might invent a whole new thing that is better in some way.

Historically
were often
creating a
making a r
creating a
places, CS
or out of E
grew out o

2.3.2. Context Matters

This context of how things were developed can influence how we understand it. We will also talk about the history of computing as we go through different topics in class so that we can build that context up.

2.3.3. Optimal is relative

The “best” way to do something is always relative to the context. “Best” is a vague term. It could be most computationally efficient theoretically, fastest to run on a particular type of hardware, or easiest for another programmer to read.

We will see how the best choice varies a lot as we investigate things at different levels of abstraction.

For finding and reading this section, add a link to the heading above `Systems are designed by programmers` to your

Skip to main content

a review from Dr Brown (`@brownsarahm`)

this will count for one community badge!

2.4. Let's get organized

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

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`

The first command we will use is `pwd` which stands for **p**rint **w**orking **d**irectory.

```
pwd
```

```
/Users/brownsarahm
```

this is called the path and specifically this is an absolute path

We can change into another directory with `cd` for **c**hange **d**irectory

```
cd Documents/
```

To see what changed, we use `pwd` again

```
pwd
```

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

Note that the current path is the same as the old one plus the place we changed to.

I moved one step further into my inclass folder

```
cd inclass/
```

We can **ma**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)

Skip to main content

```
ls
```

I have a few other folders here

```
fa22            prog4dsfa23     sp23            systems
```

And again check the path

```
pwd
```

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

then I can do into the new folder I just made

```
cd systems/
```

and look at the path one more time

```
pwd
```

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

We can change back to the home directory with `~`

```
cd ~
```

and confirm:

```
pwd
```

```
/Users/brownsarahm
```

just like before

Then we can use the relative path` of where we want to go:

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

`cd` with no path also works

```
cd
```

Skip to main content

```
pwd
```

```
/Users/brownsarahm
```

it's the home directoy just like `~`

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

`..` stands for up one level,

```
cd ../
```

```
pwd
```

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

2.5. 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, GitHub.
2. Check your kwl repo before class and see if you have recieved feedback, reply or merge accordingly.
3. Make sure you have a working environment, see the list in the syllabus, including `gh` CLI if you use mac`. Use the discussions to ask for help.
4. Sign up for announcements

2.6. Review today's class

1. review notes after they are posted, both rendered and the raw markdown include links to each in your badge PR
2. 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

2.7. More Practice

1. review notes after they are posted, both rendered and the raw markdown versions. Include links to both views in your badge PR comment.
2. read Chapter 1, "Decoding your confusion while coding" in The Programmer's Brain add a file called brain.md to your kwl repo that summarizes your thoughts on the chapter and how, if at all, it changes how you think about debugging and learning to program.
3. map out your computing knowledge and add it to your kwl chart repo in a file called prior-knowledge-map.md. Use mermaid

2.8. Experience Report Evidence

In office hours, show us that you have a folder to work in for class.

2.9. Questions After Today's Class

2.9.1. I wonder why Mac and Linux have built-in terminals but Windows, arguably the most popular OS, does not have a built-in terminal?

It does ahve a built in terminal, it just uses a Windows-only shell, `bash` is the most popular shell to learn, because it is used on unix and linux systems and those are most commonly used for **developers** and code **production** environments where you may only have remote, terminal only access.

2.9.2. No real questions, maybe curious about downloading github files using commands

that is what we will do Thursday

2.9.3. I've seen this type of material before so I would want to know about from other things I've seen would be how to interact with apps (google chrome) and stuff like that if that's even possible?

one step of the action that builds the course syllabus pdf launches chromium and prints the syllabus to a pdf.

2.9.4. Will we be piping things in the command line often, or are we mostly getting an introduction to the concept?

we will be workign with the shell A LOT

2.9.5. I want to know more about what reverting does to github

In a few classes, we will get there!

2.9.6. does making directories in a shell put them in the directory automatically?

it puts them where you say, nothing explicit, is an implicit relative path to your current working directory.

2.9.7. Are there any benefits to choosing either zsh or bash? Ease of use, compatability, or anything else

`bash` is more standard `zsh` can do some things a little faster

Skip to main content

> **Note**
>
> Learning more about the differences is an option for an explore badge

2.9.8. why does cd and cd~ do the same thing?

`~` is a shortcut for home and so is nothing.

2.9.9. I want to know more about how git is connected and will be applied to github and other tools we use.

we will do this over the next few classes

2.9.10. how do I make up lectures

Read the notes and complete the experience report (makeup) action

2.9.11. Should I merge the add my name to readme pull request?

yes

2.9.12. if my lab was approved can i merge the pull request~

yes

2.9.13. in what cases does the terminal file explorer get used as opposed to just navigating the file explorer/finder?

Once you know the terminal, it becomes faster than the GUI. Also, if you need to programmatically move files, bash allows you to make scripts!

2.9.14. I think I'm still just a little confused on badges, and if they're basically this class's version of assignments, or something completely different.

they're roughly like assignments, it's how we track what work you have completed. Not *all* assigned badges are required though, no matter what grade you want to earn.

---

[1] when we are *really* close to the hardware

[2] Of course, not *all* programmers think the same way, but when people spend time together and communicate, they start to share patterns in how they think. So, while you do **not** have to think the same way as these patterns, knowing what they are will help you reading code, and understanding things.

Skip to main content

3.1. Open a Terminal

Use the same one we used in the last class

and move to the folder we made to use as a working directory for in class

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

> **Note**
>
> we did some review, see the last notes instead of recapping that here

`..` is a special file that points to a specific relative path, of one level up.

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

```
pwd
```

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

```
cd ..
```

```
pwd
```

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

we move to make the current working directory the one, one step prior in the hierarchy with `cd ..`

```
cd systems/
```

3.2. A toy repo for in class

> **Warning**
>
> I removed the link from the public notes, but you can get it in prismia

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 (will happen once in a few weeks)

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

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

3.3.2. Windows (gitbash)

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

3.3.3. MacOS X

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

3.3.4. If nothing else works

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

Then proceed to the clone step. You may need to configure an identity later with `git config`

3.3.5. Cloning a repo

```
git clone https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
```

then we get several messages back from git and GitHub (the remote)

```
Cloning into 'github-inclass-fa23-brownsarahm'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (5/5), done.
remote: Total 8 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (8/8), done.
```

Confirm it worked with:

Skip to main content

```
ls
```

```
github-inclass-fa23-brownsarahm
```

We see the new folder that matches our repo name

3.4. What is in a repo?

We can enter that folder

```
cd github-inclass-fa23-brownsarahm/
```

When we compare the local directory to GitHub

```
ls
```

```
README.md
```

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            README.md
..              .github
```

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

3.5. How do I know what git knows?

`git status` is your friend.

Let’s see how it works:

```
git satus
```

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

The most similar command is
        status
```

Skip to main content

```
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.

this tells us:

- the branch we are on (`On branch main`)
- that we have incorporated all changes downloaded from GitHub (`up to date with 'origin/main'`)
- that our working directory matches what it was after the repo's last commit (`nothing to commit, working tree clean`)

3.6. Making a branch with GitHub.

> **Note**
>
> Run the one action in the github in class repo one time to get the issues

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.

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

```
git fetch orign
```

```
fatal: 'orign' does not appear to be a git repository
fatal: Could not read from remote repository.

Please make sure you have the correct access rights
and the repository exists.
```

Since the argument after `git fetch` is the variable anme of a remote, when I splled `origin` wrong, it says that it can't find it.

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/introcompsys/github-inclass-fa23-brownsarahm
 * [new branch]      1-create-an-about-file -> origin/1-create-an-about-file
```

then we check status again

Skip to main content

```
git status
```

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

nothing to commit, working tree clean
```

Looks like nothing so far.

Next, we switch to that branch.

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

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

and verify what happened

```
git status
```

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

nothing to commit, working tree clean
```

3.7. Creating a file on the terminal

The `touch` command creates an empty file.

```
touch about.md
```

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

```
ls
```

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

```
git status
```

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

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

Now we see something new. Git tells us that there is a file in the working directory that it has not been told to track the changes in and it knows nothing.

It also tells us what we can do next. Under “Untracked files” it gives us advice for how to handle those files specifically. If we had made more than one type of change, there would be multiple subheadings each with their own suggestions.

The very last line is advice of what do to overall.

We’re going to do a bit more work first though, by adding conent to the file.

We are going to use the nano text editor to edit the file

```
nano about.md
```

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

and we can look at the contents of it.

```
cat about.md
```

```
Sarah Brown
2027
```

Now we will check again

```
git status
```

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

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

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

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.

```
git add .
```

We will use status to see what has changed

Skip to main content

```
git status
```

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

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

Imp

If you
into
to ent
appea
termi
commi
escap
type
write(

```
git commit -m 'create and complete about file closes #1'
```

```
[1-create-an-about-file 693a2b5] create and complete about file closes #1
 1 file changed, 2 insertions(+)
 create mode 100644 about.md
```

Warning

At this point you might get an error or warning about your identity. Follow what git says to either set or update your identity using `git config

```
ls
```

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

Remember, the messages that git gives you are designed to try to help you. The developers of git know it’s a complex and powerful tool and that it’s hard to remember every little bit.

We again check in with git:

```
git status
```

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

nothing to commit, working tree clean
```

Now it tells us we have changes that GitHub does not know about.

We can send them to github with `git push`

```
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), 347 bytes | 347.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
   6a12db0..693a2b5  1-create-an-about-file -> 1-create-an-about-file
```

This tells us the steps git took to send:

- counts up what is there
- compresses them
- sends them to GitHub
- moves the `2-create-an-about-file` branch on GitHub from commit `3f54148` to commit `57de0cd`
- links the local `2-create-an-about-file` branch to the GitHub `2-create-an-about-file` branch

3.8. Review today's class

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.
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.

3.9. More Practice

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.

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.

3.10. Prepare for Next Class

1. Reply below with any questions you have about using terminals so that you can bring it up in class
2. Be prepared to compare and contrast bash, shell, terminal, and git.
3. (optional) If you like to read about things before you do them, read about merge conflicts. If you prefer to see them first, read this after.

3.11. Experience Report Evidence

link to your github inclass repo

3.12. Questions After Today's Class

3.12.1. Are the things we learned in the terminal available to revisit later on?

Yes, in these notest!

3.12.2. what's the difference between git checkout and git pull, i used pull in 305 a lot

`git pull` updates the **same** branch you are on. It calls `git fetch` and then does some extra stuff. `git checkout` switches branches

3.12.3. What situations would you use the terminal over something like VScode for?

really quick edits where the time spent to open and configure VSCode does not pay off

3.12.4. Will we learn how to fix mistakes in the terminal? As in commands we did not mean to run

Yes, for the things that can be fixed, we will also see what commands cannot be undone and what safeties are in built into CLI tools.

3.12.5. Im curious about the benefits to viewing files in the terminal as opposed to quick viewing them in finder, or other things that seem to be easier outside of terminal

Viewing a file in finder when we know it is really small and only text requires **context switching** which makes it more work. Using a GUI can be an easier way to start, it is less to remember, but once you know how things work you cannot make them any faster, you have to move your mouse so far, wait for things to load, etc. The terminal lets you gain speed once you learn.

You can also automate and script things.

Finally, sometimes a GUI is a waste of resources that you do not have to spare.

4. How do git branches work?

4.1. Review

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

```
cd Documents/inclass/systems/github-inclass-fa23-brownsarahm/
```

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

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-inclass-fa23-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-an-about-file
Your branch is up to date with 'origin/1-create-an-about-file'.

nothing to commit, working tree clean
```

this is as we left off.

4.2. Branches do not sync automatically

Skip to main content

> **Warning**
>
> the step below requires that you have the `gh` CLI. On windows with GitBash, you have not needed this, and it can be bit tricky to install. If you get it done and working well, and you submit instructions to this repo about how to get it set up, for either a community badge (if it's simple) or an explore badge if it takes a lot

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

```
Opening github.com/introcompsys/github-inclass-fa23-brownsarahm in your browser.
```

On Github, we see that it matches the branch we were on because we had merged in our PR in class on Thursday.

```
git status
```

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

nothing to commit, working tree clean
```

Back on our local computer, we will go back to the main branch, using `git checkout`

```
git checkout main
```

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

now we look at the status of our repo:

```
ls
```

```
README.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
Unpacking objects: 100% (1/1), 646 bytes | 215.00 KiB/s, done.
From https://github.com/introcompsys/github-inclass-fa23-brownsarahm
   6a12db0..caeacb5  main       -> origin/main
```

Skip to main content

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 this updates th working directory too:

```
ls
```

```
README.md
```

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

We can use git pull to fully update.

```
git pull
```

```
Updating 6a12db0..caeacb5
Fast-forward
 about.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 about.md
```

```
ls
```

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

```
git status
```

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

nothing to commit, working tree clean
```

Now we will follow the instruction sin the README issue, commit to a new branch with `closes #x` where x is the issue number in the comment and make PR.

Merge the PR and then the issue closes.

4.3. Pull will downlaod and update

We will `git pull` again now all at once, this will do the work that fetch did before and the checkout.

```
remote: Enumerating objects: 6, done.
remote: Counting objects: 100% (6/6), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 1.32 KiB | 337.00 KiB/s, done.
From https://github.com/introcompsys/github-inclass-fa23-brownsarahm
   caeacb5..5c8aaa9  main       -> origin/main
 * [new branch]      add-name   -> origin/add-name
Updating caeacb5..5c8aaa9
Fast-forward
 README.md | 4 ++++
 1 file changed, 4 insertions(+)
```

We can see commits with `git log`

```
git log
```

```
commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (HEAD -> main, origin/main, origin/HEAD)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2

commit 65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:46:00 2023 -0400

    closes #2

commit caeacb503cf4776f075b848f0faff535671f2887
Merge: 6a12db0 693a2b5
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:38:54 2023 -0400

    Merge pull request #4 from introcompsys/1-create-an-about-file

    create and complete about file closes #1

commit 693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:36:03 2023 -0400

    create and complete about file closes #1

commit 6a12db0035e7c73772f7b2348b80dd0bfb3a2a2e
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Thu Sep 14 16:43:07 2023 +0000

    Add online IDE url

commit cfe32e5066921ad876d8a2c74b1fcb00c99b1cc7
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Thu Sep 14 16:43:05 2023 +0000

    Initial commit
```

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

```
git status
```

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

nothing to commit, working tree clean
```

```
git pull
```

```
Already up to date.
```

4.4. making a new branch locally

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

```
git checkout -b fun_fact
```

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

```
git log
```

Note

notice where each branch pointer is

```
commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (HEAD -> fun_fact, origin/main, origin/HEAD, main)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2

commit 65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:46:00 2023 -0400

    closes #2

commit caeacb503cf4776f075b848f0faff535671f2887
Merge: 6a12db0 693a2b5
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:38:54 2023 -0400

    Merge pull request #4 from introcompsys/1-create-an-about-file

    create and complete about file closes #1

commit 693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:36:03 2023 -0400

    create and complete about file closes #1

commit 6a12db0035e7c73772f7b2348b80dd0bfb3a2a2e
```

Skip to main content

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.

```
nano about.md
```

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.

My file looks like this.

```
Sarah Brown
2027
- i skied competitively in high school
```

```
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")
```

This is very similar to when we checked after creating the file before, but, notice a few things are different.

- the first line tells us the branch but does not compare to origin. (this branch does not have a linked branch on GitHub)
- thefile is listed as “not staged” instead of untracked
- it gives us the choice to add it to then commit OR to restore it to undo the changes

We will add it to stage

```
git add about.md
```

```
git status
```

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

and commit:

> **Warning**
>
> here i gave an example of what to do if you commit without a message. This is not recommended, but important to know how to fix.

```
git commit
```

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 compelte your commit.

```
[fun_fact 6d4dbd3] add fun fact
 1 file changed, 1 insertion(+)
```

Now we can look again at where our branch pointers are.

```
git log
```

fun_fact moved up to the new commit, but the other branches stayed behind.

```
commit 6d4dbd33860fceb9c87bd3c4509deff8cecb3f45 (HEAD -> fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:06:54 2023 -0400

    add fun fact

commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (origin/main, origin/HEAD, main)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2

commit 65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:46:00 2023 -0400

    closes #2

commit caeacb503cf4776f075b848f0faff535671f2887
Merge: 6a12db0 693a2b5
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:38:54 2023 -0400

    Merge pull request #4 from introcompsys/1-create-an-about-file

    create and complete about file closes #1

commit 693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file)
```

Skip to main content

```
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.

git stores its list of remotes in a `.git/config` file

```
cat .git/config
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "1-create-an-about-file"]
        remote = origin
        merge = refs/heads/1-create-an-about-file
```

we can see our other branches have `remote = origin` but our new branch is not there.

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), 311 bytes | 311.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 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/introcompsys/github-inclass-fa23-brownsarahm/pull/new/fun_fact
remote:
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
 * [new branch]      fun_fact -> fun_fact
branch 'fun_fact' set up to track 'origin/fun_fact'.
```

Skip to main content

This time the returned message from the remote includes the link to the page to make a PR. We are not goign to make the PR though

4.5. Merge conflicts

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
2027
- i skied competitively in high school
- I went to Northeastern
```

now that we have edited in both places let's pull.

```
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
Unpacking objects: 100% (3/3), 697 bytes | 174.00 KiB/s, done.
From https://github.com/introcompsys/github-inclass-fa23-brownsarahm
   6d4dbd3..768dec8  fun_fact   -> origin/fun_fact
Updating 6d4dbd3..768dec8
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
```

First 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")
```

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

```
[fun_fact b378bd1] 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.

gitGraph commit commit branch fun_fact checkout fun_fact commit branch origin/fun_fact checkout origin/fun_fact commit checkout fun_fact commit
the branches have diverged means that they do not agree and that they each have at least one commit that is different from the other.

```
git log
```

```
commit b378bd148e53dfa7195c58123362e40ae12ef3e7 (HEAD -> fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:26:20 2023 -0400

    second fun fact

commit 6d4dbd33860fceb9c87bd3c4509deff8cecb3f45
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:06:54 2023 -0400

    add fun fact

commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (origin/main, origin/HEAD, main)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2

commit 65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name)
Author: Sarah Brown <brownsarahm@uri.edu>
```

```
    closes #2

commit caeacb503cf4776f075b848f0faff535671f2887
Merge: 6a12db0 693a2b5
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:38:54 2023 -0400
```

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 b378bd1... 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".
Could not apply b378bd1... second fun fact
```

Now it tells us there is a conflict.

```
nano about.md
```

We see that git marked up the conflict for us by editing the file:

```
Sarah Brown
2027
- i skied competitively in high school
<<<<<<<<< HEAD
- i started at URI in 2020
=======
- I went to Northeastern
>>>>>>>>> "add fun fact"
```

We manually edit it to be what we want it to be. We can take one change the other or both. We will choose both, so my file looks like this in the end.

```
cat about.md
```

```
Sarah Brown
2027
- i skied competitively in high school
- i started at URI in 2020
- I went to Northeastern
```

Now, we do git add, because above it said that was the next step.

```
git add about.md
```

and then this was the next thing after that.

> ⚠ **Warning**
>
> If you got somethign very different at this point, run `git --version` mine is 2.41. If yours is different, let me know as an issue.

```
git rebase --continue
```

```
[detached HEAD 756c487] second fun fact
 1 file changed, 2 insertions(+)
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), 312 bytes | 312.00 KiB/s, done.
Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (2/2), completed with 2 local objects.
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
   768dec8..756c487  fun_fact -> fun_fact
```

4.6. Prepare for Next Class

1. Read the notes from 2023-09-19 carefully
2. Examine an open source project on GitHub. Answer the reflection questions below in `software.md` in your kwl repo on a branch that is linked to this issue. You do not need to make the PR, we will work with this in class on 2023-09-21.

```
## Software Reflection

1. Link and title of the project
1. What types of files are there that are not code?
1. What different types of code files are in the project? Do they serve different goals?
1. Is it all in one language or are there multiple languages?
1. Are there files that are configurations or settings?
1. Is there help for developers? users? which support is better?
1. 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

- Jupyter book

4.7. Review today's class

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.

4.8. More Practice

1. Create a merge conflict in your github in class repo 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 ide_merge_conflict.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)
2. Learn about GitHub forks and more about git branches(you can also use other resources)
3. add `branches-forks.md` to your KWL repo and describe how branches work, what a fork is, and how they relate to one another. If you use other resources, include them in your file as markdown links.

4.9. Experience Report Evidence

> **Note**
>
> this is for if you miss class and need to make it up

run `git log >> inclass2023-09-19.md` and then copy that new file into your KWL repo on the branch for your experience report

4.10. Questions After Today's Class

4.10.1. How would you fix a merge conflict involving other people trying to commit on other systems?

The same way we did in class.

4.10.2. are there situations where if you accidently make another branch, that you can merge it can into the one you wanted to be working on?

yes, exactly!

cases?

Yes, when the merge conflict arises as a result of a pull request and it is simple you can resolve it in github.com

4.10.4. What exactly does the rebase command do?

I described it in more detail above with links to git docs.

4.10.5. What point does using git locally give?

it's the only way to keep your work in git and also run code locally

4.10.6. Why were some of my command prompts different from yours?

Not sure, but it might be the version of git.

4.10.7. What website do you find the most useful to resolve issues relating to git within the terminal?

I tend to rely on using `git status` to see where it is at, thinking about where I want to get and then searching and using eg stackoverflow for things I do not know how to do.

that said git gud can help you visualize and we will have some other visualizations soon.

4.10.8. Would it be reasonable to never use the local Terminal and just always use github?

No, that means you would not be able to run your code to work. Even if you work on most clous services, you still need to use the "local" terminal, on that cloud instance.

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)

> **Further Reading**
>
> Use these for checking facts and resources.
>
> - bash
> - git

5.1. Setup

```
cd Documents/inclass/systems/github-inclass-fa23-brownsarahm/
```

and we will make sure we are in sync with GitHub. We know we *can* fix it, but it is best to not break it.

```
git status
```

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

nothing to commit, working tree clean
```

Since our working tree is clean, now we can get updates from GitHub

```
git pull
```

```
Already up to date.
```

Now let's go back to the main branch.

```
git checkout main
```

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

It tells us we are there.

Let's look at what commits are on the main branch.

```
git log
```

```
commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (HEAD -> main, origin/main, origin/HEAD)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2

commit 65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:46:00 2023 -0400

    closes #2

commit caeacb503cf4776f075b848f0faff535671f2887
Merge: 6a12db0 693a2b5
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:38:54 2023 -0400

    Merge pull request #4 from introcompsys/1-create-an-about-file
```

```
commit 693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 14 13:36:03 2023 -0400
```

We remember that we had added fun facts and none of our commits about those are here, so the fun_fact branch is still ahead of the main branch.

It's something like this:

gitGraph commit commit commit commit branch fun_fact checkout fun_fact commit branch origin/fun_fact commit checkout fun_fact commit merge origin/fun_fact

5.2. Merging a branch offline

We can merge branches locally, without making a pull request in GitHub.

```
git merge fun_fact
```

```
Updating 5c8aaa9..756c487
Fast-forward
 about.md | 4 ++++
 1 file changed, 4 insertions(+)
```

It summarizes the changes.

Now we can use `git log` to see where our branches all point.

```
git log
```

```
commit 756c4879c0447db20980f73a26bc2ba072e08a6d (HEAD -> main, origin/fun_fact, fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:26:20 2023 -0400

    second fun fact

commit 768dec80c5e0734476d476ae83376c9c786b6450
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:21:31 2023 -0400

    Update about.md

commit 6d4dbd33860fceb9c87bd3c4509deff8cecb3f45
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:06:54 2023 -0400

    add fun fact

commit 5c8aaa9f2a129d551b8cb2cb294676f63c4af410 (origin/main, origin/HEAD)
Merge: caeacb5 65e9e39
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 12:47:08 2023 -0400

    Merge pull request #5 from introcompsys/add-name

    closes #2
```

Skip to main content

Not

This
merm
often
idea,
every
(files,
can a
merm
them

Visually, it is like this

gitGraph commit id:"0" commit id:"1" commit id:"2" commit id:"3" branch fun_fact checkout fun_fact commit id:"A" branch origin/fun_fact commit id:"B" checkout fun_fact merge origin/fun_fact commit id:"C" checkout main merge fun_fact

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

It is ahead by 3 commits now because there were 3 commits on the fun_fact branch.

5.3. Organizing a project (workign 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

next we are going to pretend we worked on the project and made a bunch of files

```
touch abstract_base_class.py helper_functions.py important_classes.py alternative_classes.py README.md LICEN
```

New bash lessons:

- `touch` can accept a list of files to create
- a list in bash is space separated with no brackets

```
ls
```

```
API.md                  about.md                setup.py
CONTRIBUTING.md         abstract_base_class.py  test_alt.py
LICENSE.md              alternative_classes.py  test_help.py
README.md               helper_functions.py     test_imp.py
_config.yml             important_classes.py    tests_abc.py
_toc.yml                overview.md
```

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 3 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        API.md
        CONTRIBUTING.md
        LICENSE.md
        _config.yml
         toc.yml
```

Skip to main content

```
        helper_functions.py
        important_classes.py
        overview.md
        setup.py
        test_alt.py
        test_help.py
        test_imp.py
        tests_abc.py

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

5.4. Echo and redirects

`cat` concatenates the contents of a file to stdout, which is a special file that our terminal reads

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406b
```

`echo` allows us to send a message to stdout.

```
ehco "age=35"
```

```
-bash: ehco: command not found
```

For example, we can echo our age

```
echo "age=35"
```

```
age=35
```

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 (`>>`)

```
echo "age=35" >> README.md
```

this command has no output, but we can use `cat` to view the file to see its impact.

```
cat README.md
```

Skip to main content

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406b
age=35
```

Now we see the text at the bottom of the file.

5.5. commit and add has limitations

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

Last class we saw we can add and commit at the same time

```
git commit -a -m 'start organizing'
```

```
[main bc28179] start organizing
 1 file changed, 1 insertion(+)
```

This is way fewer things than we have done recently (remember we also made all of those files with touch)

lets check `git status` again

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 4 commits.
  (use "git push" to publish your local commits)

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        API.md
        CONTRIBUTING.md
        LICENSE.md
        _config.yml
        _toc.yml
        abstract_base_class.py
        alternative_classes.py
        helper_functions.py
        important_classes.py
        overview.md
        setup.py
        test_alt.py
        test_help.py
        test_imp.py
        tests_abc.py

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

The `-a` option on a commit does not add untracked files

We have to explicitly add those files.

```
git add
```

Now we can commit the content.

```
git commit -m 'start organizng for real'
```

```
[main d76bc52] start organizng for real
 15 files changed, 0 insertions(+), 0 deletions(-)
 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 helper_functions.py
 create mode 100644 important_classes.py
 create mode 100644 overview.md
 create mode 100644 setup.py
 create mode 100644 test_alt.py
 create mode 100644 test_help.py
 create mode 100644 test_imp.py
 create mode 100644 tests_abc.py
```

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

5.6. Redirect file modes

Now, let's go back to thinking about redirects. We saw that with two `>>` we appended to the file. With just *one* what happens?

```
echo "age=35" > README.md
```

We check the file now

```
cat README.md
```

```
age=35
```

It wrote over. One `>` writes to the file in overwrite mode.

> **Note**
>
> **mode** is a good vocab term that could be added to the site glossary for a community badge

This would be bad, we lost content, but this is what git is for!

It is *very very* easy to undo work since our last commit.

This is good for times when you have something you have an idea and you do not know if it is going to work, so you make a commit before you try it. Then you can try it out. If it doesn't work you can undo and go back to the place where you made the commit.

To do this, we will first check in with git

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

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

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

Notice that it tells us what to do `(use "git restore <file>..." to discard changes in working directory)`. The version of README.md that we broke is in the working directory but not commited to git, so git refers to them as "changes" in the workign directory.

```
git restore README.md
```

Now lets check with git.

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Our working tree is clean now.

and it looks like it did before the `>` line. and we can check the file too

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406b
age=35
```

Skip to main content

Tip

Conc

When
use fu

-
-

both
name
have
file w

-
-
-

Refer

-
-

*C is r
the ty
"offici*

Back how we wanted it!

5.8. What are these files?

We can use a redirect to add a bunch of text to the README file.

```
 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 |
> > | tests_abc.py | tests for constructors and methods in abstract_base_class.py|
> > | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> > | tests_imp.py | tests for constructors and methods in important_classes.py|
> > | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> > | API.md | jupyterbook file to generate api documentation |
> > | _config.yml | jupyterbook config for documentation |
> > | _toc.yml | jupyter book toc file for documentation |
> > | philosophy.md | overview of how the code is organized for docs |
> > | example.md | myst notebook example of using the code |
> > | scratch.ipynb | jupyter notebook from dev |" >> README.md
```

this explains each file a little bit more than the name of it does. We see there are sort of 5 groups of files:

- about the project/repository
- code that defines a python module
- test code
- documentation
- extra files that “we know” we can delete.

We can see what it does with `cat`

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406b
age=35
|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 |
> | tests_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 |
```

```
> | 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 |
```

Note

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

We can see how the `"` allows us to go on multiple lines this way:

```
echo "sldkjfds
>
>
>
>
>
>
>
>
>
> "
```

and it will output all of the blank lines

```
sldkjfds
```

5.9. Getting organized

Recall, we have just created a bunch of files.

```
ls
```

```
API.md                  about.md                setup.py
CONTRIBUTING.md         abstract_base_class.py  test_alt.py
LICENSE.md              alternative_classes.py  test_help.py
README.md               helper_functions.py     test_imp.py
_config.yml             important_classes.py    tests_abc.py
_toc.yml                overview.md
```

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

```
mkdir docs
```

we can use `ls` to see the files that exist

```
ls
```

Skip to main content

```
README.md               docs                    test_help.py
_config.yml             helper_functions.py     test_imp.py
_toc.yml                important_classes.py    tests_abc.py
```

5.10. Moving Files

next we will move a file there with `mv`

We can see how to use this by getting the error when we do not pass any arguments.

```
mv
```

```
usage: mv [-f | -i | -n] [-v] source target
       mv [-f | -i | -n] [-v] source ... directory
```

we get an error message that tells us how to use the command, it takes two arguments: source and target.

5.10.1. Help in GitBash

```
mv --help
```

```
mv: illegal option -- -
usage: mv [-f | -i | -n] [-v] source target
       mv [-f | -i | -n] [-v] source ... directory
```

This only give a basic version on mac, but is more on Git Bash

5.10.2. Getting help on *nix systems

`man` shows the manfile for help for a specific command

```
man mv
```

It opens in a program, then we need to use `q` to exit.

5.10.3. Back to moving.

```
mv overview.md docs/
```

what this does is change the path of the file from `.../github-inclass-brownsarahm-1/overview.md` to `.../github-inclass-brownsarahm-1/docs/overview.md`

This doesn't return anything, but we can see the effect with `ls`

Skip to main content

```
API.md                  about.md                setup.py
CONTRIBUTING.md         abstract_base_class.py  test_alt.py
LICENSE.md              alternative_classes.py  test_help.py
README.md               docs                    test_imp.py
_config.yml             helper_functions.py     tests_abc.py
_toc.yml                important_classes.py
```

We can also use `ls` with a relative or absolute path of a directory to list tht location instead of our current working directory.

```
ls docs/
```

```
overview.md
```

5.10.4. Moving multiple files with patterns

let's look at the list of files again.

```
ls
```

```
API.md                  about.md                setup.py
CONTRIBUTING.md         abstract_base_class.py  test_alt.py
LICENSE.md              alternative_classes.py  test_help.py
README.md               docs                    test_imp.py
_config.yml             helper_functions.py     tests_abc.py
_toc.yml                important_classes.py
```

We have lots with similar names.

We can use the `*` wildcard operator to move all files that match the pattern. We'll start with the two `yml` (yaml) files that are both for the documentation.

```
mv *.yml docs/
```

Again, we confirm it worked by seeing that they are no longer in the working directory.

```
ls
```

```
API.md                  abstract_base_class.py  setup.py
CONTRIBUTING.md         alternative_classes.py  test_alt.py
LICENSE.md              docs                    test_help.py
README.md               helper_functions.py     test_imp.py
about.md                important_classes.py    tests_abc.py
```

and that they *are* in `docs`

```
ls docs/
```

Skip to main content

Next we will work on the test files

```
mkdir tests
```

5.11. Renaming Files

We see that most of the test files start with `test_` but one starts with `tests_`. We could use the pattern `test*.py` to move them all without conflicting with the directory `tests/` but we also want consistent names.

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 tests
```

Here I pressed tab multiple times trying to get tab complete, but since it was not unique, it showed me the two options.

```
tests/        tests_abc.py
```

Then I added `_` and pressed tab again to get it full file name.

```
mv tests_abc.py test_abc.py
```

This changes the path from `.../tests_abc.py` to `.../test_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 differnt part of the path.

```
ls
```

```
API.md                  alternative_classes.py  test_alt.py
CONTRIBUTING.md         docs                    test_help.py
LICENSE.md              helper_functions.py     test_imp.py
README.md               important_classes.py    tests
about.md                setup.py
abstract_base_class.py  test_abc.py
```

Then we can use a similar pattern to move the files.

```
mv test_*.py tests/
```

Now we can confirm that it is how we expect.

```
ls
```

```
API.md                  about.md                helper_functions.py
CONTRIBUTING.md         abstract_base_class.py  important_classes.py
LICENSE.md              alternative_classes.py  setup.py
```

Skip to main content

```
ls tests/
```

```
test_abc.py     test_alt.py     test_help.py    test_imp.py
```

5.12. Commiting Changes

we check where we are

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    overview.md
        deleted:    test_alt.py
        deleted:    test_help.py
        deleted:    test_imp.py
        deleted:    tests_abc.py

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

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

```
git add docs/
```

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 5 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   docs/_config.yml
        new file:   docs/_toc.yml
        new file:   docs/overview.md

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    overview.md
        deleted:    test_alt.py
```

```
        deleted:    tests_abc.py

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

```
git add .
```

```
git commit -m 'begin organizing
>
>
> '
```

```
[main 042a42e] begin organizing
 8 files changed, 19 insertions(+)
 rename _config.yml => docs/_config.yml (100%)
 rename _toc.yml => docs/_toc.yml (100%)
 rename overview.md => docs/overview.md (100%)
 rename test_alt.py => tests/test_abc.py (100%)
 rename test_help.py => tests/test_alt.py (100%)
 rename test_imp.py => tests/test_help.py (100%)
 rename tests_abc.py => tests/test_imp.py (100%)
```

ANd finally 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% (11/11), 1.72 KiB | 1.72 MiB/s, done.
Total 11 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), done.
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
   5c8aaa9..042a42e  main -> main
```

5.13. experience badge with a file

1. Run the action to create the branch for today's experience badge
2. Open a PR to propose changes from your prepare work brach *to* the branch for today's experience badge
3. merge that PR
4. See that in your experience badge, the prepare work is now visible
5. fill in your experience badge
6. request a review from the TA that is in your group

5.14. Prepare for Next Class

1. Review your software.md file from last prepare
2. Review the notes from 2023-09-21

Skip to main content

3. 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)
4. Update your `.github/workflows/experienceinclass.yml` file as to add another paramter to the last step (`Create Pull request`) `reviewers: <ta-gh-name>` where `<ta-gh-name>` is the github user name of the TA is in your group. You can see your group on the organiation teams page named like "Fall 2023 Group X". Do this on a branch for this issue.

5.15. Review today's class

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on.

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 `fall2023/.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. What happens and what GitHub concept that we have not used yet might fix it? see your `vocab-` repo for a list of key github concepts. (answer in the `terminal_review.md`)
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 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`
5. Find your team's repository. It will have a name like `fa23-team#` where `#` is a number 1-4. Join the discussion on that repo about naming your team. Link to your comment directly in your PR for this badge (use the 3 dots menu to get the comment specific URL).

```
# Terminal File moving reflection
1. How was this activity overall? Did this get easier toward the end?
2. When do you think that using the terminal will be better than using your GUI file explorer?
3. What questions/challenges/ reflections do you have after this exercise?

## Commands used
```

5.16. More Practice

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on.

1. Update your KWL chart with any learned items.
2. Get set up so that you can pull from the introcompsyss/fall2023 repo and push to your own fork of the class website by cloning the main repo, then forking it and adding your fork as an additional remote. Append the commands used and the contents of your `fall2023/.git/config` to a terminalpractice.md (hint: `history` outputs recent commands and redirects can work with any command, not only echo). Based on what you know so far about forks and branches, what advantage does this setup provide? (answer in the `terminal_practice.md`)
3. **lab** Organize the provided messy folder (details will be provided in lab time). Commit and push the changes. Clone that repo locally.
4. For extra practice, re/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

Skip to main content

5. Find your team's repository. It will have a name like `fa23-team#` where `#` is a number 1-4. Join the discussion on that repo about naming your team. Link to your comment directly in your PR for this badge (use the 3 dots menu to get the comment specific URL).

```
# Terminal File moving reflection
1. How was this activity overall Did this get easier toward the end?
2. How was it different working on your own computer compared to the Codespace from?
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 codesp
```

5.17. Experience Report Evidence

redirect the history to a file

```
history >> makeup_2023-09-21.md
```

then move the file created to your KWL repo on your experience report branch.

5.18. Questions After Today's Class

5.18.1. How can the "*" be used consistently to move a bunch of file? because we used it with the text infront of the similarity and behind it. i'm just wondering how it's used for multiple files, because it was used both before the similarity and after the similarity.

It fills in for any number of characters.

5.18.2. I have some badges from 2 weeks ago that are awaiting a second review after changes had been fixed.

> **Important**
>
> Do not merge an unapproved badge

Re-request a review

5.18.3. can you close pull requests on the terminal?

Not with git because pull requests are not a git feature, but it is a feature of github. The `gh` CLI can do this.

5.18.4. If you do mv * will it move all files?

In the current working directory.

5.18.5. Are there any other ways of using mv that haven't been covered yet?

No those are basically the two purposes.

5.18.6. how often should we be practicing with the terminal for git?

Ideally, you work on badges on at least several of the days we do not have class so that you are working with it close to every day.

You could also start trying to use them for your other classes.

5.18.7. Would it be beneficial to organize files with github rather than bash?

GitHub cannot organize files and doing do in broswer would be slow and difficult. We will see that GitHub code spaces give us a virtual machine that we can work with.

6. What is a commit?

> **Warning**
>
> Due to events beyond my control the full details for this class were lost. Below contains all of the notes and explanation, but not every command that was run and the actual output at each step.
>
> No notes yet because I made an error and did not save my terminal output from the codespace before I left the classroom.
>
> To produce the notes, I will have to use the commit history and the prismia log to re-do the activity in a copy of my repo that I manipulate to undo things so that I can recreate.

6.1. Housekeeping

6.1.1. In class time

reminder: you are expected to follow along with what I model in my terminal and use prismia as a backup if you miss something or get an error.

6.1.2. Notifications

reminder: I post announcements as releases

To get notified, watch the repo.

6.1.3. Penalty free zone is over.

- all past badges can be submitted at any time

6.2. Why study what a commit is?

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

today we will continue organizing the github inclass repo as our working example.

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

In intro classes, we try really hard to only use one definition for each concept 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

6.3. Using a GitHub Codespace

Today we will also work in github codespaces.

1. Navigate to your github inclass repo on Github.com
2. Use the link in the README or the green code button to open a new codespace on main.

You are allowed to have two active code spaces at any point in time. If you get an error, you have some still open. Use the codespace page to see all of yours.

You can deacitvate any that are running that you do not want to use.

> **Important**
>
> Remember: codespace is a virutal machine on a cloud platform, not cloud access to github.com
>
> - you need to commit changes
> - codespace is linux

this is why i teach bash

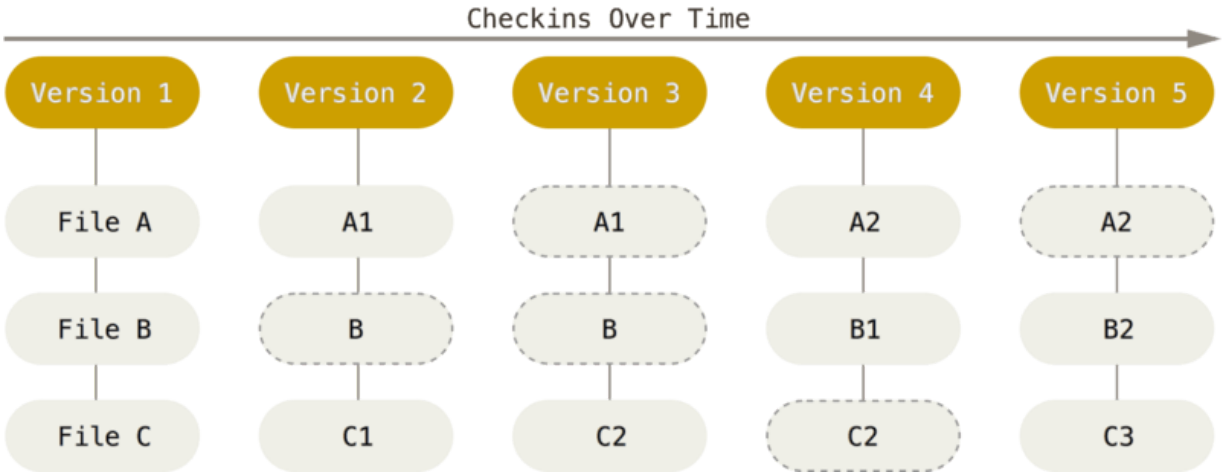
as developers, we will all interact with linux/unix at times, so bash is the best shell to know if you only know one or do not want to switch between multiple

6.4. Multiple cursors are your friend

docs

Skip to main content

6.5. Conceptually, a commit is a snapshot



6.6. A commit is the basic unit of what git manages

All other git objects are defined relative to commits

- branches are pointers to commits
- tags are pointers to commits

6.7. parts of a commit

We can use `git log` to view past commits

here we see some parts:

- hash
- (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

we can view individual commits with

`git cat-file`

Now we see more detail:

- hash (used to access)
- tree
- parent
- author with timestamp
- commiter with timestamp

6.8. Commit parents help us trace back

kind of like a linked list

6.9. Commit trees are the hash of the content

This separation is helpful.

Let's make a commit that includes changes to two files

After the commit we can use

```
git reset HEAD^
```

to put the head pointer back and reset the working directory back to how it was at that point as well.

The git reset docs describe this in great detail.

6.10. 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

6.11. Prepare for Next Class

1. Read the 2023-09-26 notes when posted
2. 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`
3. Take a few minutes to think about a time you were debugging or pay attention the next time you debug something. Make note of how you think about the problem, how do you decide what to check? how do you decide what to try? what information do you look for?

6.12. Review today's class

1. create an issue on your group repo for a tip or cheatsheet item you want to contribute. Make sure that your contribution does not overlap with one that amemb
2. clone your group repo.
3. work offline and add your contribution and then open a PR
4. review a class mate's PR.
5. 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`

6.13. More Practice

1. Find a resource/reference that helps explain a topic related to the course that you want to review. Make sure that your contribution does not overlap with one that another member is going to post by viewing other issues before you post your issue. Create an issue for your planned item.
2. Clone your group repo.
3. Work offline to add your contribution and then open a PR. Your reference review should help a classmate decide if that reference material will help them understand better or not. It should summarize the material and it's strengths/weaknesses.
4. Complete a peer review of a class mate's PR. Use inline comments for any minor corrections, provide a summary, and either approve or request changes.
5. 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.

6.14. Experience Report Evidence

write your `git log` to a file and include it with your experience report.

6.15. Questions After Today's Class

6.15.1. Can you use the terminal for any IDE that is not only VScode in Codespaces?

Yes.

6.15.2. What is the difference between using git reset and git revert?

git revert applies a new commit that is the opposite of the one that you want to "undo" reset actuall resets everything back to a particular point.

6.15.3. How can I learn more neat tricks like multicursor?! Very usefull and I do not know how I have never known about it.

I think most developers learn things like this from talking to one another or reading blogs/watching youtube/ following on social media.

We will have a class on IDEs later, where one of the tasks is for you all to share tips.

6.15.4. Is there always a way to return to a state of a repo, no matter how far along or how much as been commited or changed?

Yes

It allows a script to process the commit messages in order to produce a change log.

> **Important**
>
> Learning more about this or using it throughout could be an explore/build badge

6.15.6. Is there a way to write a script to automate this process so that I don't need to write as many commands?

Yes we will write scripts in a few weeks.

6.15.7. Is it more efficient to look for the commit id and create a branch off it in the terminal rather than to do it manually in github?

Often yes.

6.15.8. Do you know of some situations where restore saved the worst possible mistakes in a repo?

Yes, reverting and restoring helps fix bugs all the time. It is also how people can remove data that should have never been commited, before pushing.

7. How do programmers communicate about code?

Today we are going to pick up from where we left off talking about the conventional commits.

That is a core example of the types of detailed communication we do in programming that is embedded into the work.

7.1. Closing your Codespace

Use the codespace page to see all of yours.

Check that your github-inclass repo has no ucnommited changes. If it does you need to commit them, there are two strategies:

- start the codespace, commit and push
- from that page, export the changes to a branch, create a PR and merge it if needed

7.2. Build and Explore Ideas

I am going to put ideas in the discussions repo

I violated the instructions a little, but I thought it made the most sense.

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
- automation
- building

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

- describe different types of documentation
- find different information in a code repo
- generate documentation as html
- ignore content from a repo
- create a repo locally and push to GitHub

7.4. What is documentation

> **Note**
>
> We opened this page and discussed the different types of documentation in the table in class

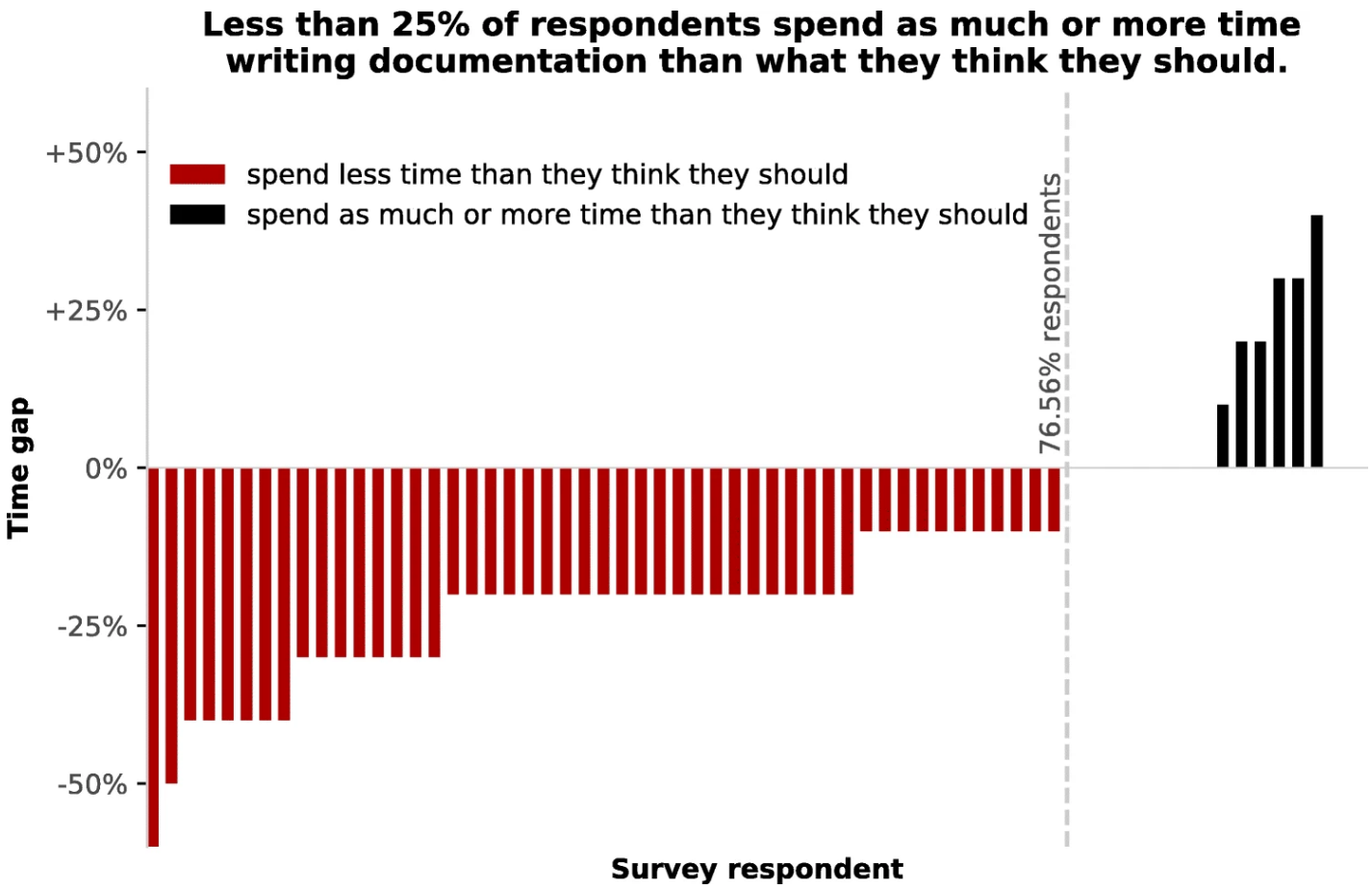
documentation types table

from ethnography of documentation data science

Data Science is only *one* type of programing, but this is still pretty representative. The main difference is that in this case the "target" users are often data scientists and more technically skilled than the "target" users of, for example, a social media app. This would mean that user-facing docs would vary.

7.5. Why is documentation so important?

we should probably spend more time on it



via source

7.6. What is a docstring

```
python
```

```
Python 3.11.4 (main, Jul  5 2023, 09:00:44) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> help(print)

>>> def ex():
...     ''' doc string
...     '''
...     padd
...
>>> help(ex)

>>>
>>> # a comment
>>> exit()
```

7.7. So, how do we do it?

Documenation Tools

Skip to main content

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

7.8. Install Jupyter-book

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 going to use this, so that we can use most of the power of sphinx without having to learn ReStructured Text which is more complex than markdown syntax.

install jupyterbook on Mac or linux those instructions will work on your regular terminal, if you have python installed. On Windows those instructions will work in the Anaconda prompt or any other terminal that is set up with python. If these steps do not make sense see the recommendations in the syllabus for more instructions including videos of the Python install process in both Mac and Windows.

```
pip install jupyter-book
```

Note

this command does a lot of stuff, it installs `jupyter-book` and all of its dependencies, which is a lot. I removed the output from the notes

the thing that matters is that near the end it will say "successfully installed"

Now we move to our inclass repo:

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

```
github-inclass-fa23-brownsarahm
```

And note that we can see the other folders we have made so far, but note that we are *not inside* any of them.

We can create this

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

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

Your book template can be found at

    tiny-book/

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

and it tells us the relative path to the new folder, we can use `ls` again to see that it is there.

```
ls
```

```
github-inclass-fa23-brownsarahm tiny-book
```

7.9. Parts of a Jupyer book

First we go in the directory:

```
cd tiny-book/
```

Then look

```
ls
```

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

7.9.1. starting a git repo locally

Before we work with the files, let's commit the current version

```
git status
```

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

We cannot check status because we are not in a git repo.

Recall a git repo is a folder that contains a `.git` directory with the versions of our project in it. This directory is typically hidden, so to confirm it is not there we need the `-a` option on `ls`

```
ls -a
```

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

We can create a new repo with `git init` and the path to wher we want it. We want it at our current working directory, which 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:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/brownsarahm/Documents/inclass/systems/tiny-book/.git/
```

Here we are faced with a social aspect of computing that is *also* a good reminder about how git actually works

7.9.2. 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.

we’ll change our default branch to main

```
git branch -m main
```

Now, we can look at the directory to see how `git init` changed the folder.

```
ls -a
```

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

Now we have the `.git` directory!

and

```
git status
```

```
On branch main

No commits yet

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

it works!

and we will commit the template.

```
git add .
```

```
git commit -m 'init jupyterbook'
```

```
[main (root-commit) 3fcd849] init jupyterbook
 9 files changed, 341 insertions(+)
 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
```

7.9.3. Structure of a Jupyter book

```
ls
```

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

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

- config defaults
- toc file formatting rules
- the `*.md` files are content
- the `.bib` file is bibiology 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

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.

7.9.4. 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

Skip to main content

Not

This i
helpir
chara
return
Mac
chara
(inclu
`LF` c
on W
chara
one li
peopl
syste
from
direct
with
worki

When
you n
will ha
collab
runs
opera
does)
make
interp
you o
Windo
when
(stagi
it’s th

git do
is add
under

7.9.5. Configuration options

The config file template is pretty complete, including the URL of the webpage with all of the detailed setting options.

The configuration file, tells jupyter-book basic iformation about the book, it provides all of the settings that jupyterbook and sphinx need to render the content as whatever output format we want.

```
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
```

7.9.6. Content Organization

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

format: jb-book
root: intro
chapters:
- file: markdown
- file: notebooks
- file: markdown-notebooks
```

Skip to main content

7.9.7. Building a book

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

Building is transforming from input to output format. In this case from markdown, yaml, and notebook files into HTML.

We will see later that building is also what we refer to the whole process from source code to executable.

```
jupyter-book build .
```

```
Running Jupyter-Book v0.15.1
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 v4.5.0
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 entries
myst v0.18.1: MdParserConfig(commonmark_only=False, gfm_only=False, enable_extensions=['colon_fence', 'dollar
myst-nb v0.17.2: NbParserConfig(custom_formats={}, metadata_key='mystnb', cell_metadata_key='mystnb', kernel_
Using jupyter-cache at: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/.jupyter_cache
building [mo]: targets for 0 po files that are out of date
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 local
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executed notebook in 1.89 secon
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executing notebook using local CWD [r
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executed notebook in 2.56 seconds [my

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
copying images... [100%] _build/jupyter_execute/137405a2a8521f521f06724f6d604e5a5544cce7bd94d903975cee58b0605
copying static files... done
copying extra files... done
dumping search index in English (code: en)... done
dumping object inventory... done
build succeeded.

The HTML pages are in _build/html.
[etoc] missing index.html written as redirect to 'intro.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
```

Skip to main content

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

Try it yourself

Which files created by the template are not included in the rendered output? How could you tell?

```
ls
```

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

we note that this made a new folder called `_build`. we can look inside there.

```
ls _build/
```

```
html            jupyter_execute
```

and in the html folder:

```
ls _build/html/
```

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

We got HTML, CSS, and javascript files without having to write them manually.

We can also copy the path to the file and open it in our browser

we can change the size of a browswer window or use the screen size settings in inspect mode to see that this site is responsive.

We didn't have to write any html and we got a responsive site!

If you wanted to change the styling with sphinx you can use built in themes which tell sphinx to put different files in the `_static` folder when it builds your site, but you don't have to change any of your content! If you like working on front end things (which is great! it's just not alwasy the goal) you can even build your own theme that can work with sphinx.

7.10. Ignoring Built files

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

We do not want to keep track of changes for the built files since they are generated from the source files. It's redundant and makes

```
git status
```

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        _build/

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

```
ls
```

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

Git helps us with this with the .gitignore

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

```
git status
```

```
On branch main
Untracked files:
  (use "git add <file>..." to include in what will be committed)
        .gitignore

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

now that's the only new file as far as git is concerned, so we will track this,

```
git add .
```

```
git commit -m 'ignores build'
```

```
[main 9c4157f] ignores build
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

We can look at the file's contents

```
cat .gitignore
```

```
_build/
```

Skip to main content

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

Right now, we do not have any remotes

```
git remote
```

If we do the same in another repo that we got by cloing *from* GitHub, they all have remotes.

```
cd ../github-inclass-fa23-brownsarahm/
git remote
```

```
origin
```

Typically they all use the default name `origin`

Back to our `tiny-book`

```
cd ../tiny-book/
```

For today, we will create an empty github repo shared with me using the link shared on prismia.

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, following the instructions from the blank repo page.

```
git remote add origin https://github.com/introcompsys/f23-tiny-book-brownsarahm.git
```

```
git branch -M main
```

and finally we can push!

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

```
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 16.45 KiB | 8.23 MiB/s, done.
Total 14 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/introcompsys/f23-tiny-book-brownsarahm.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

```
git remote
```

Skip to main content

origin

7.12. Prepare for Next Class

1. Follow up on your grade plan PR and self reflection that you complete in lab

7.13. Review today's class

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. Add `docs.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.

7.14. More 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. 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.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.

7.15. Experience Report Evidence

Tiny book repo should exist. Append your terminal output or this repo's git log to your experience report.

7.16. Questions After Today's Class

7.16.1. Why would one experience missing .exe files within their python library, even after reinstalling python?

it is probably a missing path

7.16.2. If I want to track the _build file will it automatically be tracked now or do I have to remove it from gitignore?

To track the files in the build directory you would need to delete it from the `.gitignore` file.

Also, of note, git does not track folders, it tracks files, but listing folder paths can help

Skip to main content

7.16.3. Will jupyter book look for a universal syntax for documentation or will it look for a language dependent syntax?

Default it works with Python, but the sphinx docs describe how to work with other languages. The jupyter book docs include a page on how sphinx and jupyter-book are related and a page on how to use other sphinx features in a jupyter book.

Jupyter notebooks also have alternative kernels so notebook styled pages can be written in these languages as well.

7.16.4. Can you create an entire repository with only git?

By definition git is what makes a git repository.

7.16.5. What is jupyter used for

Jupyter is a large software project designed to support computational sciences.

it includes the notebook file format specification, the lab interface for working with them, the hub for hosting it on a server.

Jupyter-book is a separate, but related, project that is highly compatible.

7.16.6. My question is related to hooking up to remote. It might be irrelavent now but could be a learning experience, I had trouble setting up my xcode local git repo with github, it kept giving an error about needing a ssh, eventually I got it working using some tutorials online but I'm wondering what would have made that happen.

This sounds like you may have copied the ssh url to the remote instead of the https version?

7.16.7. So pip3 install jupyter-book ended up not working up on Git Bash so I had to install Anaconda.Navigator. Now I am using the powershell prompt there. Should I expect massive differences even though I managed to catch up via the commands you posted in Prismia?

Python things do not work in GitBash by default, but do work in Anaconda Prompt, where git operations do not work. You may need to use two terminals, but should be able to do everything you need.

7.16.8. Is it possible to edit the Jupyter Book website on a web front site or does it have to be in the files?

You have to edit the source files not the built ones to keep the changes.

You can however, add custom HTML and/or CSS.

7.16.9. Are their alternatives to Jupyterbook that have similar functionality?

Yes, this is the practice badge.

Skip to main content

7.16.10. If I want to track the _build file will it automatically be tracked now or do I have to remove it from gitignore?

It has to not be in .gitignore to be tracked.

7.16.11. Will jupyter book look for a universal syntax for documentation or will it look for a language dependent syntax?

By default it looks for Python, but you can configure it to work with other languages.

Important

This is one thing you could do for a build badge.

7.17. Good Questions for Explore Badges

7.17.1. What other platforms/libraries are used to generate similar webpages?

Learning about the landscape of static site generators is a great topic.

8. What *is* git?

8.1. Admin

- I will be using captioning in classes going forward. This is not a persistent transcript, but instead an ephemeral, live, AI generated caption of all of what I say. At the start of each class, I will send the link to the page with the connection information for you to join the broadcast.
- To get credit for the course toward your major, use the provided template for the currciulum modification form. Complete the form with your personal information and send it to Professor DiPippo to sign.
- Reminder: how to makeup experience badges
- Notice: minor revision to that process

Note

Notice, since the course website is a Jupyter book, if you look in the raw markdown for this page, the link to the form is the relative path. One of the advantages of using a static site generator of any kind is that you can create references like variables to refer to other parts of the document and the static site generator can

8.2. Today's goals

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

Today, we are going to learn *what* git is, next we will learn about *why* git and bash are designed the way they are designed and next week we will learn *how* git creates a commit

Study Tip

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

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.

8.3. Why are we defining git in week 5 of using it?

git book is the official reference on git.

Note

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

From here, we have the full definition of git by the git developers:

> git is fundamentally a content-addressable filesystem with a VCS user interface written on top of it.
>
> —[git scm book]](https://git-scm.com/book/en/v2/Git-Internals-Plumbing-and-Porcelain)

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

Today, however, other version control systems are barely in use and this course is for computer science *students* and part of the goal is to learn what a Version Control System is.

8.4. Git is a File system

Let's break down the definition

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

Other examples of key-value pairs you may have seen before include:

- python dictionaries

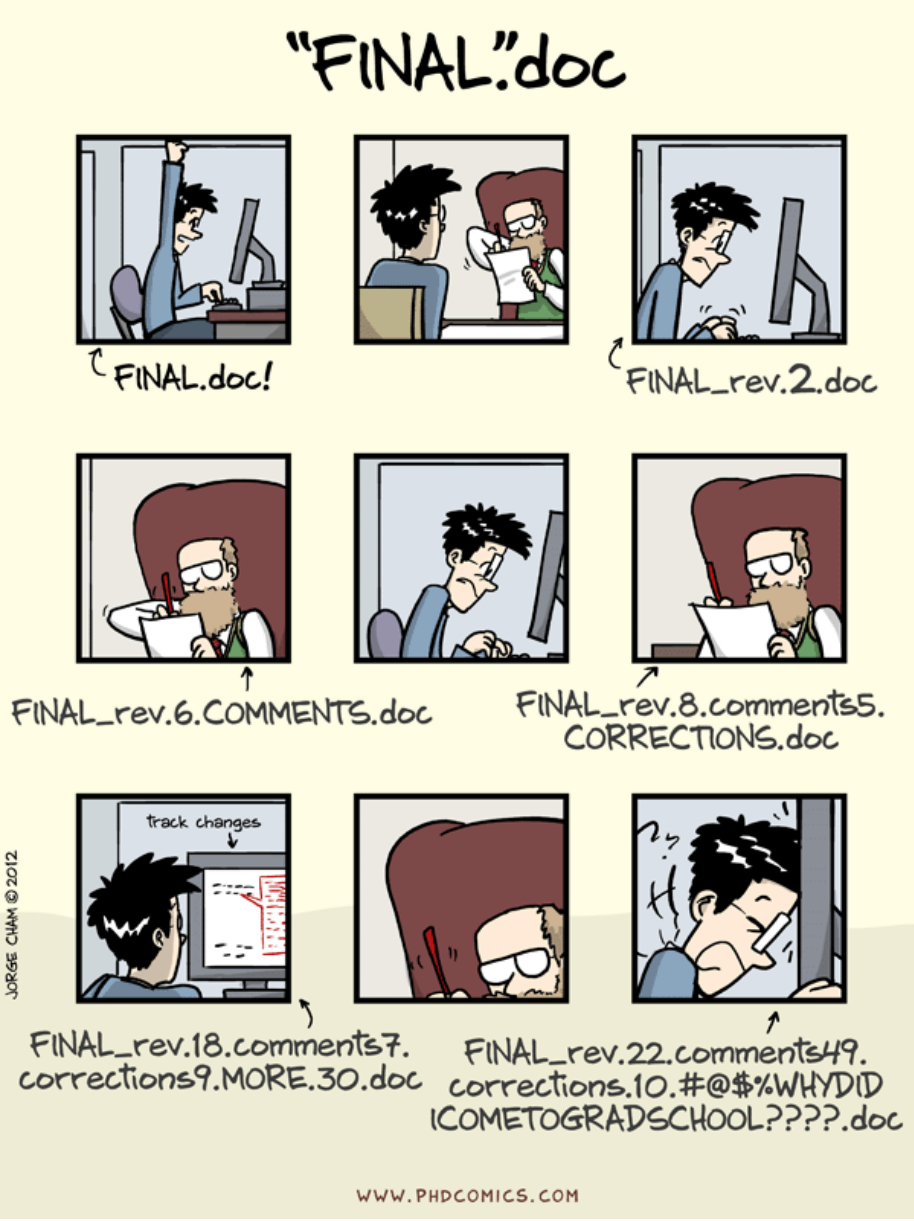
- parameter, passed values
- yaml files (from jupyter-book last week)

Again, we see that studying the developer tools is a good way to reinforce other concepts in computer science. Modularity and Abstraction are the core foundations of the field, and while they do have their limits of applicability, it means that if you get a really good understanding of the core abstractions, that makes learning other things in CS faster.

You can apply your past experience with these other concepts to help understand what to expect about how git works.

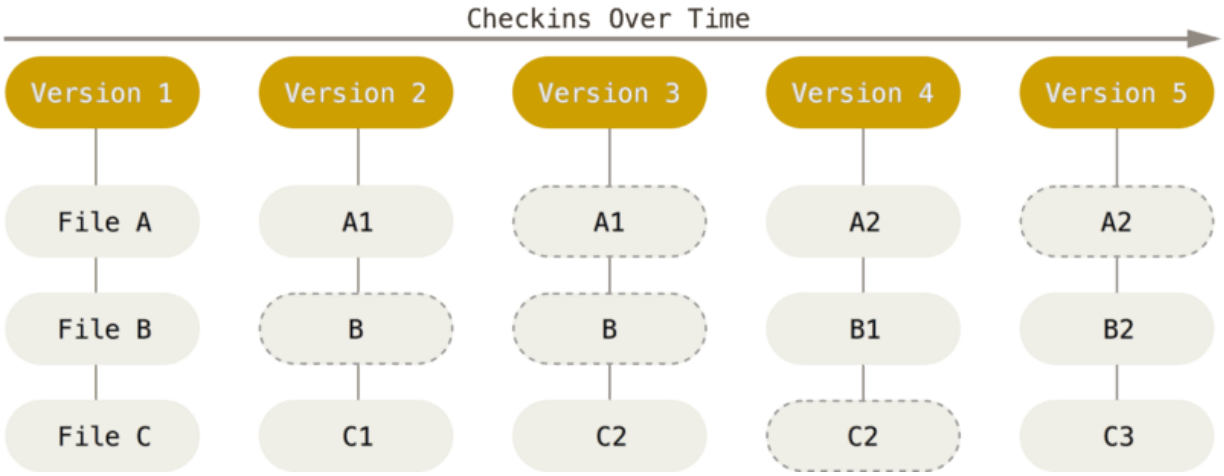
8.5. Git is a Version Control System

In the before times

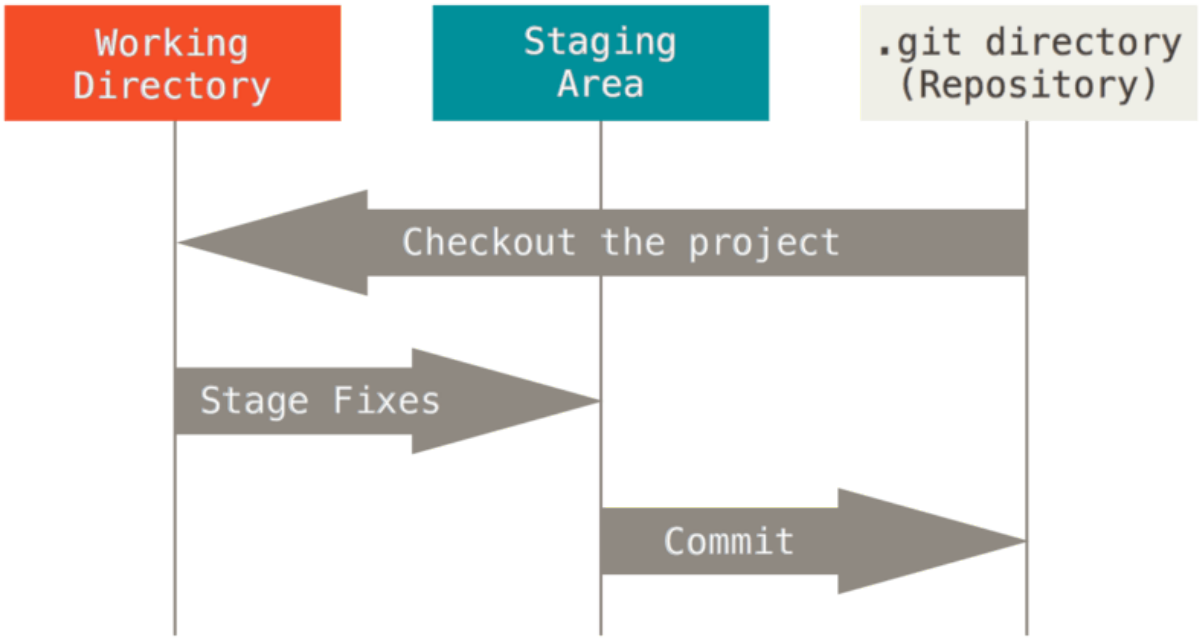


git stores **snapshots** of your work each time you commit.

Skip to main content



it uses 3 stages:



8.6. 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 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.

8.7. Git is distributed

What does that mean?

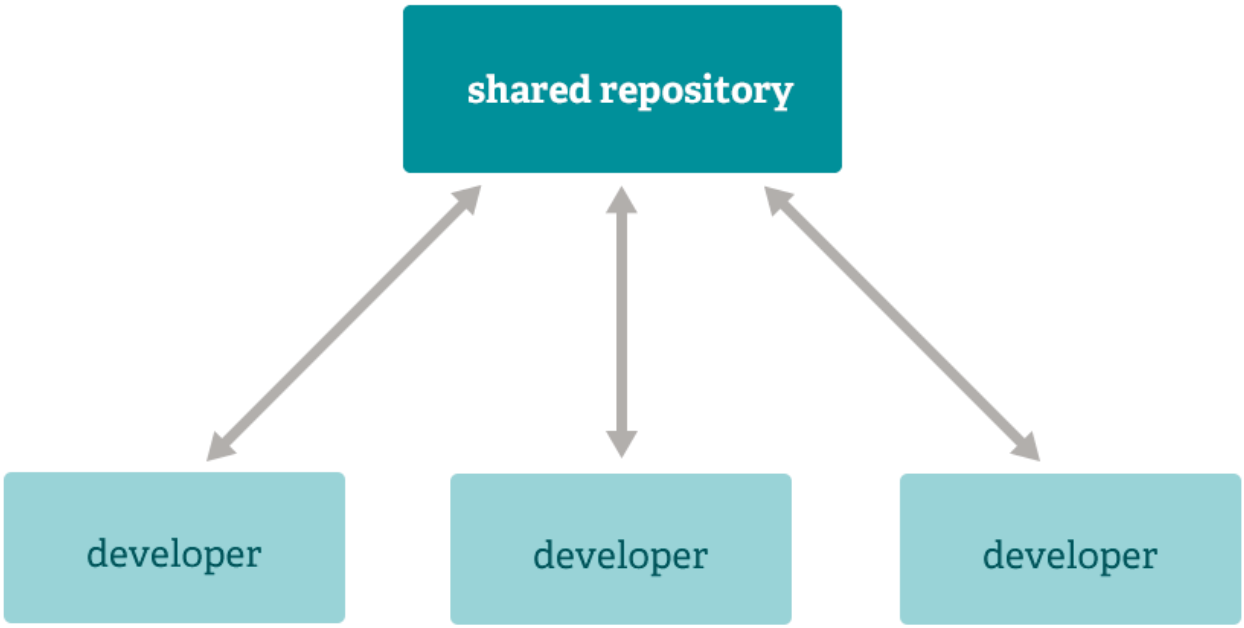
Git runs locally. It can run in many places, and has commands to help sync across remotes, but git does not require one copy of the repository to be the “official” copy and the others to be subordinate. git just sees repositories.

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.

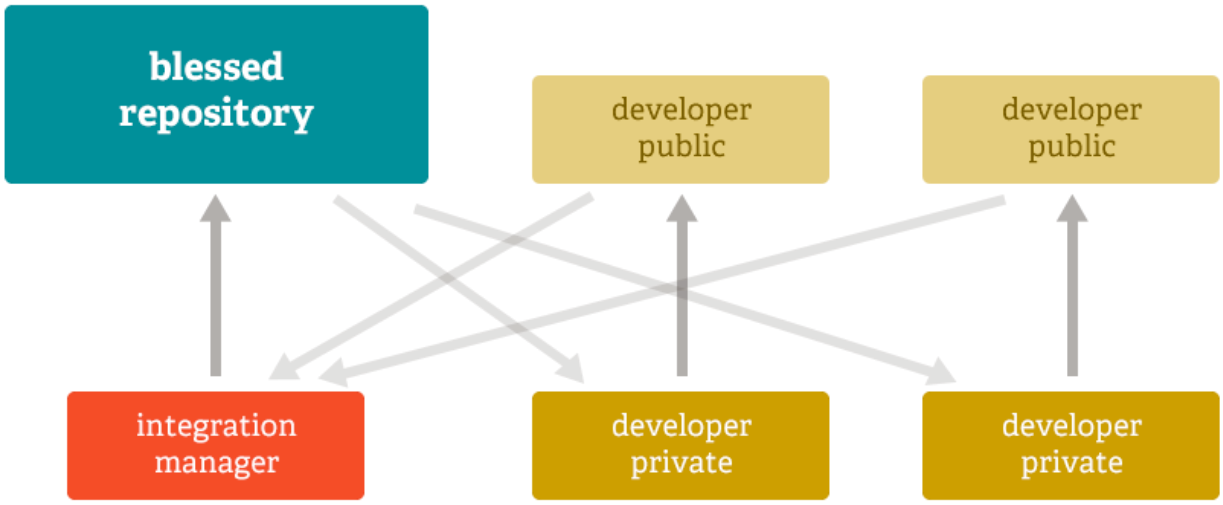
It’s about the workflows, or the ways we socially *use* the tool.

Some example workflows include:

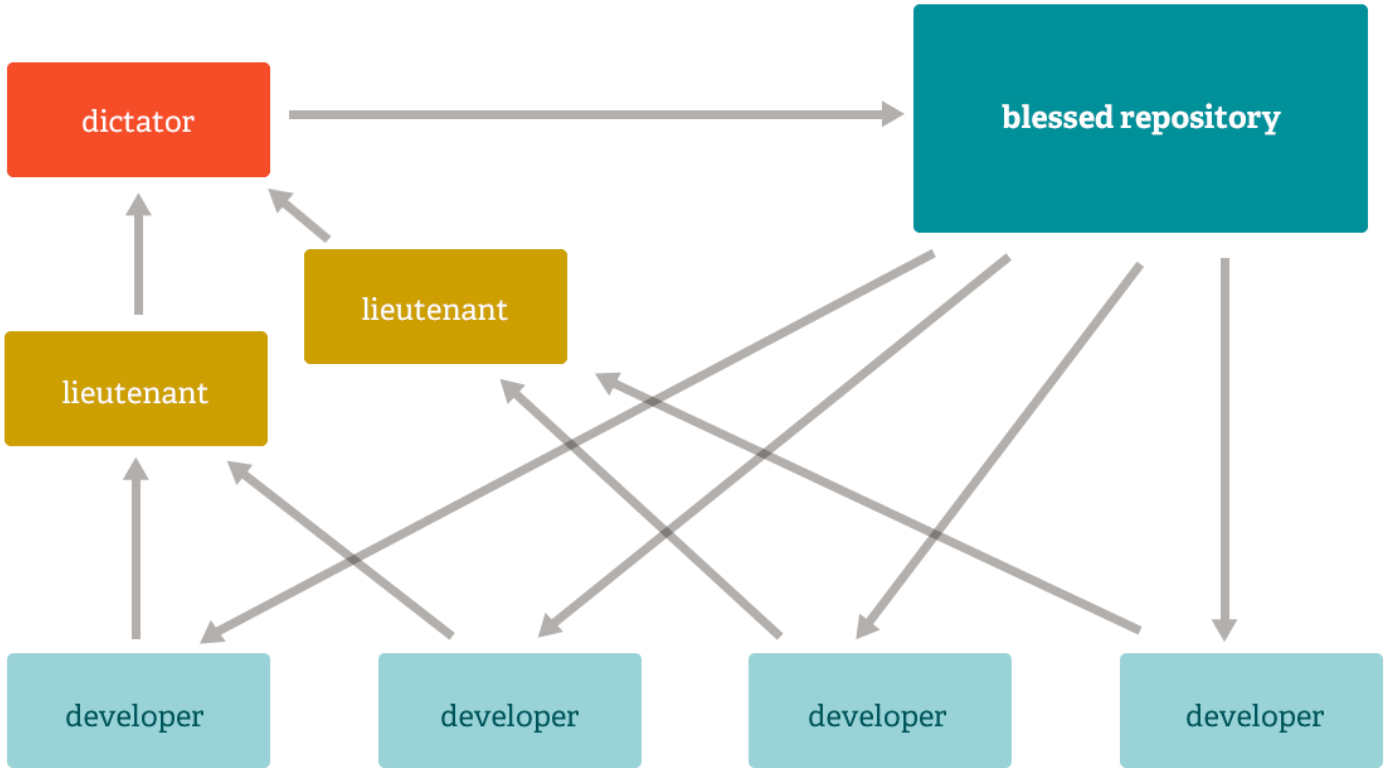
8.7.1. Subversion WOrkflow



8.7.2. Integration Manager



8.7.3. dictator and lieutenants



8.8. *How* does git do all these things?

Let's look at git again in our `github-inclass` repo.

I was still in my `tiny-book` repo so I went up one level, then used `ls` to remember the exact name of the `github-inclass` repo

```
cd ../
ls
```

Skip to main content

We will change to the github inclass repo and use `pwd` to view the current working directory.

```
cd github-inclass-fa23-brownsarahm/
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-inclass-fa23-brownsarahm
```

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
```

and we get this output:

```
.git/objects//04/2a42eb47c33ee43d793feb4d891a93e7460527
.git/objects//04/ab89e167ed77bc2a95710f69f68d91a6219471
.git/objects//69/3a2b5b9ad4c27eb3b50571b3c93dde353320a1
.git/objects//93/4c15dc2655c988c981d9a836783afebda77355
.git/objects//5a/a1ed29b82e1cebb8527019b0e594ba71dda214
.git/objects//5f/e5a9821625fad2cca4c500e497e6694132c303
.git/objects//d7/6bc523443bda5a5daae2fe7fcfbf6fba71ae6d
.git/objects//b3/78bd148e53dfa7195c58123362e40ae12ef3e7
.git/objects//bc/281792d6ab62b153d7bf44f7985ec7cfc3b850
.git/objects//ca/eacb503cf4776f075b848f0faff535671f2887
.git/objects//ca/feca302e31c65139b4a5294356e1ea8595dcb1
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.rev
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.pack
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.idx
.git/objects//11/d53c24bb5d2bf2e3f645ef188f8bc75fa9c911
.git/objects//45/fcb1dd311e5e45af759cb3627dca5f47f58f04
.git/objects//75/6c4879c0447db20980f73a26bc2ba072e08a6d
.git/objects//44/3f164cdde5059d78df6a61ca3f07bc6a605eb0
.git/objects//43/a1267370f1af98071d53f8508abbc56fa3abde
.git/objects//88/5588412d138cceb89f06ffed5e83c316c2b593
.git/objects//5c/8aaa9f2a129d551b8cb2cb294676f63c4af410
.git/objects//65/e9e39935be8400ef12cc9003592f12244b50da
.git/objects//3a/cf0fb1c2febd24561294bfb966e1ad1f033eb8
.git/objects//98/96f7a7000a7b9d2fdb12047a141524358286c3
.git/objects//37/0e04baf4f62d1e62f4949208bc5e4d33af5336
.git/objects//6d/4dbd33860fceb9c87bd3c4509deff8cecb3f45
.git/objects//39/f1c5eabb1458fa6cf9042611599b69665cf288
.git/objects//55/56d17391aeeec9b2b86d2821c011d7ed5377aa
.git/objects//b8/6eb90ba1ae5504edfcdc9ef8879e1c6d7a1b75
.git/objects//b6/2d570421c3096d8c80c7df56357cdd3203fd3a
.git/objects//ea/c84c8320a3ab4f37a441a332b828c45ecedcc9
.git/objects//e1/82616690a91a8d0e363f4143e68dd9e136ccee
.git/objects//e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects//8c/7cefb877c62a46a3b71c68a858c24075b379fe
.git/objects//76/8dec80c5e0734476d476ae83376c9c786b6450
.git/objects//2b/cb5d446129df427a1ce09e8ba658a5bb8ceca3
```

This is a lot of files! It's more than we have in our working directory.

```
ls
```

```
API.md                  about.md                helper_functions.py
CONTRIBUTING.md         abstract_base_class.py  important_classes.py
```

Skip to main content

This is a consequence of git taking snap shots and tracking both the actual contents of our working directory **and** our commit messages and other meta data about each commit. The content of the `.git/objects` is *everything* git knows about our project.

8.9. Git HEADs

If we look at the content of the `.git` directory there is always a file called `HEAD`

```
ls .git/
```

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

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.

the files in all caps are like gits variables. Lets look at the one called `HEAD` we have intereacted 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 `*HEAD` files are other "heads" or "tips" of the tree of commits. For example, I have one lingering from the last time I used `rebase` in my repo.

> **Note**
>
> You may or may not have the same heads that I do, because you may have missed a step or I may have done an extra thing in my repo to demo something to answer a question in office hours

```
cat .git/REBASE_HEAD
```

```
b378bd148e53dfa7195c58123362e40ae12ef3e7
```

This poitns to a specific commit, not a branch, because rebase occurs at a pair of specific commits, and does not move forward, this is sort of like a temporary variable, that does not get destroyed.

We should all have `ORIG_HEAD` which is `origin/head`

```
cat .git/ORIG_HEAD
```

Skip to main content

```
5c8aaa9f2a129d551b8cb2cb294676f63c4af410
```

that points to a specific commit, instead of a branch.

We can see that git status matches what branch the `HEAD` file shows.

```
git status
```

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

nothing to commit, working tree clean
```

that matches the `HEAD` as expected. Lets switch and look agian.

first, get the options,

```
git branch
```

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

then `checkout` one of the exising branches

```
git checkout fun_fact
```

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

and we can confirm that one of the things `checkout` does is updates the head file:

```
cat .git/HEAD
```

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

8.10. Branches are pointers to commits

Branches are implemented as files in `.git/refs/heads/` that contain the hash of the most recent commit “on” that branch. It’s useful to think of branches having multiple commits, and a tree like structure:

gitGraph commit commit branch fun_fact checkout fun_fact commit commit checkout main commit merge fun_fact commit
but literally, they are a pointer to a single commit. (remember each commit has a pointer to its “parent” or preceiding commit)

We can see what commit a branch points to:

Skip to main content

```
cat .git/refs/heads/fun_fact
```

```
756c4879c0447db20980f73a26bc2ba072e08a6d
```

`git branch` reads from the `.git/refs/heads` directory. It has some other options that make it more powerful, but its default behavior is very similar to

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

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

> ⚠ **Warning**
>
> This was an answer to a question, but I do not remember the question

```
cat .git/config
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "1-create-an-about-file"]
        remote = origin
        merge = refs/heads/1-create-an-about-file
[branch "fun_fact"]
        remote = origin
        merge = refs/heads/fun_fact
```

8.11. Git Objects

Lets go back to the 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

```
classDiagram
```

Skip to main content

```
      - hash: blob
      - string: type
      - string:file name
    }
    class commit{
        hash: parent
        hash: tree
        string: message
        string: author
        string: time
    }
    class blob{
        binary: contents
    }
    class object{
        hash: name

    }
    object <|-- blob
    object <|-- tree
    object <|-- commit
```

Again we can look at the list of objects:

```
find .git/objects/ -type f
```

```
.git/objects//04/2a42eb47c33ee43d793feb4d891a93e7460527
.git/objects//04/ab89e167ed77bc2a95710f69f68d91a6219471
.git/objects//69/3a2b5b9ad4c27eb3b50571b3c93dde353320a1
.git/objects//93/4c15dc2655c988c981d9a836783afebda77355
.git/objects//5a/a1ed29b82e1cebb8527019b0e594ba71dda214
.git/objects//5f/e5a9821625fad2cca4c500e497e6694132c303
.git/objects//d7/6bc523443bda5a5daae2fe7fcfbf6fba71ae6d
.git/objects//b3/78bd148e53dfa7195c58123362e40ae12ef3e7
.git/objects//bc/281792d6ab62b153d7bf44f7985ec7cfc3b850
.git/objects//ca/eacb503cf4776f075b848f0faff535671f2887
.git/objects//ca/feca302e31c65139b4a5294356e1ea8595dcb1
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.rev
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.pack
.git/objects//pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.idx
.git/objects//11/d53c24bb5d2bf2e3f645ef188f8bc75fa9c911
.git/objects//45/fcb1dd311e5e45af759cb3627dca5f47f58f04
.git/objects//75/6c4879c0447db20980f73a26bc2ba072e08a6d
.git/objects//44/3f164cdde5059d78df6a61ca3f07bc6a605eb0
.git/objects//43/a1267370f1af98071d53f8508abbc56fa3abde
.git/objects//88/5588412d138cceb89f06ffed5e83c316c2b593
.git/objects//5c/8aaa9f2a129d551b8cb2cb294676f63c4af410
.git/objects//65/e9e39935be8400ef12cc9003592f12244b50da
.git/objects//3a/cf0fb1c2febd24561294bfb966e1ad1f033eb8
.git/objects//98/96f7a7000a7b9d2fdb12047a141524358286c3
.git/objects//37/0e04baf4f62d1e62f4949208bc5e4d33af5336
.git/objects//6d/4dbd33860fceb9c87bd3c4509deff8cecb3f45
.git/objects//39/f1c5eabb1458fa6cf9042611599b69665cf288
.git/objects//55/56d17391aeeec9b2b86d2821c011d7ed5377aa
.git/objects//b8/6eb90ba1ae5504edfcdc9ef8879e1c6d7a1b75
.git/objects//b6/2d570421c3096d8c80c7df56357cdd3203fd3a
.git/objects//ea/c84c8320a3ab4f37a441a332b828c45ecedcc9
.git/objects//e1/82616690a91a8d0e363f4143e68dd9e136ccee
.git/objects//e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects//8c/7cefb877c62a46a3b71c68a858c24075b379fe
.git/objects//76/8dec80c5e0734476d476ae83376c9c786b6450
.git/objects//2b/cb5d446129df427a1ce09e8ba658a5bb8ceca3
```

I am going to look at my alphabetically last object:

Skip to main content

Not

this d
You c
with r

Imp

Creat
diagra
oppor
(one
(multi

Also,
featur
scree
findin
are n
These
webs
badge

Warning

you cannot copy my path here, you have to use **your** last path

I copied the last line from my output above to make the next command.

```
cat .git/objects//2b/cb5d446129df427a1ce09e8ba658a5bb8ceca3
```

```
x+)JMU040g01??7??$???x^?}#}??}????kW?ch``fb?????□?°#o'??u?,o?□??}?Ln?t)TQbR~i    H?????□?s+?g%??gF??O?.{
```

This looks like nonsense. That is because the object files are stored in a binary, not human-readable format. When we use cat, our terminal takes every 8 bits and turns it into an ascii character and that is what we see.

Question from class

Is that they are binary files why the files in git do not have extensions?

The answer is no. Actually *no* file strictly requires a file extension. file extensions are mostly for people

Not

this r
again
reme
a high
about

Now, we can use `git cat-file` which is the first plumbing command we have seen so far to look at the git object associated with the the last hash in the list above.

Remember git commands that take a hash as input do not need the whole hash. They need at least 4 characters and enough to unique.

With the `-t` option `git cat-file` tells you the type of object.

```
git cat-file -t 2bcb
```

```
tree
```

Warning

at this point we all got different object types, (well out of the 3)

Then I looked at the *contents* of that same objects.

```
git cat-file -p 2bcb
```

```
040000 tree 95b60ce8cdec1bc4e1df1416e0c0e6ecbd3e7a8c    .github
100644 blob b86eb90ba1ae5504edfcdc9ef8879e1c6d7a1b75    README.md
100644 blob 443f164cdde5059d78df6a61ca3f07bc6a605eb0    about.md
```

Since mine is a tree we can see that it has a list of items and each item in the list includes:

Skip to main content

- mode
- type (tree or blob)
- the hash
- the file name

Then for the README blob, I looked at the contents:

```
git cat-file -p b86e
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406
```

I can use `cat` to see the file and currently it matches.

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406
```

If I switch back to the main branch

```
git checkout main
```

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

Now it no longer matches:

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown

[![Open in Codespaces](https://classroom.github.com/assets/launch-codespace-7f7980b617ed060a017424585567c406
age=35
|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 |
> | tests_abc.py | tests for constructors and methods in abstract_base_class.py|
> | tests_helpers.py | tests for constructors and methods in helper_functions.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 |
```

Important

If your object was a commit: look at the contents of the tree object after.

If your object was a blob, look at the type of the next one up in your find output until you find a tree or commit, then trace out one step like I did above.

8.12. Prepare for Next Class

1. Review the notes from past classes
2. 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 or 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 you would describe the design of command line tools vs other GUI based too
```

8.13. Review today's class

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 each tree and blob in that tree to the same file. 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.

8.14. More Practice

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. Find the hash of the blob object that contains the content of your gitislike.md file and put that in the comment of your badge PR for this badge.

```
## 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 you
```

8.15. Experience Report Evidence

Append the contents of one of your trees or commits and one blob or tree inside of that first one to the bottom of your experience report.

8.16. Questions After Today's Class

8.16.1. Are the contents of one item of a tree stored in key-value pairs or are they like a tuple of information?

A tree can be though of like a list of tuples.

Recall the one that we looked at was like:

```
040000 tree 95b60ce8cdec1bc4e1df1416e0c0e6ecbd3e7a8c    .github
100644 blob b86eb90ba1ae5504edfcdc9ef8879e1c6d7a1b75    README.md
100644 blob 443f164cdde5059d78df6a61ca3f07bc6a605eb0    about.md
```

This ree has 3 items. Each item has a type of file (that number), a type of object, the hash of the object and the name of the content content.

The 040000 type is for folder and 100644 is for a file.

8.16.2. How often does the ordinary developer need to use plumbing commands?

Only when you make a mistake, but they are really helpful for getting a better understanding.

8.16.3. Why do we need to know about our objects file?

Buliding up a better understanding of what git is, sets us up to study how it works in greater detail. This understanding helps you use git more effectively, both to not make mistakes as often and to fix them if you (or a coworker/team mate) still do make one.

8.16.4. Can the hashes be decoded?

No these are cryptographic hashes, so they are not reversible.

8.16.5. How to learn the commands?

Practice!

8.16.6. Why would anyone ever want to use a GUI once they learn Terminnal commands? Would there still be benefits to a GUI?

Some tools have helpful utility and no command line interface to them. When using a touchscreen device?

Skip to main content

8.16.7. What command to find specific type, such as commits, or trees, or blobs?

You can get a list of all objects with `find` and then check the type.

You can also trace through following the pointers. So starting from `git log` to get the commit hashes, and then using one of those to get the tree, then the tree to get blobs.

8.16.8. What defines a VCS?

A version control system has to keep track of different versions of code in a systematic way. A more detailed answer for this is a good explore badge topic.

8.16.9. Does the commit hash account for just the email and time or does it use other data?

The commit hash is produced using the hashing function with the inputs:

- a header
- the tree
- the author with time stamp
- the committer with time stamp
- a blank line
- the commit message

8.16.10. What are the practical uses of finding the git object types for developers?

This is how you can trace out and manually recover something you thought was lost.

More importantly undersanding what happens under the hood prepares you to learn more advanced porcelain commands.

8.16.11. Will we see this in future classes?

Yes!

8.17. Questions Good for Explore badges

8.17.1. Version control systems generally

- What other version control systems are there besides git?
- what features do they have in common?

9. Why are these tools organized this way?

Skip to main content

organized.

Understanding the context and principles will help you:

- remember how to do things
- form reliable hypotheses about how to fix problems you have not seen before
- help you understand when you should do things as they have always been done and when you should challenge and change things.

9.1. Admin

Note

last class notes were posted and announced, but not linked

Important

An issue or PR is always welcome and worth a community badge if I make a release and the website does not update

Note

community badge opp

I used mulitple cursors to type out that checklist by copying from the `_toc.yml` file and then editing each line to start `- [ ] fix` instead of `  - file`

9.2. Why should we study design?

- it is easy to get distracted by implementation, syntax, algorithms
- but the core *principles* of design organize ideas into simpler rules

9.3. Why are we studying developer tools?

Recall:

> 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:

- 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

9.4. Unix Philosophy

sources:

- wiki
- a free book
- composability over monolithic design
- social conventions

The tenets:

1. Make it easy to write, test, and run programs.
2. Interactive use instead of batch processing.
3. Economy and elegance of design due to size constraints (“salvation through suffering”).
4. Self-supporting system: all Unix software is maintained under Unix.

For better or worse this is dominant, so understanding this is important.

Also, 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.

“normal folks” is the author’s term presumably attempting to describe a typical, nondeveloper computer user

If you have the `gh` CLI installed and working locally, open the terminal that it works in and navigate to your KWL repo.

If you do not have that working locally, open a codespace on the main branch of your KWL repo

(I have it working locally and working locally lets me save my terminal output for the notes more easily)

cat the action file that posts your badges

```
cd Documents/inclass/systems/fa23-kwl-brownsarahm/
```

then check the repo’s status

```
git status
```

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

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

```
name: Create badge issues (Edit before Running)
on:
  workflow_dispatch

# once you edit, change the name

jobs:
  check-contents:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v1
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        pretitle="prepare-"$(date +"%Y-%m-%d")
        sysgetassignment | gh issue create --title $pretitle --label prepare --body-file -
        rtitle="review-"$(date +"%Y-%m-%d")
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        pratitle="practice-"$(date +"%Y-%m-%d")
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file -
      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 experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

```
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
```

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

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

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

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

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

```
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
```

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

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

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

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

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

```
cd fa23-kwl-brownsarahm/
```

```
-bash: cd: fa23-kwl-brownsarahm/: No such file or directory
```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/fa23-kwl-brownsarahm
```

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

```
name: Create badge issues (Edit before Running)
on:
  workflow_dispatch

# once you edit, change the name

jobs:
  check-contents:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v1
      with:
        python-version: 3.9
```

```
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        pretitle="prepare-"$(date +"%Y-%m-%d")
        sysgetassignment | gh issue create --title $pretitle --label prepare --body-file -
        rtitle="review-"$(date +"%Y-%m-%d")
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        pratitle="practice-"$(date +"%Y-%m-%d")
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file -
      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 experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

```
history | gh issue create --title example --body-file -
```

```
Creating issue in introcompsys/fa23-kwl-brownsarahm

https://github.com/introcompsys/fa23-kwl-brownsarahm/issues/32
```

We can use `gh` commands without an option to get the help for how to use them.

```
gh issue
```

```
Work with GitHub issues.

USAGE
  gh issue <command> [flags]

GENERAL COMMANDS
  create:      Create a new issue
  list:        List issues in a repository
  status:      Show status of relevant issues

TARGETED COMMANDS
  close:       Close issue
  comment:     Add a comment to an issue
  delete:      Delete issue
  develop:     Manage linked branches for an issue
  edit:        Edit issues
  lock:        Lock issue conversation
  pin:         Pin a issue
  reopen:      Reopen issue
  transfer:    Transfer issue to another repository
  unlock:      Unlock issue conversation
  unpin:       Unpin a issue
  view:        View an issue

FLAGS
  -R, --repo [HOST/]OWNER/REPO   Select another repository using the [HOST/]OWNER/REPO format

INHERITED FLAGS
  --help   Show help for command

ARGUMENTS
  An issue can be supplied as argument in any of the following formats:
  - by number, e.g. "123"; or
  - by URL, e.g. "https://github.com/OWNER/REPO/issues/123".

EXAMPLES
  $ gh issue list
  $ gh issue create --label bug
```

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

```
gh issue view
```

```
accepts 1 arg(s), received 0
```

```
gh issue view 32
```

```
example #32
Open • brownsarahm opened about 2 minutes ago • 0 comments

  762  gh
  763  gh
  764  cd fa23-kwl-brownsarahm/
  765  pwd
  766  cat .github/workflows/getassignment.yml
  767  history | gh issue create --title example --body-file -


View this issue on GitHub: https://github.com/introcompsys/fa23-kwl-brownsarahm/issues/32
```

We can see how this group of commands work with

```
gh issue
```

```
Work with GitHub issues.

USAGE
  gh issue <command> [flags]

GENERAL COMMANDS
  create:      Create a new issue
  list:        List issues in a repository
  status:      Show status of relevant issues

TARGETED COMMANDS
  close:       Close issue
  comment:     Add a comment to an issue
  delete:      Delete issue
  develop:     Manage linked branches for an issue
  edit:        Edit issues
  lock:        Lock issue conversation
  pin:         Pin a issue
  reopen:      Reopen issue
  transfer:    Transfer issue to another repository
  unlock:      Unlock issue conversation
  unpin:       Unpin a issue
  view:        View an issue

FLAGS
  -R, --repo [HOST/]OWNER/REPO   Select another repository using the [HOST/]OWNER/REPO format

INHERITED FLAGS
  --help   Show help for command

ARGUMENTS
  An issue can be supplied as argument in any of the following formats:
```

```
EXAMPLES
  $ gh issue list
  $ gh issue create --label bug
  $ gh issue view 123 --web

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

```
gh issue close 32
```

```
✓ Closed issue #32 (example)
```

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

```
name: Create badge issues (Edit before Running)
on:
  workflow_dispatch

# once you edit, change the name

jobs:
  check-contents:
    runs-on: ubuntu-latest
    steps:
    - uses: actions/checkout@v2

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v1
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        pretitle="prepare-"$(date +"%Y-%m-%d")
        sysgetassignment | gh issue create --title $pretitle --label prepare --body-file -
        rtitle="review-"$(date +"%Y-%m-%d")
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        pratitle="practice-"$(date +"%Y-%m-%d")
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file -
      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 experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

```
gh issue list
```

```
Showing 28 of 28 open issues in introcompsys/fa23-kwl-brownsarahm

#31  practice-2023-10-04  practice  about 1 day ago
#30  review-2023-10-04    review    about 1 day ago
#29  prepare-2023-10-04   prepare   about 1 day ago
#28  Lab 4                          about 5 days ago
```

```
#25  prepare-2023-09-29   prepare   about 5 days ago
#24  practice-2023-09-27  practice  about 8 days ago
#23  review-2023-09-27    review    about 8 days ago
#22  prepare-2023-09-27   prepare   about 8 days ago
#21  Lab 3                          about 12 days ago
#20  practice-2023-09-22  practice  about 13 days ago
#19  review-2023-09-22    review    about 13 days ago
#18  prepare-2023-09-22   prepare   about 13 days ago
#17  practice-2023-09-20  practice  about 15 days ago
#16  review-2023-09-20    review    about 15 days ago
#15  prepare-2023-09-20   prepare   about 15 days ago
#14  Lab 2                          about 19 days ago
#13  practice-2023-09-15  practice  about 20 days ago
#12  review-2023-09-15    review    about 20 days ago
#11  prepare-2023-09-15   prepare   about 20 days ago
#10  practice-2023-09-13  practice  about 22 days ago
#9   review-2023-09-13    review    about 22 days ago
#8   prepare-2023-09-13   prepare   about 22 days ago
#7   practice-2023-09-07  practice  about 26 days ago
#6   review-2023-09-07    review    about 26 days ago
#5   prepare-2023-09-08   prepare   about 26 days ago
#4   Lab 1                          about 26 days ago
```

Important

This part did not happen in class due to time, but is still important.

We will see these commands again, but this reference will be useful in lab

9.4.1. Processing issues on the terminal

We can use a pipe to also get a count of the issues using the `wc` command which is for word count and its `-l` option to count lines instead.

```
gh issue list | wc -l
```

```
28
```

Or we can filter to get a subset of them using `grep` to filter. By default `grep` searches std in, so when we use it with a pipe it works well.

```
gh issue list | grep 'prepare'
```

```
29      OPEN    prepare-2023-10-04      prepare 2023-10-04 02:38:50 +0000 UTC
25      OPEN    prepare-2023-09-29      prepare 2023-09-29 17:52:35 +0000 UTC
22      OPEN    prepare-2023-09-27      prepare 2023-09-27 03:25:47 +0000 UTC
18      OPEN    prepare-2023-09-22      prepare 2023-09-22 11:11:52 +0000 UTC
15      OPEN    prepare-2023-09-20      prepare 2023-09-20 02:00:07 +0000 UTC
11      OPEN    prepare-2023-09-15      prepare 2023-09-15 02:06:05 +0000 UTC
8       OPEN    prepare-2023-09-13      prepare 2023-09-13 01:42:24 +0000 UTC
5       OPEN    prepare-2023-09-08      prepare 2023-09-08 22:11:50 +0000 UTC
```

Important

we can use `gh issue list --help` to see more options

```
List issues in a GitHub repository.

The search query syntax is documented here:
<https://docs.github.com/en/search-github/searching-on-github/searching-issues-and-pull-requests>

For more information about output formatting flags, see `gh help formatting`.

USAGE
  gh issue list [flags]

FLAGS
      --app string         Filter by GitHub App author
  -a, --assignee string    Filter by assignee
  -A, --author string      Filter by author
  -q, --jq expression      Filter JSON output using a jq expression
      --json fields        Output JSON with the specified fields
  -l, --label strings      Filter by label
  -L, --limit int          Maximum number of issues to fetch (default 30)
      --mention string     Filter by mention
  -m, --milestone string   Filter by milestone number or title
  -S, --search query       Search issues with query
  -s, --state string       Filter by state: {open|closed|all} (default "open")
  -t, --template string    Format JSON output using a Go template; see "gh help formatting"
  -w, --web                List issues in the web browser

INHERITED FLAGS
      --help                     Show help for command
  -R, --repo [HOST/]OWNER/REPO   Select another repository using the [HOST/]OWNER/REPO format

EXAMPLES
  $ gh issue list --label "bug" --label "help wanted"
  $ gh issue list --author monalisa
  $ gh issue list --assignee "@me"
  $ gh issue list --milestone "The big 1.0"
  $ gh issue list --search "error no:assignee sort:created-asc"

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

and then we can use the `-s` flag set to all to see all to make yours match mine.

9.5. How do we Study (computer) Systems

When we think of something as a system, we can study it different ways:

- input/output behavior
- components
- abstraction layers

These basic ideas apply whether a computer system or not. We can probe things in different ways.

In a lot of disciplines people are taught one or the other, or they divide professionally into theorists or experimentalists along the lines.

People are the most effective at working with, within, and manipulating systems when they have multiple ways to achieve the same goal.

Skip to main content

When we study a system we can do so in two main ways. We can look at the input/output behavior of the system as a whole and we can look at the individual components. For each component, we can look at its behavior or the subcomponents. We can take what we know from all fo the components and piece that together. However, for a complex system, we cannot match individual components up to the high level behavior. This is true in both computers and other complex systems. In the first computers in the 1940s, the only things they did was arithmetic and you could match from their components al the way up pretty easily. Modern computers connect to the internet, send signals, load complex graphics, play sounds and many other things that are harder to decompose all at once. Outside of computers, scientists have a pretty good idea of how neurons work and that appears to be the same across mammals and other species (eg squid) but we do not understand how the whole brain of a mammal works, not even smaller mammals with less complex social lives than humans. Understanding the parts is not always enough to understand all of the complex ways the parts can work together. Computers are much less complicated than brains. They were made by brains.

But that fact motivates another way to study a complex system, across levels of abstraction. You can abstract away details and focus on one representation. This can be tied literally to components, but it can also be conceptual. For example, in CSC211 you use a model of stack and heap for memory. It's useful for understanding programming, but is not exactly what the hardware does. At times, it is even more useful though than understanding exactly what the hardware does. These abstractions also serve a social, practical purpose. In computing, and society at large really, we use *standards* these are sets of guidelines for how to build things. Like when you use a function, you need to know it's API and what it is supposed to do in order to use it. The developers could change how it does that without impacting your program, as long as the API is not changed and the high level input/output behavior stays the same.

9.5.1. Behavior

> Try something, see what happens

This is probably how you first learned to use a computer. Maybe a parent showed you how to do a few things, but then you probably tried other things. For most of you, this may have been when you were very young and much less afraid of breaking things. Over time you learned how to do things and what behaviors to expect. You also learned categories of things. Like once you learned one social media app and that others were also *social media* you then looked for similar features. Maybe you learned one video game had the option to save and expected it in the next one.

Video games and social media are *classes* or *categories* of software and each game and app are *instances*. Similarly, an Integrated Development Environment (IDE) is a category of software and VS Code, … are instances. Also, version control is a category of software and git is an instance. A git host is also a category and GitHub is an instance. Just as before you were worried about details you transferred features from one instance to another within categories, I want you to think about what you know from one IDE and how that would help you learn another. We will study the actual features of IDE a=and what you might want to know about them so that you can choose your own. Becoming a more independent developer you'll start to have your own opinions about which one is better. Think about about a person in your life who finds computers and technology overall intimidating or frustrating. They likely only use one social media app if at all, or maybe they only know to make documents in Microsoft word and they think that Google Docs is too much to learn, because they didn't transfer ideas from one to the other.

We have focused on the behavior of individual applications to this point, but there is also the overall behavior of the system in broad terms, typing on the keyboard we expect the characters to show (and when they don't for example in a shell password, we're surprised and concerned it is not working).

9.5.2. Components

> take it apart, assess the pieces

We have the high level parts: keyboard, mouse, monitor/screen, tower/computer. Inside we also tend to know there is a power supply, a motherboard, graphics card, memory, etc.

We can study how each of these parts works while not worrying about the others but having them there. This is probably how you learned to use a mouse. You focused your attention on the mouse and saw what else happened.

Or we can take an individual component and isolate it to study it alone. For a mouse this would be hard. Without a computer attached its output is not very visible. To do this, we would need additional tools to interpret its output and examine it. Most computer components actually would need additional tools, to measure the electrical signals, but we could examine what happens at each part one at a time to then build up what they do.

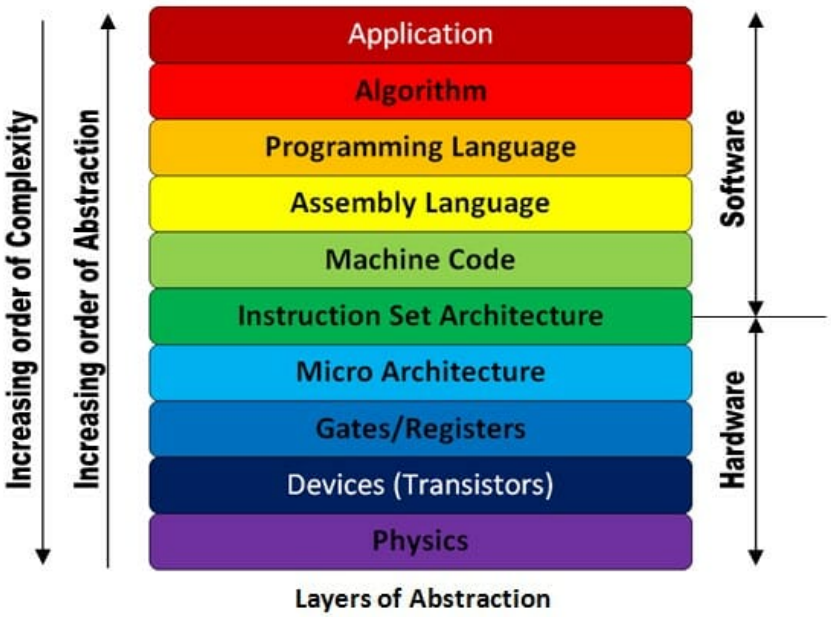
This idea, however that we can use another tool to understand each component is an important one. This is also a way to again, take care and study each piece even within a software-alone system without worrying about the hardware.

9.5.3. Abstraction

> use a model

As we talked about the behavior and abstraction, we talked some about software and some about hardware. These are the two coarsest ways we can think about a computer system at different levels of abstraction. We can think about it only in physical terms and examine the patterns of electricity flow or we can think about only the software and not worry about the hardware, at a higher level of abstraction.

However, two levels is not really enough to understand how computer systems are designed.



- Application - the software you run.
- Algorithm - the way that it is implemented, in mathematical level
- Programming language - the way that it is implemented for a computer.

Skip to main content

After the Programming language level, there is assembly. The advantage to assembly is that it is hardware independent and human readable. It is low level and limited to what the hardware *can* do, but it is a version of that that can be run on different hardware. It is much lower level. When you compile a program, it is translated to assembly. At this level, programs written in different level become indistinguishable. This has much lower level operations. We can do various calculations, but not things like compare. Things that were one step before, like assign become two, choose a memory location, then write to memory. This level of abstraction is the level of detail we will think about most. We'll look at the others, but spend much less time below here.

The instruction set architecture is, notice, where the line between software and hardware lives. This is because these are specific to the actual hardware, this is the level where there are different instructions if you have an Intel chip vs an Apple chip. This level reduces down the instructions even more specifically to the specifics things that an individual piece of hardware does and how they dit.

The microarchitecture is the specific circuits: networks of smaller individual components. Again, we can treat the components as blocks and focus on how they work together. At this level we still have calculations like add, multiply, compare, negate, and we can store values and read them. That is all we have at this point though. At this level there are all binary numbers.

The actual gates (components that implement logical operations) and registers (components that hold values) break everything down to logical operations. Instead of adding, we have a series of `and`, `nand`, `or`, and `xor` put together over individual bits. Instead of numbers, we have `registers` that store individual zeros or ones. In a modern digital, electrical computer, at this level we have to actually watch the flow of electricity through the circuit and worry about things like the number of gates and whether or not the calculations finish at the same time or having other parts wait so that they are all working together. We will see later that when we try to allow multiple cores to work independently, we have to handle these timing issues at the higher levels as well. However, a register and gate can be implemented in different ways at the device level.

The device (or transistor in modern electrical digital computers) level, is where things transition between analog and digital. The world we live in is actually all analog. We just pay attention to lots of things at a time scale at which they appear to be be digital. Over time devices have changed from mechanical switches to electronic transistors. Material science innovations at the physics level have improved the transistors further over time, allowing them to be smaller and more heat efficient. Because of abstraction, these changes could be plugged into new hardware without having to make any changes at any software levels. However, they do *enable* improvements at the higher levels.

9.6. Prepare for Next Class

note this is posted on 2023-10-05 because October 10th is a "Monday" for URI classes

1. Read the notes from October 5 and October 3. We will build on both of them on October 12. Make sure you have completed all of the steps in the github inclass repo from September too.
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 kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for much later, it does not go in the October 12 experience branch**

9.7. Review today's class

1. Read today's notes when they are posted.
2. Add to your software.md a section about if that project does or does not adhere to the unix philosophy.

Skip to main content

```
- which of the three methods for studying a system do you use most often when debugging?
- do you think using a different strategy might help you debug faster sometimes? why or why not?
```

9.8. More Practice

1. Add to your software.md a section about if that project does or does not adhere to the unix philosophy and why. You can see what badge it was previously assigned in and the instructions on the KWL file list.
2. create methods.md and answer the following:

```
- which of the three methods for studying a system do you use most often when debugging?
- which do you use when you are trying to understand something new?
- do you think the ones you use most often are consistently effective? why or why not? When do they work or n
- what are you most interested in trying that might be different?
```

9.9. Experience Report Evidence

Give examples of how you have used different ways of understanding things.

9.10. Questions After Today's Class

9.10.1. Why would we need to know about the lower layers of hardware and software if there are abstractions in place for us?

This is a great question!

Most of the time that is exactly the advantage to the abstractions, but you typically need to know about the things at least in the adjacent levels. Some times knowing about more layers can help you make better design choices or understand edge case behavior. We will see some of those throught the rest of the semester.

Another part of why we teach these things in a CS degree is so that you can make a more informed choice about what you want to specialize in later, too.

9.10.2. what an example of Instruction set architecture

the x86 isa is an example.

9.10.3. How to install `gh`?

from their docs

9.10.4. How often do software engineers touch machine code?

Software engineers is a very broad category. If you write drivers, that might be very different than if you write front ends.

9.10.5. how physics play into abstraction

The actual physics of how electricity works is the lowest level of abstraction, or most concrete/least abstract way to describe how a computer works.

9.10.6. why do brew and gh keep uninstalling from my terminal?

This might be a setting somewhere in your system that erases or resets things? If you find the answer to this and add it as an FAQ, that is worth a community badge.

9.10.7. Should I install the `gh` CLI?

You can, it's a little harder on Windows, but possible.

9.10.8. What is a GUI based tool?

GUI stands for graphical user interface. Everything on a modern compputer except the terminal is a GUI.

9.10.9. Is it useful to create models during the process of development, or is it mostly useful after the fact?

Models are often made before things are built. Sometimes they are refined after, but usually they are used to guide the development.

9.10.10. How deep should we really understand abstraction layers and how useful is it as a developer to know it?

We will see them again and they will come up at least implicitly in other classes as well. It is important to know that they exist and what they are for sure. How much you directly interact with them will vary though.

9.10.11. How will our view on memory change with abstraction as we get to the higher level classes?

We will talk about the physical device that implements memory later and you will learn more about it again in 411 if you take that.

10. How does git create 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 conceptually reinforce important concepts and practically 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.

Skip to main content

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.

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:

- get more practice with bash
- see a practical example of hashing
- reinforce through examples what a pointer does

10.1. Review to set the stage

Let's look at the github inclass repo.

```
cd Documents/inclass/systems/github-inclass-fa23-brownsarahm/
```

Recall: git stores important content in *files* that it uses like variables.

All of gits files, that is uses to perform the version control system actions are in the `.git` directory. Since this starts with a `.` it is "hidden" so to see it we use the `-a` option on `ls`.

```
ls -a
```

```
.                       LICENSE.md              helper_functions.py
..                      README.md               important_classes.py
.git                    about.md                setup.py
.github                 abstract_base_class.py  tests
API.md                  alternative_classes.py
CONTRIBUTING.md         docs
```

Inside there we can see the files:

```
ls .git
```

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

For example:

```
cat .git/HEAD
```

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

Skip to main content

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

```
042a42eb47c33ee43d793feb4d891a93e7460527
```

we also see the config file

```
cat .git/config
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "1-create-an-about-file"]
        remote = origin
        merge = refs/heads/1-create-an-about-file
[branch "fun_fact"]
        remote = origin
        merge = refs/heads/fun_fact
```

that stores information about the different branches and remotes.

```
ls -a
```

```
.                       LICENSE.md              helper_functions.py
..                      README.md               important_classes.py
.git                    about.md                setup.py
.github                 abstract_base_class.py  tests
API.md                  alternative_classes.py
CONTRIBUTING.md         docs
```

`.gitignore` is a file in the working direcotry that contains alist of files and patterns to not track.

```
cd ../tiny-book/
ls -a
```

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

```
cat .gitignore
```

Skip to main content

Remmeber the `.gitignore` file lives **outside** of the `.git` directory because it is intended to be manually edited by the user, instead of written to by the `git` program.

10.2. Creating a repo from scratch

We will start in the top level course directory.

```
cd ..
```

```
ls
```

```
fa23-kwl-brownsarahm            tiny-book
github-inclass-fa23-brownsarahm
```

Yours should also have your group repo, your messy folder, etc.

We can create an empty repo from scratch using `git init <path>`

Last time we used an existing directory like `git init .`

We will make a new directory for our repo:

```
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/
```

We can see what happened:

```
ls
```

```
fa23-kwl-brownsarahm            test
github-inclass-fa23-brownsarahm tiny-book
```

we see the new directory

then we can enter it

and then rename the branch

```
git branch -m main
```

As usual, we will look at the status

```
git status
```

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Notice that there are no commits, and no origin.

```
ls .git
```

```
HEAD            description     info            refs
config          hooks           objects
```

10.3. Searching the file system

We can use the bash command `find` to search the file *system* note that this does not search the **contents** of the files, just the names.

```
find .git/objects/
```

```
.git/objects/
.git/objects//pack
.git/objects//info
```

we have a few items in that directory and the directory itself.

We can limit by type, to only files with the `-type` option set to `f`

```
find .git/objects/ -type f
```

And we have no results. We have no objects yet. Because this is an empty repo

10.4. Creating Git Blob Objects Directly

There are 3 types:

- blob objects: the content of your files (data)

Skip to main content

- Commit Objects: stores information about the sha values of the snapshots

classDiagram class tree{ List: - hash: blob - string: type - string:file name } class commit{ hash: parent hash: tree string: message string: author string: time } class blob{ binary: contents } class object{ hash: name } object <|-- blob object <|-- tree object <|-- commit
All git objects are files stored with the name that is the hash of the content in the file

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` function 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.

We can put content into the stdout file with `echo`

```
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.

and we can use a pipe to connect std out of one command to stdin of the next command. Then we can use the `--stdin` option to tell git hash-object to read from there.

```
echo "test content" | git hash-object -w --stdin
```

We can break down this command:

- `git hash-object` would take the content you handed to it and merely return the unique key
- `-w` option tells the command to also write that object to the database
- `--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`

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we can check if it wrote to the directory.

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we see a file that it was supposed to have!

This file contains binary content that is hard to read. 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

Imp
pipes
we're
uses,
Pipes
comn

Skip to main content

`cat-file` requires *an* option, we have used 2 so far:

- `-p` is for pretty print
- `-t` is for type

We can then view the type of this object

```
git cat-file -t d670
```

```
blob
```

and its actual content

```
git cat-file -p d670
```

```
test content
```

This is the content that we put in, as expected.

10.4.1. Hashing a file

To hash content that corresponds to a file, we have to have a file first. We can use a redirect `>` to send content from echo to a different file.

```
echo "version 1" > test.txt
```

and store it, by hashing it

```
git hash-object -w test.txt
```

```
83baae61804e65cc73a7201a7252750c76066a30
```

we can look at what we have.

```
find .git/objects/ -type f
```

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Now this is the status of our repo.

classDiagram class d67046{ + "test content" +(blob) } class 83baae{ + Version 1 + (blob) }
We can check the type of files with `-t` and `git cat-file`

```
git cat-file -t 83ba
```

Skip to main content

```
blob
```

```
git cat-file -t d670
```

```
blob
```

We have hashed content, but our git status is still saying no commits and only one untracked file.

```
git status
```

```
On branch main

No commits yet

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

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

So far, even though we have hashed the object, git still thinks the file is untracked, because it is not in the tree and there are no commits that point to that part of the tree.

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.

Notice, that we only have one file in the working directory.

```
ls
```

```
test.txt
```

10.5. Writing a tree

We can write a tree with the `write-tree` command

```
git write-tree
```

```
4b825dc642cb6eb9a060e54bf8d69288fbee4904
```

we can see the object exists

```
find .git/objects/ -type f
```

Skip to main content

```
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

we can verify that it is a tree

```
git cat-file -t 4b82
```

```
tree
```

and look at the tree contents

```
git cat-file -p 4b82
```

but it is empty.

We have a file that was created, `git write-tree` did *something* but we told it to write a file without putting anything in the index, or staging area, that it reads from.

classDiagram class d67046{ +"test content" +(blob) } class 83baae{ +Version 1 +(blob) } class 4b825d{ +(tree) }
Now, we can add our file and its hashed object as it is to the index.

```
git update-index --add --cacheinfo 100644 \
 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

the `\` lets us wrap onto a second line.

- this the plumbing command `git update-index` updates (or in this case creates an index, 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

10.6. A staging and snapshot edge case

Let's check git status again:

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
```

Skip to main content

Now the file is staged.

Let's edit it further.

```
echo "version 2" >> test.txt
```

So the file has two lines

```
cat test.txt
```

```
version 1
version 2
```

Now check status again.

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

We added the first version of the file to the staging area, so that version is ready to commit but we have changed the version in our working directory relative to the version from the hash object that we put in the staging area so we *also* have changes not staged.

We can hash and store this version too.

```
git hash-object -w test.txt
```

```
0c1e7391ca4e59584f8b773ecdbbb9467eba1547
```

and verify a new object was created?

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

- plain content
- 2 versions of the file
- an empty tree

classDiagram class d67046{ +"test content" +(blob) } class 83baae{ +Version 1 +(blob) } class 4b825d{ +(tree) } class 0c1e73{ +Version 1 +Verson 2 +(blob) }
Again, let's check in

```
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 content does not change the status here at all, because `git status` only traces from the `HEAD` pointer, compares that commit's tree to the index and the current working directory. We have no commits still and the index has not changed by hashing the content and creating a blob for it.

10.7. Writing a tree

Now we can write a tre and it will have content.

```
git write-tree
```

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

Lets examine the tree, first check the type

```
git cat-file -t d832
```

```
tree
```

and now we can look at its contents

```
git cat-file -p d832
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

it tells us the mode of the single file is a regular file, that it is a blob, the hash where the file's content is snapshotted and the file name.

classDiagram class d67046{ +"test content" +(blob) } class 83baae{ +Version 1 +(blob) } class 4b825d{ +(tree) } class d8329f{ +blob: 83baae +filename: test.txt +(tree) } class 0c1e73{ +Version 1 +Verson 2 +(blob) } d8329f --|> 83baae
This only keeps track of th objects, there are also still the HEAD that we have not dealt with and the index.

Now two of our objects are linked! the tree points to the blob of the first version of the file. We are getting closer to what would happend when we make a commit with a porcelain command.

```
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
```

10.8. Creating a commit manually

We can echo a commit message through a pipe into the commit-tree plumbing function to commit a particular hashed object.

```
echo "first commit" | git commit-tree d8329
```

```
0ec6f39c964a3149d426ae45bef4385e14444b2c
```

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.

The above hash is the one I got during class, but if I rerun I will get a different hash even though I have the same name and e-mail because the time changed.

> **Important**
>
> You can find the hash of your commit object later by comparing to mine and getting the one that is unique or by checkign the type of all of your objects until you find the one that is a commit.

We can also look at its type

```
git cat-file -t 0ec6
```

Skip to main content

and we can look at the content

```
git cat-file -p 0ec6
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1697131751 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1697131751 -0400

first commit
```

Now we check the final status of our repo

```
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.

Important

Check that you also have 6 objects and 5 of them should match mine, the one you should not have is the `0ec6f` one but you should have a different hash for your commit.

classDiagram class d67046{ test content (blob) } class 83baae{ Version 1 (blob) } class 4b825d{ (tree) } class d8329f{ blob: 83baae filename: test.txt (tree) } class 0c1e73{ Version 1 Verson 2 (blob) } class 0ec6f{ tree d8329f author name commit time } d8329f --|> 83baae 0ec6f --|> d8329f
It still says no commits, even though we just verified that it has one.

10.9. Git references

Remember git status 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?

Notice, git said we have no commits yet even though we have written a commit.

Skip to main content

This is because the main branch does not *point* to any commit.

We can verify by looking at the `HEAD` file

```
cat .git/HEAD
```

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

and see that:

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

```
cat: .git/refs/heads/main: No such file or directory
```

which does not even exist!

we can see the objects though:

```
find .git/objects/ -type f
```

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//0e/c6f39c964a3149d426ae45bef4385e14444b2c
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

and we can create that file by putting the full hash of the commit into the file

```
echo 0ec6f39c964a3149d426ae45bef4385e14444b2c > .git/refs/heads/main
```

so that

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

shows the hash:

```
0ec6f39c964a3149d426ae45bef4385e14444b2c
```

Now we check status and we are all set!

```
git status
```

```
On branch main
Changes not staged for commit:
```

Skip to main content

```
        modified:   test.txt

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

10.10. Prepare for Next Class

1. Read the notes from October 12. We will build on these directly in the future. **You need to have the `test` repo with the same status for lab on 10/13 and class on 10/17** Make sure you have completed all of the steps in the github inclass repo from September too.
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. Record which IDE(s) you use, what tasks you do, what features you use, what extensions, etc. Store your notes in your kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for much later, it does not go in the October 17 experience badge branch**

10.11. Review today's class

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plubming and porcelain. The table should have two columns, one for each type of command. Each row should have one git plumbing command and at least one of the corresponding git porcelain command(s). Include two rows: `add` and `commit`.
2. Contribute to your group repo and review a classmate's contribution. Include a link to your contribution and review in your badge PR comment using markdown link syntax. Your contribution can be a short how to with a code excerpt or a resource. Include a link to your contribution and review in your badge PR comment using markdown link syntax:

```
[text to display](url/of/link)
```

10.12. More Practice

2. Read more details about git internals to review what we did in class in greater detail. Make a file gitplumbingdetail.md and create a visualization that is compatible with version control (eg can be viewed in plain text and compared line by line, such as table or mermaid graph) that shows the relationship between at least three porcelain commands and their corresponding plumbing commands.
3. 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.
4. Contribute to your group repo and review a classmate's contribution. Include a link to your contribution and review in your badge PR comment using markdown link syntax. Your contribution can be a short how to with a code excerpt or a resource. (view the raw version of this issue page for the git internals link above for an example)

10.13. Experience Report Evidence

Put your list of objects in your `test` repo into a file `test_repo_ojbects.txt` in your Experience report branch.

10.14.1. Can a pointer to an old file exist as an object in the git objects?

a tree will have a pointer to the previous version of the file yes.

10.14.2. Is there a manual way to do other commands like git push and git add?

We did git add today. We hashed the object and added the hash to the index. Push does have some other steps in it, those are in the practice badge.

10.14.3. Why you are allowed to create empty trees if it is basically a wasted file?

Because it does not hurt anything.

10.14.4. How are pipes and redirect different?

Pipes connect stdout to stdin, to connect commands. Redirect just changes the source.

10.14.5. When do people manually create git objects and commits?

Manually exploring is not a common scenario. These commands *exist* primarily for git itself, remember modular code gives more power.

We went through doing it manually to see and understand. You can end up in edge case scenarios in other ways, but using the plubming commands is the best way to to see that.

10.14.6. Why are these commands so easy to confuse and mix up with each other? I have a hard time remembering some of the longer commands.

Some of the commands we have used are detailed low frequency use commands that you do not need to memorize.

10.14.7. Is there a quicker way to find the onbject type if you get lost. Instead of going through each of them with the “git cat-file -t” command?

For this activity, you can copmare to mine. In a normal use scenario, you would not do this manually, so you would have some other way to know what commit or object you were looking for.

10.14.8. what are hashes?

A hash is a fixed length representation of variable lenght input. We will see more about them in the coming weeks.

10.14.9. What are trees?

Trees are one type of git object, that contains information about the directory at a given moment. It contains pointers to the blob obiects andinformation about each file.

Skip to main content

10.14.10. How do I change my inclass branch from master to main??

```
git branch -m main
```

10.14.11. What is the difference between the -t and -p flags? I noticed them while going through the comnmands.

these are options on `git cat-file`. They're defined above and at the link.

10.14.12. Would there be any reason to use these commands rather than relying on the porcelain commands?

Not if things are going well, but they could be used to try to figure out what went wrong if you find a repo in a weird state.

10.14.13. when would we use this over doing it on GitHub itself aside from offline?

We cannot run these detailed steps on github in browser. Recall, we do not even directly "add" files there. In browser, we have to commit in order to save a file.

10.14.14. Why does the file still have untracked changes?

We hashed the file, but did not put the second hash into the tree or create a second commiit.

11. What is a commit number?

11.1. Announcements

> **Important**
>
> follow the prepare for 10/19 to get announcements from the repo commit messages

11.2. 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.

Skip to main content

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

What are some ways a hash could 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)

11.3. 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.

Some sites are negligent and store passwords unencrypted, if your browser warns you about such a site, proceed with caution and definitely do not reuse a password you ever use. (you *should never* reuse passwords, but especially do not if there is a warning)

An attacker who gets one of those databases, cannot actually read the passwords, but they could build a lookup table. For example, “password” is a bad password because it has been hashed in basically every algorithm and then the value of it can be reversed. Choosing an uncommon password makes it less likely that your password exists in a lookup table.

```
echo "password" | git hash-object --stdin
```

11.4. 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.

11.4.1. Git hashes are cryptographic

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.

We can use the git hashing algorithm without writing to the repo too:

```
echo "what's up" | git hash-object --stdin
```

```
bd26a07a5f7b350d3bce48721cdc5a8c51f65306
```

```
echo "what's up " | git hash-object --stdin
```

```
1b0095f6628174ff6b4c9341e19230db33071fce
```

```
echo "what up " | git hash-object --stdin
```

```
d562044d1a61d1b86e3b15e70d6c7f5922e54498
```

```
echo "password" | git hash-object --stdin
```

```
f3097ab13082b70f67202aab7dd9d1b35b7ceac2
```

```
echo "passworksjfwklsjfd" | git hash-object --stdin
```

```
251c2d02f1fcbe5230648f4c386ba7d9afdd2cbe
```

```
echo "passworksjfwklsjfd" | git hash-object --stdin
```

```
251c2d02f1fcbe5230648f4c386ba7d9afdd2cbe
```

Not

I char
easie
outpu

Skip to main content

11.4.2. Git hashing algorithm

Git was originally designed to use SHA-1.

The SHA-1 Algorithm hashes content into a fixed length of 160 bits.

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

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.

Then the SHA-1 collision attack was discovered

Git switched to hardened HSA-1 in response to a collision.

> In that case it adjusts the SHA-1 computation to result in a safe hash. This means that it will compute the regular SHA-1 hash for files without a collision attack, but produce a special hash for files with a collision attack, where both files will have a different unpredictable hash. from.

they will change again soon

GitHub *uses* git, it is not an alternative implementation or a fork, so yes it will switch too. The developers at GitHub an other git hosts are among the most impacted by the change since they write code that directly interacts with git objects.

11.4.3. Remember we have many more git objects than commits

```
find github-inclass-fa23-brownsarahm/.git/objects -type f
```

```
github-inclass-fa23-brownsarahm/.git/objects/04/2a42eb47c33ee43d793feb4d891a93e7460527
github-inclass-fa23-brownsarahm/.git/objects/04/ab89e167ed77bc2a95710f69f68d91a6219471
github-inclass-fa23-brownsarahm/.git/objects/69/3a2b5b9ad4c27eb3b50571b3c93dde353320a1
github-inclass-fa23-brownsarahm/.git/objects/93/4c15dc2655c988c981d9a836783afebda77355
github-inclass-fa23-brownsarahm/.git/objects/5a/a1ed29b82e1cebb8527019b0e594ba71dda214
github-inclass-fa23-brownsarahm/.git/objects/5f/e5a9821625fad2cca4c500e497e6694132c303
github-inclass-fa23-brownsarahm/.git/objects/d7/6bc523443bda5a5daae2fe7fcfbf6fba71ae6d
github-inclass-fa23-brownsarahm/.git/objects/b3/78bd148e53dfa7195c58123362e40ae12ef3e7
github-inclass-fa23-brownsarahm/.git/objects/bc/281792d6ab62b153d7bf44f7985ec7cfc3b850
github-inclass-fa23-brownsarahm/.git/objects/ca/eacb503cf4776f075b848f0faff535671f2887
github-inclass-fa23-brownsarahm/.git/objects/ca/feca302e31c65139b4a5294356e1ea8595dcb1
github-inclass-fa23-brownsarahm/.git/objects/pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.rev
github-inclass-fa23-brownsarahm/.git/objects/pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.pack
github-inclass-fa23-brownsarahm/.git/objects/pack/pack-8631fedd908bc07c0b64786e9a83f5bf7a4de110.idx
github-inclass-fa23-brownsarahm/.git/objects/11/d53c24bb5d2bf2e3f645ef188f8bc75fa9c911
github-inclass-fa23-brownsarahm/.git/objects/45/fcb1dd311e5e45af759cb3627dca5f47f58f04
github-inclass-fa23-brownsarahm/.git/objects/75/6c4879c0447db20980f73a26bc2ba072e08a6d
```

```
github-inclass-fa23-brownsarahm/.git/objects/5c/8aaa9f2a129d551b8cb2cb294676f63c4af410
github-inclass-fa23-brownsarahm/.git/objects/65/e9e39935be8400ef12cc9003592f12244b50da
github-inclass-fa23-brownsarahm/.git/objects/3a/cf0fb1c2febd24561294bfb966e1ad1f033eb8
github-inclass-fa23-brownsarahm/.git/objects/98/96f7a7000a7b9d2fdb12047a141524358286c3
github-inclass-fa23-brownsarahm/.git/objects/37/0e04baf4f62d1e62f4949208bc5e4d33af5336
github-inclass-fa23-brownsarahm/.git/objects/6d/4dbd33860fceb9c87bd3c4509deff8cecb3f45
github-inclass-fa23-brownsarahm/.git/objects/39/f1c5eabb1458fa6cf9042611599b69665cf288
github-inclass-fa23-brownsarahm/.git/objects/55/56d17391aeeec9b2b86d2821c011d7ed5377aa
github-inclass-fa23-brownsarahm/.git/objects/b8/6eb90ba1ae5504edfcdc9ef8879e1c6d7a1b75
github-inclass-fa23-brownsarahm/.git/objects/b6/2d570421c3096d8c80c7df56357cdd3203fd3a
github-inclass-fa23-brownsarahm/.git/objects/ea/c84c8320a3ab4f37a441a332b828c45ecedcc9
github-inclass-fa23-brownsarahm/.git/objects/e1/82616690a91a8d0e363f4143e68dd9e136ccee
github-inclass-fa23-brownsarahm/.git/objects/e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
github-inclass-fa23-brownsarahm/.git/objects/8c/7cefb877c62a46a3b71c68a858c24075b379fe
github-inclass-fa23-brownsarahm/.git/objects/76/8dec80c5e0734476d476ae83376c9c786b6450
github-inclass-fa23-brownsarahm/.git/objects/2b/cb5d446129df427a1ce09e8ba658a5bb8ceca3
```

```
echo "passworksjfwklsjfd" | git hash-object --stdin
```

```
251c2d02f1fcbe5230648f4c386ba7d9afdd2cbe
```

```
python
```

```
Python 3.11.4 (main, Jul  5 2023, 09:00:44) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> 16*16
256
>>> exit()
```

```
python
```

```
Python 3.11.4 (main, Jul  5 2023, 09:00:44) [Clang 14.0.6 ] on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> len('251c2d02f1fcbe5230648f4c386ba7d9afdd2cbe')
40
>>> exit()
```

11.4.4. Workign with git hashes

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

Git log by default shows us the full length

```
cd github-inclass-fa23-brownsarahm/
```

```
git log
```

Skip to main content

```
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 13:26:59 2023 -0400

    begin organizing

commit d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 12:53:14 2023 -0400

    start organizng for real

commit bc281792d6ab62b153d7bf44f7985ec7cfc3b850
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 12:51:22 2023 -0400

    start organizing

commit 756c4879c0447db20980f73a26bc2ba072e08a6d (origin/fun_fact, fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:26:20 2023 -0400

    second fun fact

commit 768dec80c5e0734476d476ae83376c9c786b6450
```

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.

```
git log --abbrev-commit --pretty=oneline
```

```
042a42e (HEAD -> main, origin/main, origin/HEAD) begin organizing
d76bc52 start organizng for real
bc28179 start organizing
756c487 (origin/fun_fact, fun_fact) second fun fact
768dec8 Update about.md
6d4dbd3 add fun fact
5c8aaa9 Merge pull request #5 from introcompsys/add-name
65e9e39 (origin/add-name) closes #2
caeacb5 Merge pull request #4 from introcompsys/1-create-an-about-file
693a2b5 (origin/1-create-an-about-file, 1-create-an-about-file) create and complete about file closes #1
6a12db0 Add online IDE url
cfe32e5 Initial commit
```

```
git checkout bc28179
```

```
Note: switching to 'bc28179'.

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
```

Then we can check and the `HEAD` is moved to that commit

```
git log --abbrev-commit --pretty=oneline
```

```
bc28179 (HEAD) start organizing
756c487 (origin/fun_fact, fun_fact) second fun fact
768dec8 Update about.md
6d4dbd3 add fun fact
5c8aaa9 Merge pull request #5 from introcompsys/add-name
65e9e39 (origin/add-name) closes #2
caeacb5 Merge pull request #4 from introcompsys/1-create-an-about-file
693a2b5 (origin/1-create-an-about-file, 1-create-an-about-file) create and complete about file closes #1
6a12db0 Add online IDE url
cfe32e5 Initial commit
```

```
git checkout main
```

```
Previous HEAD position was bc28179 start organizing
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

```
git cat-file -p 6d4dbd3
```

```
tree 370e04baf4f62d1e62f4949208bc5e4d33af5336
parent 5c8aaa9f2a129d551b8cb2cb294676f63c4af410
author Sarah M Brown <brownsarahm@uri.edu> 1695143214 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1695143214 -0400

add fun fact
```

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

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 uses 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.

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.

Not all systems are place based, for example Roman numerals. 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.

Then

- III = 1+1+1 = 3
- IV = -1 + 5 = 4
- VI = 5+1 = 6
- XLIX = -10 + 50 -1 +10 = 49.

This feel hard because it is unfamiliar

11.7. Different Bases

11.7.1. Decimal

Skip to main content

$$10 = 10 * 1 + 1 * 0$$

$$22 = 10 * 2 + 1 * 2$$

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 different characters.

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) operations and then digital computers use on and off in their circuits.

We 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.

$$10 => 2 * 1 + 1 * 0 = 2$$

so this 10 in binary is 2 in decimal

$$1001 => 8 * 1 + 4 * 0 + 2 * 0 + 1 * 1 = 9$$

11.7.3. Octal

Is base 8. This too has history in other cultures, not only in computer science. It is rooted in cultures that counted using the spaces *between* fingers instead of counting using fingers.

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.

$$10 => 8 * 1 + 1 * 0 = 8$$

so 10 in octal is 8 in decimal

$$401 => 64 * 4 + 8 * 0 + 1 * 1 = 257$$

Skip to main content

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.

11.7.4. Hexadecimal

base 16, commonin 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

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.

11.8. Prepare for Next Class

1. Review the course website releases to get familiar with what content appears there
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 may not work on the rendered website, but will work on issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue.
4. Use `git log` to view recent updates to the course website from the October 12 class to the October 17 class release
5. review the open PR about a build badge if you are intersted in that option and comment if anything is unclear or approve if it is helpful.

11.9. Review today's class

1. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) that are current. Describe them in `numbers.md`. Include links to your sources and be sure that the sources are trust worthy.
2. Calculate the maximum number of git objects that a repo can have without needing 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`. Describe the scenario that would get you to that number of objects with the maximum *or* minimum number of commits in terms of what types of objects would exist. 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.*

11.10. More Practice

1. Read about the Learn more about the SHA-1 collision attach
2. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below gittransition.md
3. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) that are current. Describe them in numbers.md
4. Calculate the maximum number of git objects that a repo can have without needing 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 many files could you have? How could you

use with porcelain commands, not atypical cases with plubming commands. *If you get stuck, outline what you know and then request a review.*

11.11. gittransition

```
## transition questions

1. Why make the switch?
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?
```

11.12. Experience Report Evidence

none

11.13. Questions After Today's Class

11.13.1. What might the password table looklike?

You can check your own passwords against a collection of breech-revealed passwords, or download such a table at haveibeenpwned

Conceptually it could be as simmpel as 2 columns:

password	hash
…	…

11.13.2. Will we have to keep creating bigger and bigger number systems when computers get too fast at brute forcing hashes?

Most systems lock people out after a few tries to try to contol this, but potentially.

This is also why we typically have 2 factors now.

11.13.3. Are the letters used in hexadecimal representation global? Like if all countries that may not use english have to use the english characters in hex A,B,C,D,E,F?

The letters we use are not limited to English. The English alphabet descends largely from the latin alphabet, as do many other European languages. Further, since globalization, even some languages that historically had their own alphabets have adopted the English alphabet or one that is very similar.

Historically people may have used others for hex, but in programming contexts we all use the same latin alphabet .

Fundamentally, counting can happen in many different ways. What we talked about today is abstractions and representations that have been developed over 1000s of years over the entire planet.

People are complicated and vary a alot

11.13.5. What is the main difference between hash tables and hash functions?

Strictly speaking, a hash function could be used to produce a hash table, a hash table is sort of a special case.

A hash function tells how to produce the hash from the input. A hash table could assign hashes to content arbitrarily.

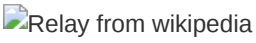
11.13.6. If the word “password” hash has been easily reverised, couldn’t that just be done with every word in a language?

Yes, it could, that is why we have rules about what makes a good password. That increases the possible options.

There are a lot of words in a language, but it’s a lot less than the total number of valid pass words, which do not have to be meaningful and typically can be any mix of letters in upper (26) or lower (+26) case, numbers (+10), or special characters (maybe another +20 or more). This means a password of length d can be $26 * 2 + 10 + 20 = 82^d$ options. Even for just length 2 there are then 6724 possible options. For length 4 there are over 45 million. So once you allow long passwords that are 30+ characters, it becomes really hard to store all of those options.

11.13.7. What exactly is a relay?

We will see this later when we talk more about hardware. It is an electrically operated switch.


Relay from wikipedia

the wikipedia article also has an animation

11.13.8. Are there any hashing algorithms that use numerical systems greater than base 16?

The hashing algorithm does not necessarily select a specific base. For example the SHA-1 algorithm is defined using a series of bitwise and mathematical operations on the binary representation of the content:

- split the messages into batches of length 512 bits
- break into 32 bit words
- xor
- and
- comparisons (if statments)
- bit shifts
- add
- or

you can see the full psuedocode for the algorithm

Any base can be used to write out the strings. 16 is the highest commonly used, but for example the wikipedia article on SHA-1 uses both base 64 and 16 in the examples

11.13.9. How would a developer create a function to hash passwords and store them securely?

In general, a developer would not develop a new hash function, you wouls use an existing one. Increasingly, authentication standards or using other authentication tools are common.

11.14. Questions that are a good explore badge

> **Where to collate these?**
>
> Ideas for where/how to collate questions like this are welcome and worth a community badge

11.14.1. What are good practices for adding authentication to my program?

This could be a good explore badge or potentially even a build on this topic

11.14.2. I want to learn more about encryption

If you make this a little more specific it could be a great explore badge. You could try out an encryption library or read and write a summary or something else that makes sense to you.

11.14.3. Hashing and Quantum computing

- Once quantum computing is more prominent, will hashing algorithms change because now it will can be both 0 and 1?
- Will current hashing algorithms be safe against quantum computing?

A good answer to this would rely on multiple sources and some independent reasoning.

> **Where to collate these**
>
> Ideas for where/how to collate questions like this are welcome and worth a community badge

12. Why did we learn the plubming commands?

You will not typically use them on a day to day basis, but they are a good way to see what happens at the interim steps and make sure that you have the right understanding of what git does.

A **correct** understanding is essential for using more advanced features

While there is of course some content that we want you to know after this course, my goal is also to teach you process, by modeling it

No one will every know all of the things but you can be fast or slow at finding answers.

And you can find correct answers, incorrect answers, or looks-okay-but-you-will-regret-this-later answers.

> **Important**
>
> My goal is that you get good at *quickly* finding **correct** answers.

12.1. When is the blob created?

First lets add some conent to revise a file

```
echo "a sentence" >> about.md
```

and then use git status to check the repo

```
git status
```

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

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

at this point file is changed, but it has only been changed in the working directory.

git does not run in the background automatically.

We can use `find` to inspect as we have, with `wc -l` to count the objects.

```
find .git/objects/ -type f | wc -l
```

```
      36
```

we start with 36

Now we add the file to the staging area

```
git add .
```

and count the objects again

```
find .git/objects/ -type f | wc -l
```

Skip to main content

```
      37
```

and we see it increased. So the blob object is created when we stage the file.

This is how we can have a file that is both staged for commit and not staged for commit.

Right now we have a version of the file that has been previously hashed and committed at various times, and the version that is both edited locally *and* hashed and stored for a future commit.

Next we will edit the file again.

```
echo "a sentence agian" >> about.md
```

and check the status again

```
git status
```

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

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

Now there are 3 versions of this file encountered by `git status` (active, there are actually more in past commits):

- the version in the last commit
- the version staged for commit
- the version in the working directory

12.2. How does git track file names?

Let's trace it out? First we will look at our recent commits

```
git log
```

```
commit 042a42eb47c33ee43d793feb4d891a93e7460527 (HEAD -> main, origin/main, origin/HEAD)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 13:26:59 2023 -0400

    begin organizing

commit d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 12:53:14 2023 -0400

    start organizng for real
```

```
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Sep 21 12:51:22 2023 -0400

    start organizing

commit 756c4879c0447db20980f73a26bc2ba072e08a6d (origin/fun_fact, fun_fact)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:26:20 2023 -0400

    second fun fact
```

Now we can use the first four characters (the minimum) or more if needed to be unique to look at that commit object.

```
git cat-file -p 042a
```

```
tree 11d53c24bb5d2bf2e3f645ef188f8bc75fa9c911
parent d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d
author Sarah M Brown <brownsarahm@uri.edu> 1695317219 -0400
committer Sarah M Brown <brownsarahm@uri.edu> 1695317219 -0400

begin organizing
```

Here, in the commit, we see that there are no file names. So we can keep tracing, we will look at the tree object from this commit.

```
git cat-file -p 11d5
```

```
040000 tree 95b60ce8cdec1bc4e1df1416e0c0e6ecbd3e7a8c    .github
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    API.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    CONTRIBUTING.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    LICENSE.md
100644 blob 8c7cefb877c62a46a3b71c68a858c24075b379fe    README.md
100644 blob b62d570421c3096d8c80c7df56357cdd3203fd3a    about.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    abstract_base_class.py
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    alternative_classes.py
040000 tree 9896f7a7000a7b9d2fdb12047a141524358286c3    docs
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    helper_functions.py
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    important_classes.py
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    setup.py
040000 tree 45fcb1dd311e5e45af759cb3627dca5f47f58f04    tests
```

Here in the tree we see that each line includes the name from the working directory of the file that the blob came from or the folder that is represented by that additional tree.

Let's check a blob object for completeness.

```
git cat-file -p b62d
```

```
Sarah Brown
2027
- i skied competitively in high school
- i started at URI in 2020
- I went to Northeastern
```

Note here that there is no mention of the file name, the file name is **only** in the tree.

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

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

12.3. What is the index?

The staging area/index

```
cat .git/index
```

```
fk#.github/workflows/create_issues.ymle.??.m?0e.??.m?0??d????⛲??CK?)?wZ???S?API.mde.??.Me.??.M??e????⛲??CK
LICENSE.mde.??.?t?e.??.?t???g?????|?w?*F??h?X?@u?y?     README.mde1[?5??e1[?5??o?8????0f?DTj0|????,?tabout.m
docs3 0
????
{?/?z$5???tests4 0
E???1^E?u??b}?_G??.github1 1
??
  ?????????>z?workflows1 0
nm?```



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

```
fatal: Not a valid object name .git/index
```

```
git status
```

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

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

12.4. Why does it show the help

Important

This was a real question from class and is a good question!

Most command line tools are designed to show you the help, not only an error when you use it incorrectly.

```
git stats
```

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

The most similar command is
        status
```

12.5. Git References

Note

I have commented out for now where the inspection of git tags did not go as expected. I hope to fill this in with the answer and notify you later.

12.6. Making a git tag

We will work in the github-inclass repo for this.

```
cd ../github-inclass-fa23-brownsarahm/
```

The `git tag` command works a lot like `git branch`. With no parameters it lists the current tags. With options it can also create or manage them.

```
git tag
```

If we provide a name for it, we can create a lightweight tag

```
git tag v1
```

A lightweight tag is just a pointer to a commit.

```
git tag
```

```
v1
```

We can see the tag is added to the annotation on the commits with git log.

Skip to main content

```
git log --pretty=oneline
```

```
042a42eb47c33ee43d793feb4d891a93e7460527 (HEAD -> main, tag: v1, origin/main, origin/HEAD) begin organizing
d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d start organizng for real
bc281792d6ab62b153d7bf44f7985ec7cfc3b850 start organizing
756c4879c0447db20980f73a26bc2ba072e08a6d (origin/fun_fact, fun_fact) second fun fact
768dec80c5e0734476d476ae83376c9c786b6450 Update about.md
6d4dbd33860fceb9c87bd3c4509deff8cecb3f45 add fun fact
5c8aaa9f2a129d551b8cb2cb294676f63c4af410 Merge pull request #5 from introcompsys/add-name
65e9e39935be8400ef12cc9003592f12244b50da (origin/add-name) closes #2
caeacb503cf4776f075b848f0faff535671f2887 Merge pull request #4 from introcompsys/1-create-an-about-file
693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file) create and c
6a12db0035e7c73772f7b2348b80dd0bfb3a2a2e Add online IDE url
cfe32e5066921ad876d8a2c74b1fcb00c99b1cc7 Initial commit
```

```
git tag --help
```

We can also add tags at past commits by passing a commit hash.

```
git tag v0 65e9
```

and see it the same way.

```
git log --pretty=oneline
```

```
042a42eb47c33ee43d793feb4d891a93e7460527 (HEAD -> main, tag: v1, origin/main, origin/HEAD) begin organizing
d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d start organizng for real
bc281792d6ab62b153d7bf44f7985ec7cfc3b850 start organizing
756c4879c0447db20980f73a26bc2ba072e08a6d (origin/fun_fact, fun_fact) second fun fact
768dec80c5e0734476d476ae83376c9c786b6450 Update about.md
6d4dbd33860fceb9c87bd3c4509deff8cecb3f45 add fun fact
5c8aaa9f2a129d551b8cb2cb294676f63c4af410 Merge pull request #5 from introcompsys/add-name
65e9e39935be8400ef12cc9003592f12244b50da (tag: v0, origin/add-name) closes #2
caeacb503cf4776f075b848f0faff535671f2887 Merge pull request #4 from introcompsys/1-create-an-about-file
693a2b5b9ad4c27eb3b50571b3c93dde353320a1 (origin/1-create-an-about-file, 1-create-an-about-file) create and c
6a12db0035e7c73772f7b2348b80dd0bfb3a2a2e Add online IDE url
cfe32e5066921ad876d8a2c74b1fcb00c99b1cc7 Initial commit
```

These create files in the .git direcotry with the hash in them

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

```
v0      v1
```

```
cat .git/refs/tags/v0
```

```
65e9e39935be8400ef12cc9003592f12244b50da
```

Tags can be used with other git commands like `git log` to show only a subset of the commits.

Skip to main content

```
git log --pretty=oneline v0..v1
```

```
042a42eb47c33ee43d793feb4d891a93e7460527 (HEAD -> main, tag: v1, origin/main, origin/HEAD) begin organizing
d76bc523443bda5a5daae2fe7fcfbf6fba71ae6d start organizng for real
bc281792d6ab62b153d7bf44f7985ec7cfc3b850 start organizing
756c4879c0447db20980f73a26bc2ba072e08a6d (origin/fun_fact, fun_fact) second fun fact
768dec80c5e0734476d476ae83376c9c786b6450 Update about.md
6d4dbd33860fceb9c87bd3c4509deff8cecb3f45 add fun fact
5c8aaa9f2a129d551b8cb2cb294676f63c4af410 Merge pull request #5 from introcompsys/add-name
```

Note

I used that to generate a list of changes in the class 11 release

12.7. What if I have changes I do not want to lose *or* commit yet

We want to pusht the tag, but we cannot push right now:

```
git push
```

```
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
 ! [rejected]        main -> main (fetch first)
error: failed to push some refs to 'https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git'
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

because we have our local and remote main branches out of sync

```
git pull
```

```
remote: Enumerating objects: 9, done.
remote: Counting objects: 100% (9/9), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 6 (delta 3), reused 6 (delta 3), pack-reused 0
Unpacking objects: 100% (6/6), 745 bytes | 82.00 KiB/s, done.
From https://github.com/introcompsys/github-inclass-fa23-brownsarahm
   042a42e..4202c16  main       -> origin/main
Updating 042a42e..4202c16
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
```

Pulling did not work either.

However, technically the tags can be pushed anyway, even without pushing the rest of our files.

```
git push --tags
```

```
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-inclass-fa23-brownsarahm.git
 * [new tag]         v0 -> v0
 * [new tag]         v1 -> v1
```

We can then see them in the browser:

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

```
Opening github.com/introcompsys/github-inclass-fa23-brownsarahm in your browser.
```

12.8. Using git to debug

> **Note**
>
> I am going to revise this later, to understand for now, refer to the git book chapter

If you find a bug in your code and you do not know when it was introduced `git bisect` can help you perform a binary search over your commits to find the first commit that introduced the bug.

```
git bisect start
```

```
status: waiting for both good and bad commits
```

It takes 2 inputs as separate calles after you start it a known bad commit and a known good commit.

We will say that we know our current commit is bad.

```
git bisect bad
```

```
status: waiting for good commit(s), bad commit known
```

And that `v0` tag was good.

> **Note**
>
> This is a good use of tags!

```
git bisect good v0
```

```
Bisecting: 3 revisions left to test after this (roughly 2 steps)
error: Your local changes to the following files would be overwritten by checkout:
        about.md
Please commit your changes or stash them before you switch branches
```

Now that working directory update that we made comes back to get in the way.

We have the option to commit or stash.

12.9. What if I want to keep changes, but not commit them yet?

Stash creates git objects of content that you can recover later. It works like a first-in-first-out queue by default and is a conceptually separate set of content from the commit history (though it is still stored in the `.git/` directory)

```
git stash
```

```
Saved working directory and index state WIP on main: 042a42e begin organizing
```

once we stash it, our working directory is clean and we can go back to debugging

12.10. Binary searchiing with bisect

Now we can tell it the good commit again

```
git bisect good v0
```

```
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[768dec80c5e0734476d476ae83376c9c786b6450] Update about.md
```

What git does here is checks out the commit in the middle of the commits we labeled good and bad, this allows us to check whatever we were looking for and then we have to label each commit it checks out for us as good or bad.

```
git bisect good
```

```
Bisecting: 1 revision left to test after this (roughly 1 step)
[bc281792d6ab62b153d7bf44f7985ec7cfc3b850] start organizing
```

If we say it is good, it takes the next commit to be halfway between this commit and the one labeled bad.

```
git bisect bad
```

```
Bisecting: 0 revisions left to test after this (roughly 0 steps)
[756c4879c0447db20980f73a26bc2ba072e08a6d] second fun fact
```

When we label a commit bad, it checks out the commit between the current commit and the closest one labeled good

Here we do whatever we need to do to figure out if the currently checked out version is good or bad.

And label this comit

Skip to main content

```
git bisect bad
```

```
756c4879c0447db20980f73a26bc2ba072e08a6d is the first bad commit
commit 756c4879c0447db20980f73a26bc2ba072e08a6d
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Sep 19 13:26:20 2023 -0400

    second fun fact

 about.md | 2 ++
 1 file changed, 2 insertions(+)
```

In this case, once we label it bad, now git has enough informtion to tell us what commit was the first bad one and what changes were made in this commit.

This wil help us decide how to fix it.

```
git status
```

```
HEAD detached at 756c487
You are currently bisecting, started from branch 'main'.
  (use "git bisect reset" to get back to the original branch)

nothing to commit, working tree clean
```

“detatched” HEAD means that the head pointer is set to a specific commit rather than a branch.

When we are done, we can use reset to to back to main and then apply our fix.

```
git bisect reset
```

```
Previous HEAD position was 756c487 second fun fact
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)
```

12.11. What about the stashed content

Now that we are on main with a clean working directory it is a good time to pull the remote changes.

```
git pull
```

```
Updating 042a42e..4202c16
Fast-forward
 README.md                 | 37 +++++++++++++++++++-------------------
 about.md => docs/about.md |  0
 2 files changed, 19 insertions(+), 18 deletions(-)
 rename about.md => docs/about.md (100%)
```

```
git stash list
```

```
stash@{0}: WIP on main: 042a42e begin organizing
```

and then `apply` to apply them

```
git stash apply
```

```
On branch main
Your branch is up to date with 'origin/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:   docs/about.md

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

it updates the working directory but not the index by default.

If we re stage the file

```
git add .
```

and and make more edits

```
echo "another edit" >> docs/about.md
```

so that we have both staged and unstaged changes again

```
git status
```

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

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

and stash this status

```
git stash
```

```
Saved working directory and index state WIP on main: 4202c16 fix readme to remove >
```

Skip to main content

```
git status
```

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

nothing to commit, working tree clean
```

we now have 2 stashes

```
git stash list
```

```
stash@{0}: WIP on main: 4202c16 fix readme to remove >
stash@{1}: WIP on main: 042a42e begin organizing
```

we can apply the on in 0 to both the workign directory and the staging area with the `--index` option.

```
git stash apply --index
```

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

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

12.12. Prepare for Next Class

2. If you have not already or right before class, review the GitHub Action files in your KWL repo and make note of what if any syntax in there is unfamilar. (note that link may not work on the rendered website, but will work on issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue.
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.

12.13. Review today's class

1. Use git bisect to find the first bad commit in the toy bug repo, save the command history and the bad commit hash to git_debug.md
2. Create tagtypeexplore.md with the template below. Determine how many of the tags in the course website are annotated vs lightweight using `git cat-file` to inspect the tags and the `git tag` help to determine which tags are of each type.

```
# Tags
```

```
## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

12.14. More Practice

1. Use your github-inclass repo to create a scenario where you can fix a problem using a git command from the patching or debugging section of the docs. Create a log of what you did using the history or git log into a file gitstory.md. If you have a project in another class or another badge in this class that causes you to use one in a real scenario, that can count. If not, for example, you could create a "bug" and then use bisect to find it.
2. Create tagtypes.md with the template below. Include an experiment that shows which if either type of tag creates a new git object. There are two types, try creating one of each a lightwight tag (provide only the tag name- what we did in class) and an annotated (provide a name and a message with `-m`). Determine many of the tags in the course website are annotated vs lightweight using what you learned about how tags are represented and `git cat-file` to see which is which.

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Comparing tag types

<!-- include your experiment terminal history  and interpretation -->

## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

12.15. Experience Report Evidence

save your command history to `2023-10-19-log.txt` and put that file in your KWL repo

12.16. Questions After Today's Class

> **Note**
>
> due to scheduling issues this will be late today

13. How can I use bash to automate things?

13.1. What is bash again?

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

- `mv`
- `cd`
- `pwd`
- `ls`

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.

We can describe this relationship with a concept map. Each shape corresponds to a *concept* and the arrows are labeled to describe the relationship between the two concepts. This can be a helpful way of writing out how you understand things.

flowchart TD shell --> |is accessed through| terminal bash --> |is instance of| shell zsh --> |is instance of| shell git --> |is a program accessed though | shell
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

We can add this information to the diagram:

flowchart TD shell --> |is accessed through| terminal bash --> |is instance of| shell zsh --> |is instance of| shell git --> |is a program accessed though | shell subgraph shell pl[programming language] i[command interpreter] end

13.2. What does this definiton of bash mean?

Today we will start from the main course directory.

Note on macOS when I run `bash` to use the bash shell, it reminds me that MacOS prefers a different shell.

```
    Last login: Sat Oct 21 17:29:01 on ttys080

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.
```

We can have different shells in the same terminal in some cases, and all of the different shells on the system have access to the same file system.

Then I will move to my systems dir

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

and look at files

```
ls
```

```
fa23-kwl-brownsarahm            github-inclass-fa23-brownsarahm
fall2023                        test
```

13.3. Working from an Example

> **Important**
>
> Here I used `cd fa23-kwl-brownsarahm` and `gh repo view --web` to go to my KWL repo in the browser and then `cd ..` to go back

In your KWL repo the `.github/workflows` directory has several `.yml` files that are github action files.

These files are yaml, with key:value pairs. The values of the `run` key are commands that are passed to bash. So these are a place where you can see examples of what I did to look for particular syntax.

Learning from examples is a common strategy in CS and how you can learn new technology especially in a job, so today in class, we modeled that by looking for different syntax in the action files and building up what those things do.

13.4. Variables in Bash

We can create variables like `VARNAME=value`.

```
NAME='Sarah'
```

notice that there are **no spaces** around the `=`. spaces in bash separate commands and options, so they cannot be in variable declarations.

A common mistake is to put a space around the `=` sign, this is actually considered **good style** in many other languages.

```
NAME ='Sarah'
```

```
-bash: NAME: command not found
```

In bash, however, this creates an error. When there is a space after `NAME`, `bash` tried to interpet `NAME` as a bash command, but then it does not find it, so it gives an error.

```
NAME='Sarah'
```

We can use variables with a `$` before the variable name.

```
echo $NAME
```

```
Sarah
```

> **Important**
>
> This variable is local, in memory, to the current terminal window, so if we open a separate window and try `echo $NAME` it

Skip to main content

```
echo $NAME
```

```
Sarah
```

We can see that it does not create any files by comparing ls now to what it was before:

```
ls
```

```
fa23-kwl-brownsarahm            github-inclass-fa23-brownsarahm
fall2023                        test
ghic                            tiny-book
```

```
ls -a
```

```
.                               ghic
..                              github-inclass-fa23-brownsarahm
fa23-kwl-brownsarahm            test
fall2023                        tiny-book
```

We can, however use the variable at different working directories. So if we move

```
cd github-inclass-fa23-brownsarahm/
```

and `echo` agian

```
echo $NAME
```

```
Sarah
```

it still works!

The `$` is essential syntax for recalling variables in bash. If we forget it, it treats it as a literal

```
echo NAME
```

```
NAME
```

so we get the variable **name** out instead of the variable **value**

We'll go back up a level now:

```
cd ..
```

Skip to main content

13.5. Bash Loops

We can also make loops like

```
for loopvar in list
do
# loop body
echo $loopvar
done
```

So for example we can loop over some names

```
for name in Sarah Amoy Marcin
> do
> echo $name
> done
```

A few important things, to make note of:

- loop variable does not need to be an iterator. the loop variable here `name` takes each value from a list (`Sarah Amoy Marcin`)
- lists is bash are defined with no brackets and no commas `Sarah Amoy Marcin` is a list
- we start the loop body with `do` and close it with `done` these are like the `{` and `}` in some langauges.
- after we start the loop with a line starting with `for` bash continues that “command” by putting `>` at the start of each line until we end the loop with `done` and then it runs.

And we see it echos each name:

```
Sarah
Amoy
Marcin
```

Note that the loop variable is not scope limited to the loop. After the loop it still has the same value of the last time through the loop.

```
echo $name
```

```
Marcin
```

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 add another name in

```
for name in Sarah Amoy Marcin Osman; do echo $name; done
```

```
Sarah
Amoy
Marcin
Osman
```

the output is like before, but with one more

Imp

If you
error,
this c
butyo

Skip to main content

```
echo $name
```

```
Osman
```

and this time `Osman` is the name left in the loop variable later.

13.6. Nesting commands

There are examples of this in your KWL repo actions. If we want the output of a command to be in a variable this is how to do it.

To do this, let's move to a folder with some files in it.

```
cd github-inclass-fa23-brownsarahm/
```

here we have a number of files in this working directory:

```
ls
```

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

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
```

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

the `$()` tells bash to run that command first and then hold its output as a variable for use elsewhere

For example, in your `getassignment.yml` I have a line like this:

```
pretitle="prepare-"$(sysgetbadgedate --prepare)
```

Skip to main content

```
echo $pretitle
```

```
prepare-2023-10-26
```

We can also modify our previous loop with an option:

```
for FILE in $(ls -a); do echo $FILE; done
```

```
.
..
.git
.github
API.md
CONTRIBUTING.md
LICENSE.md
README.md
abstract_base_class.py
alternative_classes.py
docs
helper_functions.py
important_classes.py
setup.py
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
- the `>` work again like the loop and the `\` we did in the past

this if does nothing though, because `docs` is a directory not a file.

If we switch it, we get output:

```
if test -f API.md ; then echo "file"; fi
```

```
file
```

Skip to main content

13.8. Script files

We can put our script into a file

```
nano filecheck.sh
```

and check the content of the file

```
cat filecheck.sh
```

```
for file in $(ls)
do
        if test -f $file
        then
                echo $file
        fi
done
```

and run it with `bash <filename>`

```
bash filecheck.sh
```

```
API.md
CONTRIBUTING.md
LICENSE.md
README.md
abstract_base_class.py
alternative_classes.py
filecheck.sh
helper_functions.py
important_classes.py
setup.py
```

And this is as `ls` except without the folders:

```
ls
```

```
API.md                  abstract_base_class.py  helper_functions.py
CONTRIBUTING.md         alternative_classes.py  important_classes.py
LICENSE.md              docs                    setup.py
README.md               filecheck.sh            tests
```

Important

We also looked at the structure and some examples of the action files in the KWL repo and course website.

I recommend the docs, starting from the quickstart and the understand section

13.9. Using GH

Skip to main content

```
cd ../fall2023/
```

```
gh pr list
```

```
Showing 5 of 5 open pull requests in introcompsys/fall2023

#15  Create mermaid.md            introcompsys:trevmoy-pat...  about 5 days ago
#12  sample proposals for 212...  212build                     about 10 days ago
#11  Community-typos              jacksonarich:main            about 16 days ago
#10  Hash notes                   NoahV17:hash_notes           about 18 days ago
#5   Added question to 9 26 n...  NoahV17:added-question-t...  about 23 days ago
```

Grep can search content, so with a pipe we can use it to filter results.

```
gh pr list | grep sample
```

```
12      sample proposals for 212 docs build     212build        OPEN    2023-10-14 17:02:14 +0000 UTC
```

13.10. Terminal coloring and customizing your prompt.

Your path settings, aliases, and the your prompt can be set with a file.

On MacOS the file is called `.bash_profile` and is stored at your `~` home directory. If you use `zsh` instead of bash there is a different file, but it is similar.

On linux it is `.bashrc` it should be the same in WSL on Windows.

> **Community Badge opportunity**
>
> If you use windows, can you PR to update these notes with what file is used and which terminal/shell you are using (bash via GitBash, powershell, bash in WSL, etc.)

```
cat ~/.bash_profile
```

I have some aliases set, this way when I use Python, I can use the `python` command instead of `python3`.

the `PS1` variable describes the main default prompt, according to the spec. this tool can help you customize your prompt

```
alias pip=pip3
alias python=python3

export PS1="\u@\W $ "

# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/brownsarahm/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
```

```
        . "/Users/brownsarahm/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/brownsarahm/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

13.11. Prepare for Next Class

1. Read the 3 bulleted examples of why use a cluster from HPC carpentry.
2. Read this discussion of why using a remote server

13.12. Review Badge

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that you can run in your group repo to generate a file with a list of all of your PRs and. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md in your KWL repo.

13.13. Practice Badge

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that you can run in your group repo to generate a file with a list of all of your PRs and PR reviews (you may go in and assign yourself to these PRs to make this easier). Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md in your KWL repo.

13.14. Experience Report Evidence

13.15. Questions After Today's Class

13.15.1. How often will we be writing scripts for this class?

Just a few times to get practice and exposure, but they will come back in 412 and are handy in general.

13.15.2. Are bash scripts commonly used today in real world applications or is it something that was used more in the 70s?

They are still super common. GitHub actions can be written using bash scripting as a strategy. Lots of systems have build scripts.

The thing we would not do that often that we did today was write it from nano instead of a full IDE.

13.15.3. What is the scope of loop variables?

13.15.4. When would I use bash instead of a different programming language?

When you need to do things that are file operations or other os level actions, it can be faster to run if it is in bash.

13.15.5. What specific tasks do developers like to use bash to automate?

build processes, file manipulations, git operations. Anything even just a few steps that would be repetitive and you would have to do often.

13.15.6. do people write full programs in bash using things like nano or directly into the terminal?

Yes! Some people especialy people who started programming a long time ago still do.

More often, using a terminal text editor is a handy tool to use on occasion, but not for every day work. We will work on a remote server, which is an occasion you would use that, on Thursday.

13.15.7. Is there a way to save files that you create in the shell?

yes, today we saved the filecheck.sh file.

13.15.8. what else can we do with bash?

Its mostly for automating things and processing files or outputs.

It *can* do anything though.

13.15.9. Does running a bash script with `bash `` remove the need to add #!/bin/bash to the top of it and then make it executable prior to running?

Yes. by running it like `bash <filename` it works just like passing those lines in the file to bash.

Alternatively you can put the shebang `#!/bin/bash` at the top to tell the computer what shell to use to run it and then treat it as an executable.

We will learn about making it executable permissions wise on Thursday.

13.15.10. Is it possible to create functions in bash scripts?

Yes! see the GNU bash docs on functions

13.15.11. Where would we see bash the most as a developer? Would we use this language more for an embedded swe role or is it something that's present in every type of programming job.

repetitive things that you do not want to do manually. You may write more or less, but it's always a handy tool to have.

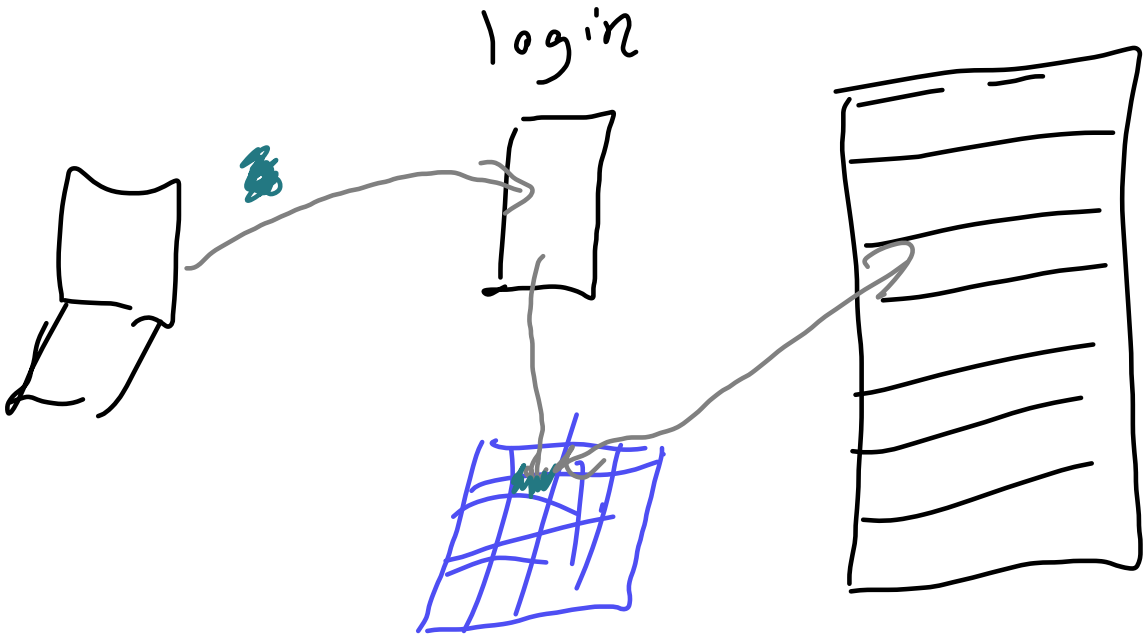
13.15.12. Where could I learn more about writing something similar to the workflows we saw?

The github actions documentation has instructions with how tos, API like documentation, and examples.

14. How can I work on a remote server?

Today we will connect to a remote server and learn new bash commands for working with the *content* of files.

14.1. What are remote servers and HPC systems?



14.2. Connecting to Seawulf

We connect with secure shell or `ssh` from our terminal (GitBash or Putty on windows) to URI's teaching High Performance Computing (HPC) Cluster Seawulf.

Our login is the part of your uri e-mail address before the @.

```
ssh -l brownsarahm seawulf.uri.edu
```

When it logs in it looks like this and requires you to change your password. They configure it with a default and with it past expired.

Note

This block is sort of weird, because it is interactive terminal. I have rendered it all as output, but broken it down to separate chunks to add explanation.

Skip to main content

```
The authenticity of host 'seawulf.uri.edu (131.128.217.210)' can't be established.
ECDSA key fingerprint is SHA256:RwhTUyjWLqwohXiRw+tYlTiJEbqX2n/drCpkIwQVCro.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
```

Follow the instruction to type `yes`

I will tell you how to find your default password if you missed class (do not want to post it publicly). Comment on your experience report PR to ask for this information.

```
Warning: Permanently added 'seawulf.uri.edu,131.128.217.210' (ECDSA) to the list of known hosts.
brownsarahm@seawulf.uri.edu's password:
```

it does not show charachters when you type your password, but it works when hyou press enter

Then it requires you to change your password

```
You are required to change your password immediately (root enforced)
WARNING: Your password has expired.
You must change your password now and login again!
```

To change, it asks for you current (default) password first,

Important

You use the default password when prompted for your username's password. Then again when it asks for the `(current) UNIX password:`. Then you must type the same, new password twice.

Choose a new password you will remember, we will come back to this server

```
Changing password for user brownsarahm.
Changing password for brownsarahm.
(current) UNIX password:
```

then the new one twice

```
New password:
Retype new password:
passwd: all authentication tokens updated successfully.
Connection to seawulf.uri.edu closed.
```

after you give it a new password, then it logs you out and you have to log back in.

```
brownsarahm@~ $ ssh -l brownsarahm seawulf.uri.edu
```

wehn you log in it shows you information about your recent logins.

We have logged into our home directory which is empty

Skip to main content

```
[brownsarahm@seawulf ~]$ pwd
```

```
/home/brownsarahm
```

Notice that the prompt says `uriusername@seawulf` to indicate that you are logged into the server, not working locally.

14.3. Downloading files

`wget` allows you to get files from the web.

```
[brownsarahm@seawulf ~]$ wget http://www.hpc-carpentry.org/hpc-shell/files/bash-lesson.tar.gz
```

```
--2022-03-08 12:58:09--  http://www.hpc-carpentry.org/hpc-shell/files/bash-lesson.tar.gz
Resolving www.hpc-carpentry.org (www.hpc-carpentry.org)... 104.21.33.152, 172.67.146.136
Connecting to www.hpc-carpentry.org (www.hpc-carpentry.org)|104.21.33.152|:80... connected.
HTTP request sent, awaiting response... 200 OK
Length: 12534006 (12M) [application/gzip]
Saving to: ‘bash-lesson.tar.gz’

100%[======================================>] 12,534,006  4.19MB/s   in 2.9s

2022-03-08 12:58:12 (4.19 MB/s) - ‘bash-lesson.tar.gz’ saved [12534006/12534006]
```

Note that this is a reasonably sized download and it finished very quickly. This is because the download happened **on the remote server** not your laptop. The server has a high quality hard-wired connection to the internet that is very fast, unlike the wifi in our classroom.

This is an advantage of using a remote system. If your connection is slow, but stable enough to connect, you can do the work on a different computer that has better connection.

Now we see we have the file.

```
[brownsarahm@seawulf ~]$ ls
```

```
bash-lesson.tar.gz
```

we see the new file!

14.4. Unzipping a file on the command line

This file is compressed.

We can use `man tar` to see the manual aka man file of the `tar` program to learn how it works. You can also read man files online from GNU where you can choose your format, this page shows the full version.

```
[brownsarahm@seawulf ~]$ tar -xvf bash-lesson.tar.gz
```

```
dmel-all-r6.19.gtf
dmel_unique_protein_isoforms_fb_2016_01.tsv
gene_association.fb
SRR307023_1.fastq
SRR307023_2.fastq
SRR307024_1.fastq
SRR307024_2.fastq
SRR307025_1.fastq
SRR307025_2.fastq
SRR307026_1.fastq
SRR307026_2.fastq
SRR307027_1.fastq
SRR307027_2.fastq
SRR307028_1.fastq
SRR307028_2.fastq
SRR307029_1.fastq
SRR307029_2.fastq
SRR307030_1.fastq
SRR307030_2.fastq
```

This command uses the `tar` program and:

- `v` makes it verbose (to output all that info while it works)
- `x` makes it extract
- `f` option accepts the file name to work on

We can see what it did with `ls`

```
[brownsarahm@seawulf ~]$ ls
```

```
bash-lesson.tar.gz                          SRR307026_1.fastq
dmel-all-r6.19.gtf                          SRR307026_2.fastq
dmel_unique_protein_isoforms_fb_2016_01.tsv  SRR307027_1.fastq
gene_association.fb                         SRR307027_2.fastq
SRR307023_1.fastq                           SRR307028_1.fastq
SRR307023_2.fastq                           SRR307028_2.fastq
SRR307024_1.fastq                           SRR307029_1.fastq
SRR307024_2.fastq                           SRR307029_2.fastq
SRR307025_1.fastq                           SRR307030_1.fastq
SRR307025_2.fastq                           SRR307030_2.fastq
```

14.5. Using a compute note

14.6. Working with large files

One of these files, contains the entire genome for the common fruitfly, let's take a look at it:

```
[brownsarahm@seawulf ~]$ cat dmel-all-r6.19.gtf
```

```
X       FlyBase gene    19961297        19969323        .       +       .       gene_id "FBgn0031081"; gene_s
2L      FlyBase 3UTR    782825  782885  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
```

Skip to main content

> **Warning**
>
> this output is truncated for display purposes

We see that this actually take a long time to output and is way tooo much information to actually read. In fact, in order to make the website work, I had to cut that content using command line tools, my text editor couldn't open the file and GitHub was unhappy when I pushed it.

> **Note**
>
> I initially used `head` and `tail` to pick out the excerpts of the terminal output that I needed, but this year I reused what I had done before

For a file like this, we don't really want to read the whole file but we do need to know what it's strucutred like in order to design programs to work with it.

`head` lets us look at the first 10 lines.

```
[brownsarahm@seawulf ~]$ head dmel-all-r6.19.gtf
```

```
X       FlyBase gene    19961297        19969323        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase mRNA    19961689        19968479        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase 5UTR    19961689        19961845        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19961689        19961845        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19963955        19964071        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19964782        19964944        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19965006        19965126        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19965197        19965511        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19965577        19966071        .       +       .       gene_id "FBgn0031081"; gene_s
X       FlyBase exon    19966183        19967012        .       +       .       gene_id "FBgn0031081"; gene_s
```

We use the `man` file for `head` to look at its options.

Historically there were only the single character options, but these are limited (you can only have 52) and cryptic. They were good though when you only could type and had minimal memory. the longer ones are good because they are more descriptive.

The long versions are better to use in scripts because they are easier for someone else to read. The single character versions save time when you are working interactively.

We can use the `-n` parameter to change the number.

And, tails shows the last few.

```
[brownsarahm@seawulf ~]$ tail dmel-all-r6.19.gtf
```

which in this case looks mostly the same

```
2L      FlyBase exon    782124  782181  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase exon    782238  782441  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase exon    782495  782885  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
```

Skip to main content

```
2L      FlyBase CDS     782124  782181  .       +       1       gene_id "FBgn0041250"; gene_symbol "Gr21a";
2L      FlyBase CDS     782238  782441  .       +       0       gene_id "FBgn0041250"; gene_symbol "Gr21a";
2L      FlyBase CDS     782495  782821  .       +       0       gene_id "FBgn0041250"; gene_symbol "Gr21a";
2L      FlyBase stop_codon      782822  782824  .       +       0       gene_id "FBgn0041250"; gene_symbol "
2L      FlyBase 3UTR    782825  782885  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a";
```

We can also see how much content is in the file `wc` give a word count and with its `-l` parameter gives us the number of lines.

```
[brownsarahm@seawulf ~]$ wc -l dmel-all-r6.19.gtf
```

```
542048 dmel-all-r6.19.gtf
```

Over five hundred forty thousand lines is a lot.

How can we get the number of lines in each of the `.fastq` files?

```
[brownsarahm@seawulf ~]$ wc -l *.fastw
wc: *.fastw: No such file or directory
```

note that in my typo, it tells me no files matched my pattern.

```
[brownsarahm@seawulf ~]$ wc -l *.fastq
   20000 SRR307023_1.fastq
   20000 SRR307023_2.fastq
   20000 SRR307024_1.fastq
   20000 SRR307024_2.fastq
   20000 SRR307025_1.fastq
   20000 SRR307025_2.fastq
   20000 SRR307026_1.fastq
   20000 SRR307026_2.fastq
   20000 SRR307027_1.fastq
   20000 SRR307027_2.fastq
   20000 SRR307028_1.fastq
   20000 SRR307028_2.fastq
   20000 SRR307029_1.fastq
   20000 SRR307029_2.fastq
   20000 SRR307030_1.fastq
   20000 SRR307030_2.fastq
  320000 total
```

when it does work, we also get the total.

We can use redirects as before to save these to a file:

```
[brownsarahm@seawulf ~]$ wc -l *.fastq > linecounts.txt
```

```
[brownsarahm@seawulf ~]$ cat linecounts.txt
   20000 SRR307023_1.fastq
   20000 SRR307023_2.fastq
   20000 SRR307024_1.fastq
   20000 SRR307024_2.fastq
   20000 SRR307025_1.fastq
   20000 SRR307025_2.fastq
   20000 SRR307026_1.fastq
   20000 SRR307026_2.fastq
   20000 SRR307027_1.fastq
```

```
  20000 SRR307029_1.fastq
  20000 SRR307029_2.fastq
  20000 SRR307030_1.fastq
  20000 SRR307030_2.fastq
 320000 total
```

We can also search files, without loading them all into memory or displaying them, with `grep`:

```
[brownsarahm@seawulf ~]$ grep Act5c dmel-all-r6.19.gtf
```

```
[brownsarahm@seawulf ~]$ grep mRNA dmel-all-r6.19.gtf
```

this output a lot, so the output is truncated here

```
X       FlyBase mRNA    19961689        19968479        .       +       .       gene_id "FBgn0031081"; gene_s
2L      FlyBase mRNA    781276  782885  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
```

and we can combine `grep` with `wc` to count occurences.

```
[brownsarahm@seawulf ~]$ grep mRNA dmel-all-r6.19.gtf | wc -l
34025
```

14.7. Feedback

> **Note**
>
> will fill in later

14.8. Prepare for Next Class

1. Read about connection protocols in general and specifically https and ssh. Wikipedia is a good source to start from,and use additional sources to verify anything you find confusing. Be sure you have the basic terminology down and bring questions to class. Plan to check off your questions as they are answered during class on Tuesady and then submit others in your experience reflection.

14.9. Review today's class

> **Important**
>
> This is an integrative 2x badge.

integrative 2: The PR title must be `review-2023-10-26` verbatim to get the double credit

create a `vocab-quiz.md` file with 10 mutliple choice questions that cover topics from at least 5 different class sessions. Each

should check that a person understand the key terms of the first half of the course. For each option explain why it is/not correct in a way that would help clarify someone's confusion if they had picked that answer instead of the correct answer.

**For this badge you must request a review from `@brownsarahm` **

Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
- [x] correct answer marked with x
- [ ] another wong

---

[section title](link/to/notes)
- explanation for first wrong
- explanation for second wrong
- key point about correct
- explantion for third wrong

---

Next question
```

14.10. Practice

> **Important**
>
> This is an integrative 2x badge.

integrative 2: The PR title must be `practice-2023-10-26` verbatim to get the double credit

create a `midterm.md` file in your kwl repo with 10 mutliple choice questions that cover topics from at least 5 different class sessions. Each question should have 4 options, 1 correct and 3 that represent a reasonable, but incorrect idea someone may have. All 10 questions should check understanding of key *concepts*, not only terminology or the name of a command. For each option explain why it is/not correct in a way that would help clarify someone's confusion if they had picked that answer instead of the correct answer. Each solution block must contain a link to the applicable class notes. **For this badge you must request a review from `@brownsarahm` **

Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
- [x] correct answer marked with x
- [ ] another wong

---

[section title](link/to/notes)
- explanation for first wrong
- explanation for second wrong
```

```
---

Next question
```

14.11. Experience Report Evidence

No specific files.

14.12. Questions After Today's Class

14.12.1. Will we be incorporating remote server use with coding at all in this class? Maybe some bash scripts?

Yes. Some in lab tomorrow and more after we learn building C code.

14.12.2. When would we want to use as many resources as possible but the reomte server would not be appropriate because we still need to connect to it?

This is an interesting question, but I can not come up with such a scenario.

14.12.3. What is a real use case of using a remote machine to do computing?

My research students, as do Professor Daniels' use a server to run their code because the problems are too big to do on a laptop.

ML works that way in industry too. As does real genomics research both in universities and in pharma companies.

14.12.4. How do I set up SSH keys for secure access to a remote Git server in Git Bash?(If possible)

We will set up ssh keys for server access in class next week and you will then be assigned to add them to git from GitHub docs.

14.12.5. Is it expensive to have a remote server?

At face value, yes, but they're typically shared costs. It might cost $10-50k to add a unit to the server, but then each use is not.

You could look up billing for AWS, MSFT Azure, or Google Cloud to see hourly use rates on their servers.

Also, the cost is compared to people's salary being spent waiting around. Even $50k, that lasts for only 3 years, could be worth it, if you consider it saves time of people who each have a salary over over$ 100k/ year.

14.12.6. how do we set up our own remote server?

badge.

14.12.7. What are other large remote servers that are free to use, and are there unsafe servers?

Most are not really free. Some give free access to students, many give free trials.

Unsafe would be like insecure in this sense that the data is not secure.

14.12.8. Would it be worthwhile to look into using a remote server for synchronizing school work between my laptop and my desktop (work beyond the scope of git/GitHub)?

That's not the point of an HPC or server like we used today. That is a good use for GitHub, or a cloud service that is file-focused like DropBox.

14.12.9. What's the point of Seawulf?

To learn about using remote servers. It's separate from the research servers so that we doing low priority class work do not slow down people doing research. The research ones are also funded with grant dollars and Seawulf is not.

14.12.10. What would be an instance that we may need to use a remote system for in school?

in CSC411 there is a homework server so that all students have the same environment.

14.12.11. How exactly computer intensive is graphics and pretty shapes, such that it is a necessity to cut them out for maximum computer resources available?

This is a good explore badge topic to monitor over a bit of time how much of your compute resources are being used by visualization vs the actual computation.

You could do a small experiment where you print vs do not and time how long the code takes. This could be done in any language.

14.13. Questions after this topic in previous semesters

14.13.1. What is meant by the term "gene symbol" ?

For the purpose of this class it is just a bit of the content in the file. For completeness, since the file is genomic data is is a name for the gene at that line.

14.13.2. When would I use these remote servers most often?

Remote servers are used for production code, and for large computations in any sort of science setting or machine learning setting.

How does it version track and store files?

It is a regular unix system with a standard file system. If we want to track versions, we use git. (or another version control system)

14.13.4. Will the files on our seawulf, still be available after exiting.

Yes, but not forever. Seawulf is a teaching server that makes no promise of backing up or keeping your data forever.

14.13.5. What are .fastq and .gz files?

.gz is a zip file, .fastq is a plain text output of a sequencing tool.

14.13.6. Can we connect to other servers other than the URI server?

Yes, if you have credentials, the ssh works basically the same way. We will learn one more thing that will give you more ability to use other servers next week.

14.13.7. What is the point of a remote server?

More compute power.

14.13.8. Do the files download directly to my computer or to the remote server?

The file goes to the system where the command is run, in this case, it went to the the server.

14.13.9. where is the ssh server downloaded on my computer?

ssh is the *connection protocol* or the set of rules by which our information is sent.

The server is another computer, you were using that computer *through* your local terminal.

We did not create any files on your local system today.

14.13.10. Is access to the chmod command restricted on computers and servers?

on a file by file basis typically, eys.

14.13.11. Can I access the remote files using vscode and write code?

Theoretically yes, but a more typical workflow would be to edit large files locally running vscode on your system then send the code to the remote server to run it. You might send one file at a time directly there or push from your local system to for example GitHub and then from GitHub on the server.

14.13.12. Will I work with files more like this or locally as a software engineer?

Maybe not this type specifically, but you will almost always access/store a large file at some point. You will also likely access/store a

14.13.13. for files that are impossible to open with text editors because they’re too long, is it possible to edit parts of the file just through terminal commands?

Yes! This is a good explore badge as well.

14.13.14. How can I zip from tthe terminal?

`zip file file` see the man

14.13.15. Are you able to run a website through a terminal?

You could launch a web server from a terminal. You can also launch a browser.

For he course website when I build the pdf, version a bash script launches a browser, uses the browsers print to pdf function and closes the browser.

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.

> **Tip**
>
> You c
> on yo

Minimum Rows

```
# KWL Chart


<!-- replace the  _ in the table or add new rows as needed -->

| Topic | Know | Want to Know | Learned |
| ------| ------- | ------ | ------- |
| Git | _ | _ | _ |
| GitHub | _ | _ | _ |
| Terminal | _ | _ | _ |
| IDE | _ | _ | _ |
| text editors | _ | _ | _ |
|file system | _ | _ |_ |
|bash | _ | _ | _ |
|abstraction | _ | _ | _ |
|programming languages | _ | _ | _ |
|git workflows | _ | _ | _ |
| git branches | _ | _ | _ |
| bash redirects | _ | _ | _ |
```

```
| documentation | _ | _ | _ |
| templating | _ | _ | _ |
|bash scripting | _ | _ | _ |
| developer tools | _ | _ | _ |
| networking | _ | _ | _ |
|ssh | _ | _ | _ |
| ssh keys | _ | _ | _ |
|compiling | _ | _ | _ |
| linking   | _ | _ | _ |
| building | _ | _ | _ |
| machine representation  | _ | _ | _ |
| integers   | _ | _ | _ |
| floating point  | _ | _ | _ |
|logic gates | _ | _ | _ |
| ALU | _ | _ | _ |
| binary operations | _ | _ | _ |
| memory | _ | _ | _ |
| cache | _ | _ | _ |
| register | _ | _ | _ |
| clock | _ | _ | _ |
| Concurrency | _ | _ | _ |
```

Required Files

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

Team Repo

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. Most days there will be specific additional activities and questions to answer. These should be in your KWL repo. 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.

2023-09-07

related notes

Activities:

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart (on a branch for this badge).
3. review git and github vocabulary (include link in your badge PR)
4. Post an introduction to your classmates on our discussion forum

related notes

Activities:

1. review notes after they are posted, both rendered and the raw markdown include links to each in your badge PR
2. 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

2023-09-14

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.
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.

2023-09-19

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.

2023-09-21

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 `fall2023/.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. What happens and what GitHub concept that we have not used yet might fix it? see your `vocab-` repo for a list of key github concepts. (answer in the `terminal_review.md`)
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 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`
5. Find your team's repository. It will have a name like `fa23-team#` where `#` is a number 1-4. Join the discussion on that repo about naming your team. Link to your comment directly in your PR for this badge (use the 3 dots menu to get the comment specific URL).

```
# Terminal File moving reflection
1. How was this activity overall? Did this get easier toward the end?
2. When do you think that using the terminal will be better than using your GUI file explorer?
3. What questions/challenges/ reflections do you have after this exercise?

## Commands used
```

2023-09-26

related notes

Activities:

1. create an issue on your group repo for a tip or cheatsheet item you want to contribute. Make sure that your contribution does not overlap with one that amemb
2. clone your group repo.
3. work offline and add your contribution and then open a PR
4. review a class mate's PR.
5. 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.

2023-09-28

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. Add `docs.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.

Skip to main content

2023-10-03

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 each tree and blob in that tree to the same file. 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.

2023-10-05

related notes

Activities:

1. Read today's notes when they are posted.
2. Add to your software.md a section about if that project does or does not adhere to the unix philosophy.
3. create methods.md and answer the following:

```
- which of the three methods for studying a system do you use most often when debugging?
- do you think using a different strategy might help you debug faster sometimes? why or why not?
```

2023-10-12

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: plubming and porcelain. The table should have two columns, one for each type of command. Each row should have one git plumbing command and at least one of the corresponding git porcelain command(s). Include two rows: `add` and `commit`.
2. Contribute to your group repo and review a classmate's contribution. Include a link to your contribution and review in your badge PR comment using markdown link syntax. Your contribution can be a short how to with a code excerpt or a resource. Include a link to your contribution and review in your badge PR comment using markdown link syntax:

```
[text to display](url/of/link)
```

2023-10-17

related notes

1. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) that are current. Describe them in `numbers.md`. Include links to your sources and be sure that the sources are trust worthy.
2. Calculate the maximum number of git objects that a repo can have without needing 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`. Describe the scenario that would get you to that number of objects with the maximum *or* minimum number of commits in terms of what types of objects would exist. 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.*

2023-10-19

related notes

Activities:

1. Use git bisect to find the first bad commit in the toy bug repo, save the command history and the bad commit hash to git_debug.md
2. Create tagtypeexplore.md with the template below. Determine how many of the tags in the course website are annotated vs lightweight using `git cat-file` to inspect the tags and the `git tag` help to determine which tags are of each type.

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

2023-10-24

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that you can run in your group repo to generate a file with a list of all of your PRs and. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md in your KWL repo.

Prepare for the next class

These tasks are not always based on things that we have already done. Sometimes they are to have you start thinking about the topic that we are *about* to cover. Getting whatever you know about the topic fresh in your mind in advance of class will help what we do in class stick for you when we start.

The correct answer is not as important for these activities as it is to do them before class. We will build on these in class. These are evaluated on completion only, but we may ask you questions or leave comments if appropriate, in that event you should reply and

2023-09-12

Activities:

1. (for lab) Read the syllabus section of the course website carefully and explore the whole course website
2. Bring questions about the course to lab
3. (for class) 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?

2023-09-14

Activities:

1. Find the glossary page for the course website, link it below. Review the terms for the next class: shell, terminal, bash, git, GitHub.
2. Check your kwl repo before class and see if you have recieved feedback, reply or merge accordingly.
3. Make sure you have a working environment, see the list in the syllabus, including `gh` CLI if you use mac`. Use the discussions to ask for help.
4. Sign up for announcements

2023-09-19

Activities:

1. Reply below with any questions you have about using terminals so that you can bring it up in class
2. Be prepared to compare and contrast bash, shell, terminal, and git.
3. (optional) If you like to read about things before you do them, read about merge conflicts. If you prefer to see them first, read this after.

2023-09-21

Activities:

1. Read the notes from 2023-09-19 carefully
2. Examine an open source project on GitHub. Answer the reflection questions below in `software.md` in your kwl repo on a branch that is linked to this issue. You do not need to make the PR, we will work with this in class on 2023-09-21.

```
## Software Reflection

1. Link and title of the project
1. What types of files are there that are not code?
1. What different types of code files are in the project? Do they serve different goals?
1. Is it all in one language or are there multiple languages?
1. Are there files that are configurations or settings?
1. Is there help for developers? users? which support is better?
1. What type of things are in the hidden files? who would need to see those files vs not?
```

Skip to main content

- pandas
- numpy
- GitHub CLI
- Rust language
- vs code
- Typescript
- Swift
- Jupyter book

2023-09-26

Activities:

1. Review your software.md file from last prepare
2. Review the notes from 2023-09-21
3. 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)
4. Update your `.github/workflows/experienceinclass.yml` file as to add another paramter to the last step (`Create Pull request`) `reviewers: <ta-gh-name>` where `<ta-gh-name>` is the github user name of the TA is in your group. You can see your group on the organiation teams page named like “Fall 2023 Group X”. Do this on a branch for this issue.

2023-09-28

Activities:

1. Read the 2023-09-26 notes when posted
2. 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`
3. Take a few minutes to think about a time you were debugging or pay attention the next time you debug something. Make note of how you think about the problem, how do you decide what to check? how do you decide what to try? what information do you look for?

2023-10-03

Activities:

1. Follow up on your grade plan PR and self reflection that you complete in lab

2023-10-05

Activities:

1. Review the notes from past classes
2. Think through and make some notes about what you have learned about design so far. Try to answer the questions below in

```
- What past experiences with making decisions about or 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 you would describe the design of command line tools vs other GUI based too
```

2023-10-12

Activities:

note this is posted on 2023-10-05 because October 10th is a "Monday" for URI classes

1. Read the notes from October 5 and October 3. We will build on both of them on October 12. Make sure you have completed all of the steps in the github inclass repo from September too.
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 kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for much later, it does not go in the October 12 experience branch**

2023-10-17

Activities:

1. Read the notes from October 12. We will build on these directly in the future. **You need to have the `test` repo with the same status for lab on 10/13 and class on 10/17** Make sure you have completed all of the steps in the github inclass repo from September too.
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. Record which IDE(s) you use, what tasks you do, what features you use, what extensions, etc. Store your notes in your kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for much later, it does not go in the October 17 experience badge branch**

2023-10-19

Activities:

1. Review the course website releases to get familiar with what content appears there
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 may not work on the rendered website, but will work on issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue.
4. Use `git log` to view recent updates to the course website from the October 12 class to the October 17 class release
5. review the open PR about a build badge if you are intersted in that option and comment if anything is unclear or approve if it is helpful.

2023-10-24

Activities:

2. If you have not already or right before class, review the GitHub Action files in your KWL repo and make note of what if any syntax in there is unfamilar. (note that link may not work on the rendered website, but will work on issues)
3. Use quote reply or edit to see how I made a relative path to a location within the repo in this issue.
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.

2023-10-26

Activities:

1. Read the 3 bulleted examples of why use a cluster from HPC carpentry.
2. Read this discussion of why using a remote server

More Practice Badges

> **Note**
>
> these are listed by the date they were *posted*

More practice exercises are a chance to try new dimensions of the concepts that we cover in class.

> **Note**
>
> Activities will appear here once the semester begins

2023-09-07

related notes

Activities:

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart (on a branch for this badge).
3. review git and github vocabulary be sure to edit a file and make an issue or PR (include link in your badge PR)
4. Post an introduction to your classmates on our discussion forum

2023-09-12

related notes

Activities:

1. review notes after they are posted, both rendered and the raw markdown versions. Include links to both views in your badge PR comment

2. read Chapter 1, “Decoding your confusion while coding” in The Programmer’s Brain add a file called brain.md to your kwl repo that summarizes your thoughts on the chapter and how, if at all, it changes how you think about debugging and learning to program.
3. map out your computing knowledge and add it to your kwl chart repo in a file called prior-knowledge-map.md. Use mermaid syntax, to draw your map. GitHub can render it for you including while you work using the preview button.
4. Read more about version control in general and add a “version control” row to your KWL chart with all 3 columns filled in.

2023-09-14

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.
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.

2023-09-19

related notes

Activities:

1. Create a merge conflict in your github in class repo 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 ide_merge_conflict.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)
2. Learn about GitHub forks and more about git branches(you can also use other resources)
3. add `branches-forks.md` to your KWL repo and describe how branches work, what a fork is, and how they relate to one another. If you use other resources, include them in your file as markdown links.

2023-09-21

related notes

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on.

1. Update your KWL chart with any learned items.
2. Get set up so that you can pull from the introcompsyss/fall2023 repo and push to your own fork of the class website by cloning the main repo, then forking it and adding your fork as an additional remote. Append the commands used and the contents of your `fall2023/.git/config` to a terminalpractice.md (hint: `history` outputs recent commands and redirects can work with any command, not only echo). Based on what you know so far about forks and branches, what advantage does this setup provide? (answer in the `terminal_practice.md`)
3. **lab** Organize the provided messy folder (details will be provided in lab time). Commit and push the changes. Clone that repo locally.
4. For extra practice, re/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.
5. Find your team's repository. It will have a name like `fa23-team#` where `#` is a number 1-4. Join the discussion on that repo about naming your team. Link to your comment directly in your PR for this badge (use the 3 dots menu to get the comment specific URL).

```
# Terminal File moving reflection
1. How was this activity overall Did this get easier toward the end?
2. How was it different working on your own computer compared to the Codespace from?
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 codesp
```

2023-09-26

related notes

Activities:

1. Find a resource/reference that helps explain a topic related to the course that you want to review. Make sure that your contribution does not overlap with one that another member is going to post by viewing other issues before you post your issue. Create an issue for your planned item.
2. Clone your group repo.
3. Work offline to add your contribution and then open a PR. Your reference review should help a classmate decide if that reference material will help them understand better or not. It should summarize the material and it's strengths/weaknesses.
4. Complete a peer review of a class mate's PR. Use inline comments for any minor corrections, provide a summary, and either approve or request changes.
5. 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.

2023-09-28

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.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.

2023-10-03

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. Find the hash of the blob object that contains the content of your gitislike.md file and put that in the comment of your badge PR for this badge.

```
## 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 you
```

2023-10-05

related notes

Activities:

1. Add to your software.md a section about if that project does or does not adhere to the unix philosophy and why. You can see what badge it was previously assigned in and the instructions on the KWL file list.
2. create methods.md and answer the following:

```
- which of the three methods for studying a system do you use most often when debugging?
- which do you use when you are trying to understand something new?
- do you think the ones you use most often are consistently effective? why or why not? When do they work or n
- what are you most interested in trying that might be different?
```

2023-10-12

related notes

2. Read more details about git internals to review what we did in class in greater detail. Make a file gitplumbingdetail.md and create a visualization that is compatible with version control (eg can be viewed in plain text and compared line by line, such as table or mermaid graph) that shows the relationship between at least three porcelain commands and their corresponding plumbing commands.
3. 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.
4. Contribute to your group repo and review a classmate's contribution. Include a link to your contribution and review in your badge PR comment using markdown link syntax. Your contribution can be a short how to with a code excerpt or a resource. (view the raw version of this issue page for the git internals link above for an example)

2023-10-17

related notes

Activities:

1. Read about the Learn more about the SHA-1 collision attach
2. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below gittransition.md
3. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) that are current. Describe them in numbers.md
4. Calculate the maximum number of git objects that a repo can have without needing 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 many files could you have? 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.*

gittransition

```
## transition questions

1. Why make the switch?
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?
```

2023-10-19

related notes

Activities:

1. Use your github-inclass repo to create a scenario where you can fix a problem using a git command from the patching or

project in another class or another badge in this class that causes you to use one in a real scenario, that can count. If not, for example, you could create a “bug” and then use bisect to find it.

2. Create tagtypes.md with the template below. Include an experiment that shows which if either type of tag creates a new git object. There are two types, try creating one of each a lightwight tag (provide only the tag name- what we did in class) and an annotated (provide a name and a message with `-m`). Determine many of the tags in the course website are annotated vs lightweight using what you learned about how tags are represented and `git cat-file` to see which is which.

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Comparing tag types

<!-- include your experiment terminal history  and interpretation -->

## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

2023-10-24

related notes

Activities:

1. Update your KWL Chart learned column with what you’ve learned
2. Write a bash script that you can run in your group repo to generate a file with a list of all of your PRs and PR reviews (you may go in and assign yourself to these PRs to make this easier). Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md in your KWL repo.

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

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.

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.

> **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

Skip to main content

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 solv

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build? wi

### 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.

Skip to main content

1. **Abstract** a one paragraph “abstract” type overview of what your project consists of. This should be written for a general audience, something that anyone who has taken up to 211 could understand. It should follow guidance of a scientific abstract.
2. **Reflection** a one paragraph reflection that summarizes challenges faced and what you learned doing your project
3. **Artifacts** links to other materials required for assessing the project. This can be a public facing web resource, a private repository, or a shared file on URI google Drive.

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

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`.

For working alone there are two options, for working with a partner there is one.

212 Project Solo- Docs focus

Use this option if your team for your 212 project is not currently enrolled in this class or does not want to do a collaborative build. This version focuses on the user docs.

```
## 212 Project Doc & Developer onboarding

Add documentation website and developer onboarding information to your CSC 212 project.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will solv
```

```
 <!-- describe what you will do , will it be research, write & present? will there be something you build? wi
1. ensure there is API level documentation in the code files
1. build a documentation website using [jupyterbook/ sphinx/doxygen/] that includes setup instructions and ex
1. configure the repo to automatically build the documentation website each time the main branch is updated


### Deliverables


- link to repo with the contents listed in method in the reflection file


### Milestones

<!-- give a target timeline -->
```

212 Project Solo- Developer focus

Use this option if your team for your 212 project is not currently enrolled in this class or does not want to do a collaborative build. This version focuses on the contributor experience.

```
## 212 Project Doc & Developer onboarding

Add documentation website and developer onboarding information to your CSC 212 project.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will solv

This project will provide information for a user to use the data structure implemented for a CSC 212 project

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build? wi

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 some
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 -->
```

212 Project Pair

Use this option if your teammate for your 212 project is in this class and wants to do a collaborative build.

```
## 212 Project Doc & Developer onboarding
```

```
### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will solv

This project will provide information for a user to use the data structure implemented for a CSC 212 project

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build? wi
1. ensure there is API level documentation in the code files
1. build a documentation website using [jupyterbook/ sphinx/doxygen/] that includes setup instructions and ex
1. configure the repo to automatically build the documentation website each time the main branch is updated
1. add [code tours](https://marketplace.visualstudio.com/items?itemName=vsls-contrib.codetour) that help some
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 -->
```

Syllabus and Grading FAQ

How much does activity x weigh in my grade?

There is no specific weight for any activities, because your grade is based on earning the badges. Everything at a level must be complete and correct.

How do I keep track of my earned badges?

You will have several options. You will have a project board that you can track assigned work, in progress work and earned badges with in one place. This is quite different than checking your grade in BrightSpace, but using tools like this represents the real tools used by developers.

You will be able to use provided command line tools and github actions to produce a report of your status at any time from your PR list, starting in the third week. Additionally, at particular points in the course, an in class or class preparation activity will be for you to review a “progress report” that we help you create and update your success plan for the course.

Also, when are each badge due, time wise?

Review and practice must start within a week, but I recommend starting before the next class. Must be a good faith completion within 2 weeks, but again recommend finishing sooner.

Experience reports for missing class is on a case by case basis depending on why you missed class. You must have a plan by the next class.

Who should I request to review my work?

- Experience badge (inclass): TA in your group
- Experience report(makeup) `@brownsarahm``
- Review badge: `@instructors` team (it will convert to one of the three of us)
- Explore Proposal: `@brownsarahm`
- Explore Badge: `@brownsarahm`
- Build Proposal: `@brownsarahm`
- Build Badge: `@brownsarahm`

Will everything done in the penalty free zone be approved even if there are mistakes?

No. In the penalty-free zone I still want you to learn things, but we will do extra work to make sure that you get credit for all of your effort even if you make mistakes in how to use GitHub. We will ask you to fix things that we have taught you to fix, but not things that we will not cover until later.

The goal is to make things more fair while you get used to GitHub. It's a nontrivial thing to learn, but getting used to it is worth it.

I want this class to be a safe place for you to try things, make mistakes and learn from them without penalty. A job is a much higher stakes place to learn a tool as hard as GitHub, so I want this to be lower stakes, even though I cannot promise it will be easy.

Once we make revisions on a pull request, how do we notify you that we have done them?

You do not have to do anything, GitHub will automatically notify which ever one of us who reviewed it initially when you make changes.

What should work for an explore badge look like and where do I put it?

It should be a tutorial or blog style piece of writing, likely with code excerpts or screenshots embedded in it.

an example that uses mostly screenshots

an example of heavily annotated code

They should be markdown files in your KWL repo. I recommend myst markdown.

Git and GitHub

I can't push to my repository, I get an error that updates were rejected

If your error looks like this…

```
 ! [rejected] main -> main (fetch first)
error: failed to push some refs to <repository name>
hint: Updates were rejected because the remote contains work that you do
hint: not have locally. This is usually caused by another repository pushing
hint: to the same ref. You may want to first integrate the remote changes
hint: (e.g., 'git pull ...') before pushing again.
hint: See the 'Note about fast-forwards' in 'git push --help' for details.
```

Your local version and github version are out of sync, you need to pull the changes from github to your local computer before you can push new changes there.

After you run

```
git pull
```

You'll probably have to resolve a merge conflict

My command line says I cannot use a password

GitHub has strong rules about authentication You need to use SSH with a public/private key; HTTPS with a Personal Access Token or use the GitHub CLI auth

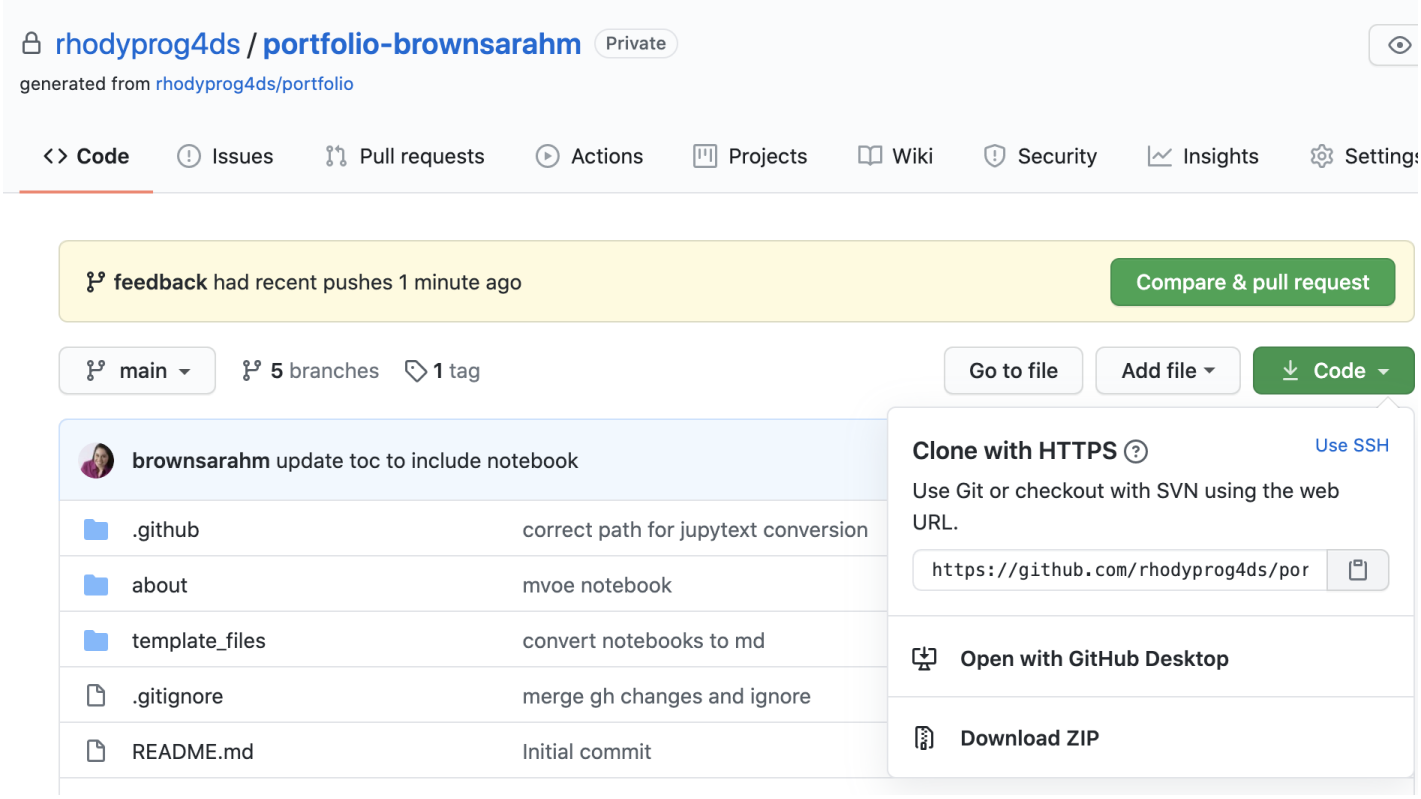
Help! I accidentally merged the Badge Pull Request before my assignment was graded

That's ok. You can fix it.

note: these instructions use the main branch the way we use the badge branches and the feedback branch the way we use the main branch in this course

You'll have to work offline and use GitHub in your browser together for this fix. The following instuctions will work in terminal on Mac or Linux or in GitBash for Windows. (see Programming Environment section on the tools page).

First get the url to clone your repository (unless you already have it cloned then skip ahead): on the main page for your repository, click the green "Code" button, then copy the url that's show



Next open a terminal or GitBash and type the following.

```
git clone
```

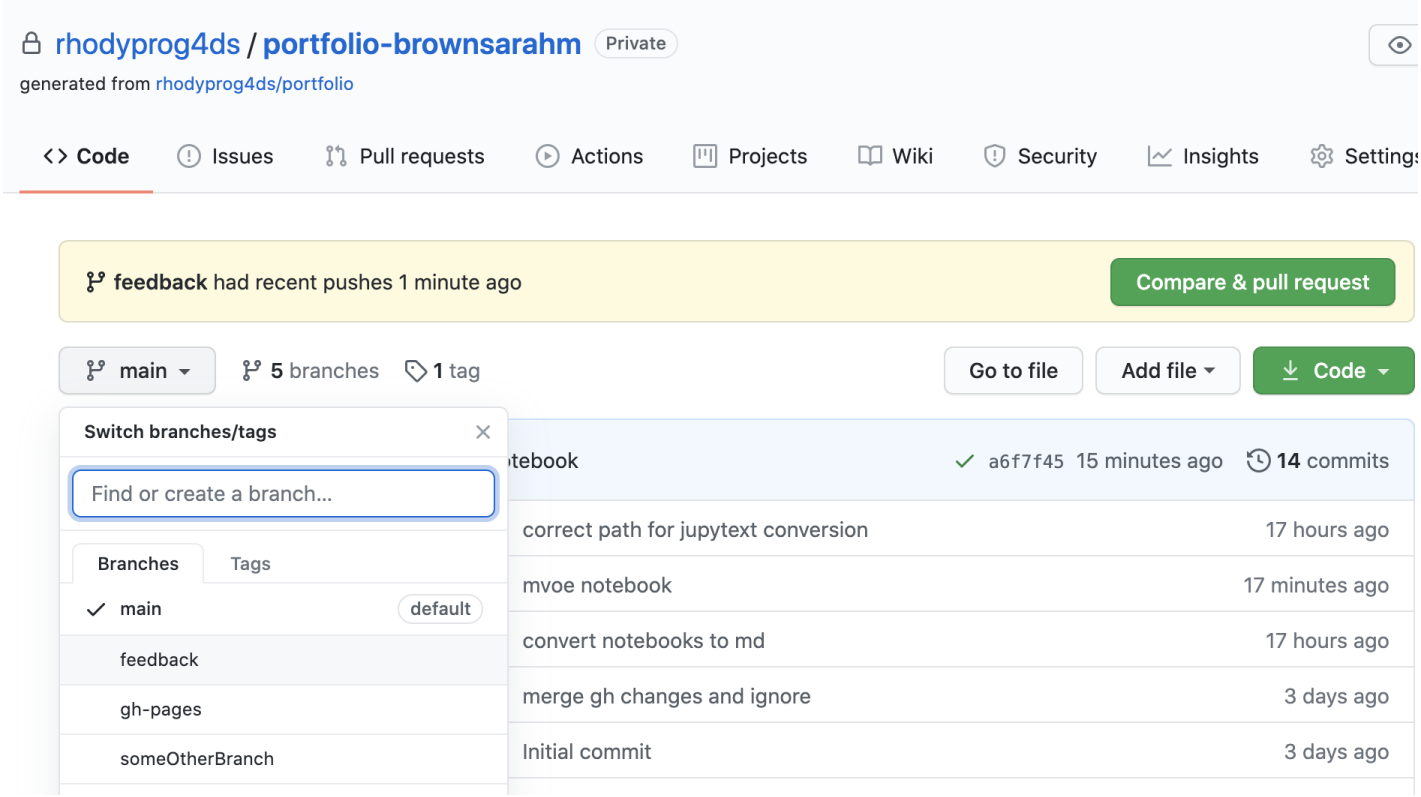
then past your url that you copied. It will look something like this, but the last part will be the current assignment repo and your username.

```
git clone https://github.com/rhodyprog4ds/portfolio-brownsarahm.git
```

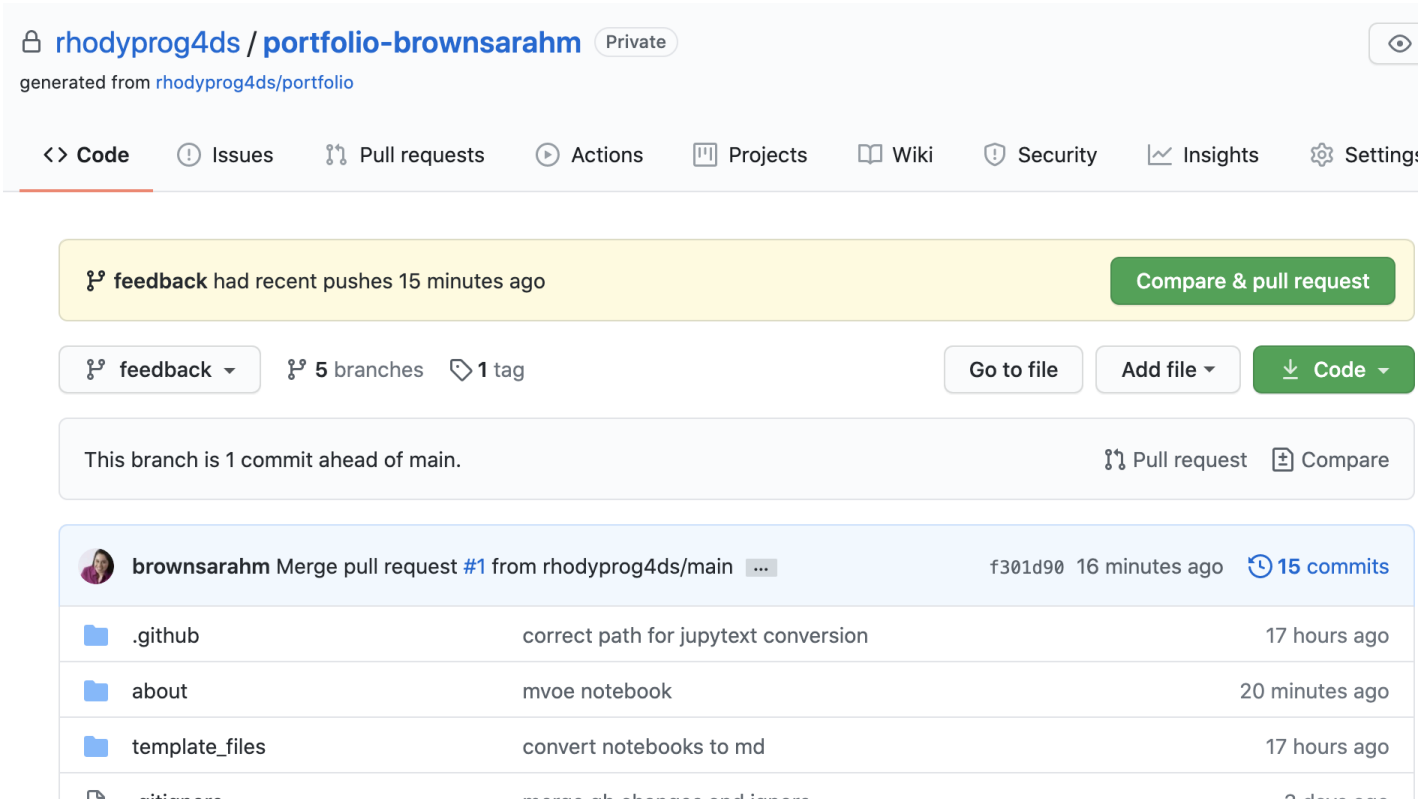
When you merged the Feedback pull request you advanced the `feedback` branch, so we need to hard reset it back to before you did any work. To do this, first check it out, by navigating into the folder for your repository (created when you cloned above) and then checking it out, and making sure it's up to date with the `remote` (the copy on GitHub)

```
cd portfolio-brownsarahm
git checkout feedback
git pull
```

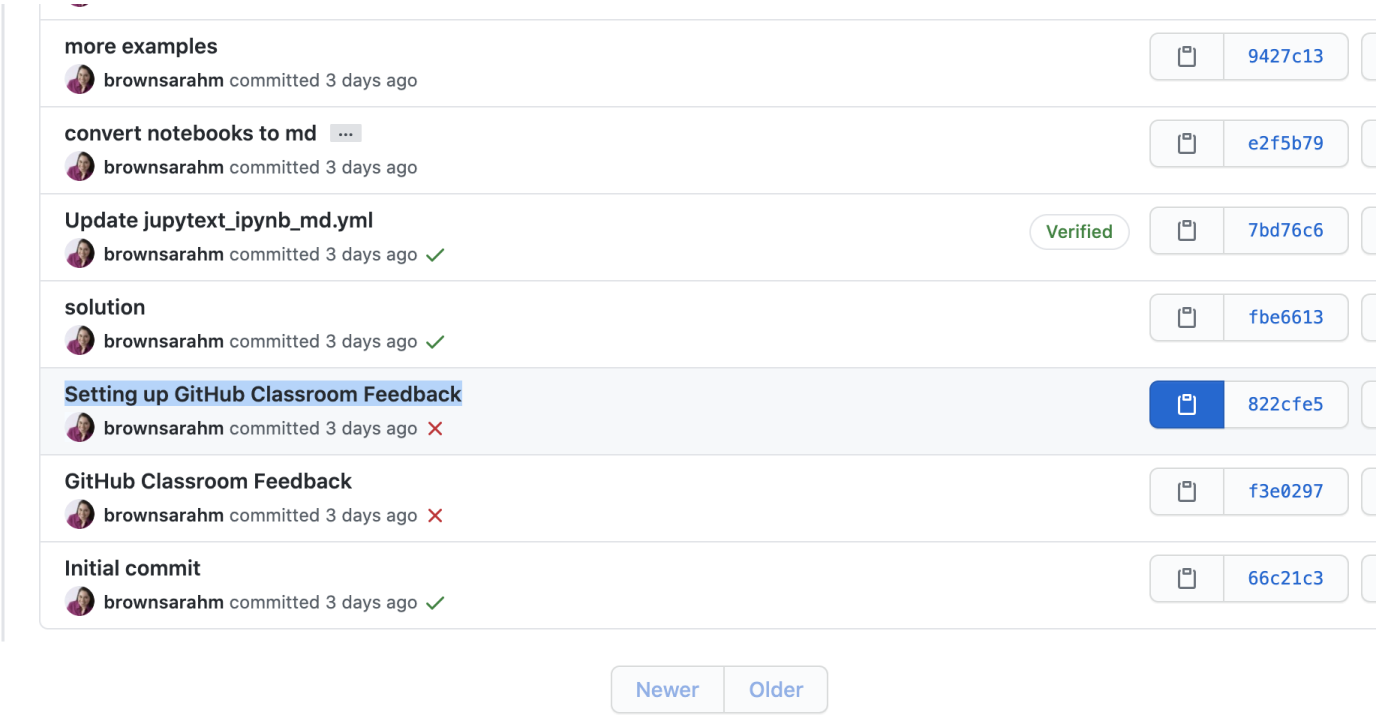
Now, you have to figure out what commit to revert to, so go back to GitHub in your browser, and swithc to the feedback branch there. Click on where it says `main` on the top right next to the branch icon and choose feedback from the list.



Now view the list of all of the commits to this branch, by clicking on the clock icon with a number of commits



On the commits page scroll down and find the commit titled “Setting up GitHub Classroom Feedback” and copy its hash, by clicking on the clipboard icon next to the short version.



Now, back on your terminal, type the following

```
git reset --hard
```

then paste the commit hash you copied, it will look something like the following, but your hash will be different.

```
git reset --hard 822cfe51a70d356d448bcaede5b15282838a5028
```

If it works, your terminal will say something like

```
HEAD is now at 822cfe5 Setting up GitHub Classroom Feedback
```

but the number on yours will be different.

Now your local copy of the `feedback` branch is reverted back as if you had not merged the pull request and what's left to do is to push those changes to GitHub. By default, GitHub won't let you push changes unless you have all of the changes that have been made on their side, so we have to tell Git to force GitHub to do this.

Since we're about to do something with forcing, we should first check that we're doing the right thing.

```
git status
```

and it should show something like

```
On branch feedback
```

Your number of commits will probably be different but the important things to see here is that it says `On branch feedback` so that you know you're not deleting the `main` copy of your work and `Your branch is behind origin/feedback` to know that reverting worked.

Now to make GitHub match your reverted local copy.

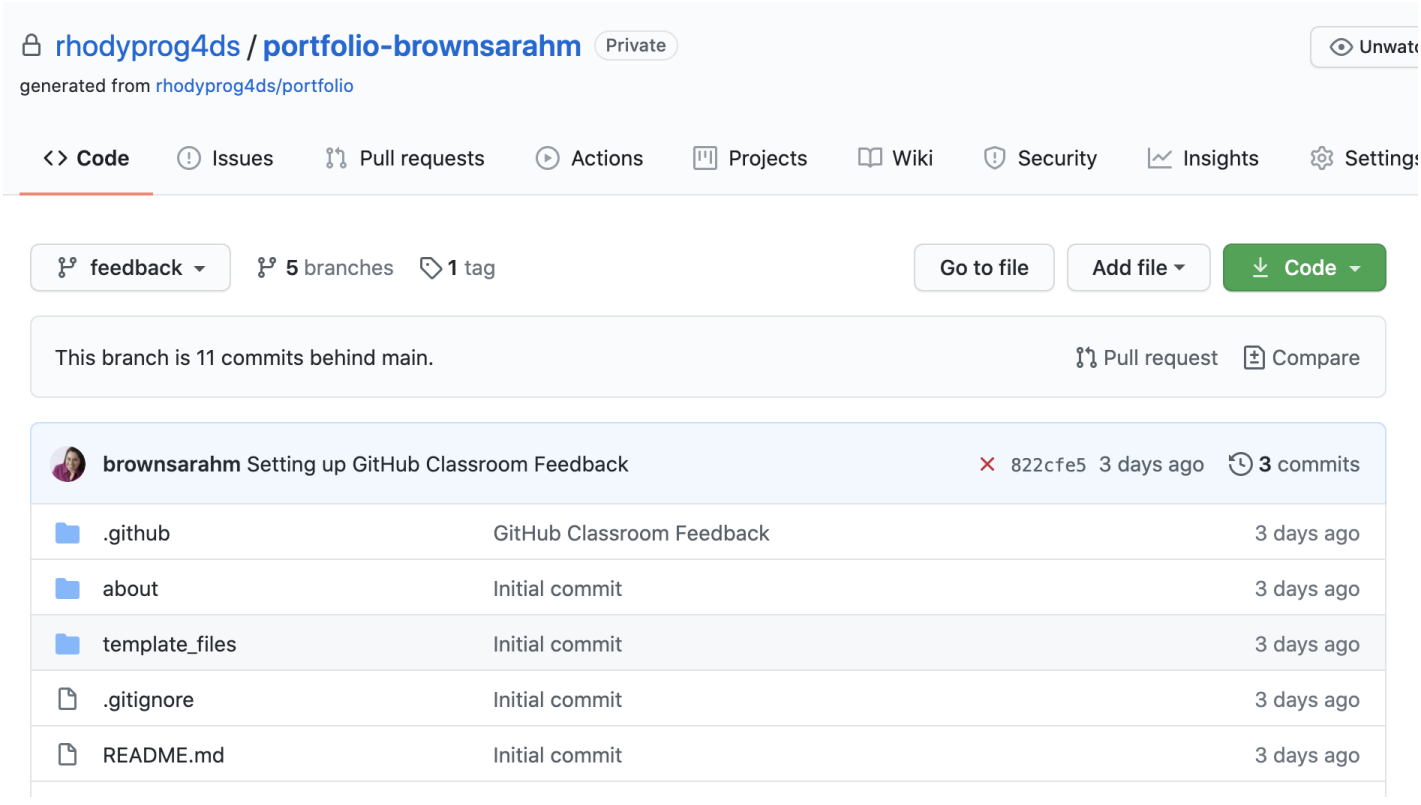
```
git push origin -f
```

and you'll get something like this to know that it worked

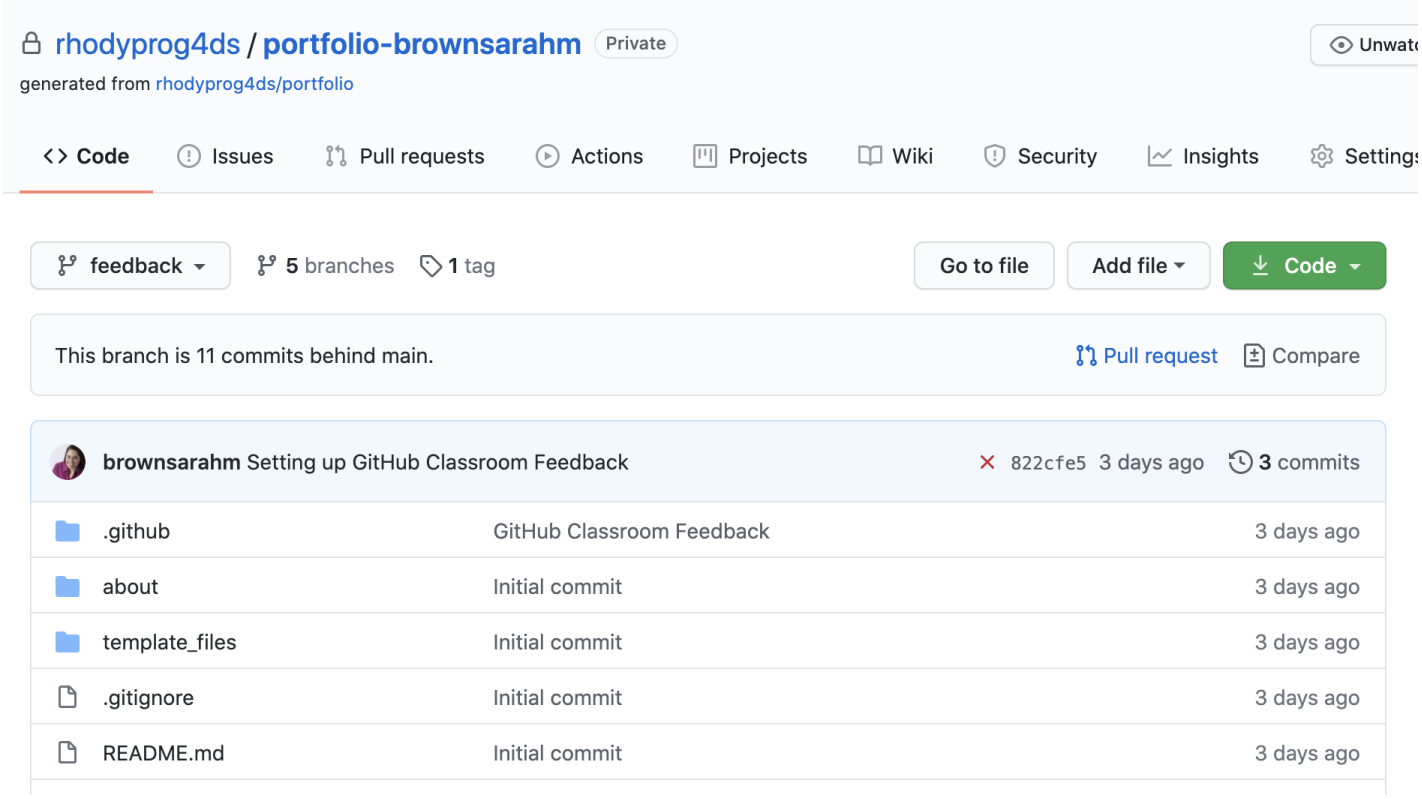
```
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/rhodyprog4ds/portfolio-brownsarahm.git
 + f301d90...822cfe5 feedback -> feedback (forced update)
```

Again, the numbers will be different and it will be your url, not mine.

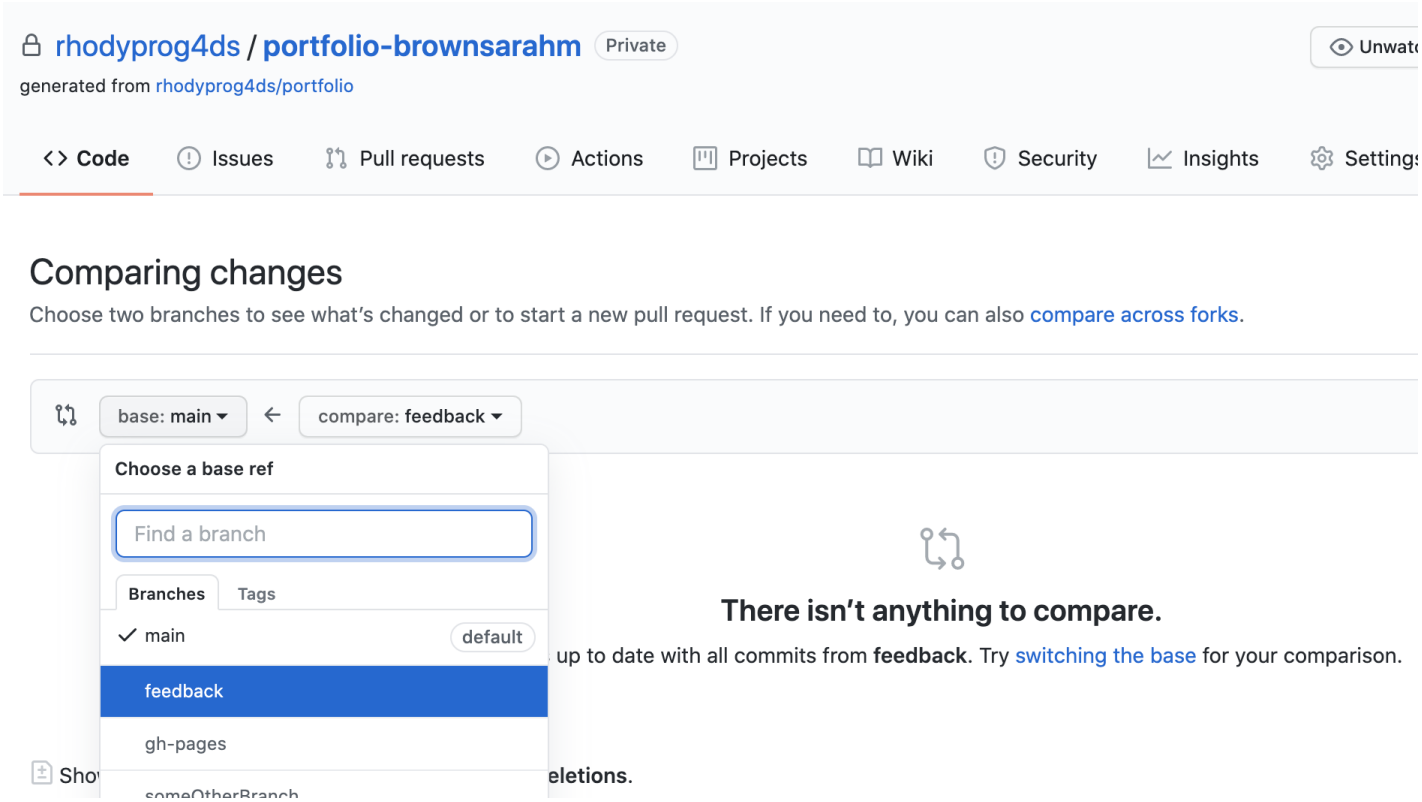
Now back on GitHub, in your browser, click on the code tab. It should look something like this now. Notice that it says, "This branch is 11 commits behind main" your number will be different but it should be 1 less than the number you had when you checked `git status`. This is because we reverted the changes you made to main (11 for me) and the 1 commit for merging main into feedback. Also the last commit (at the top, should say "Setting up GitHub Classroom Feedback").



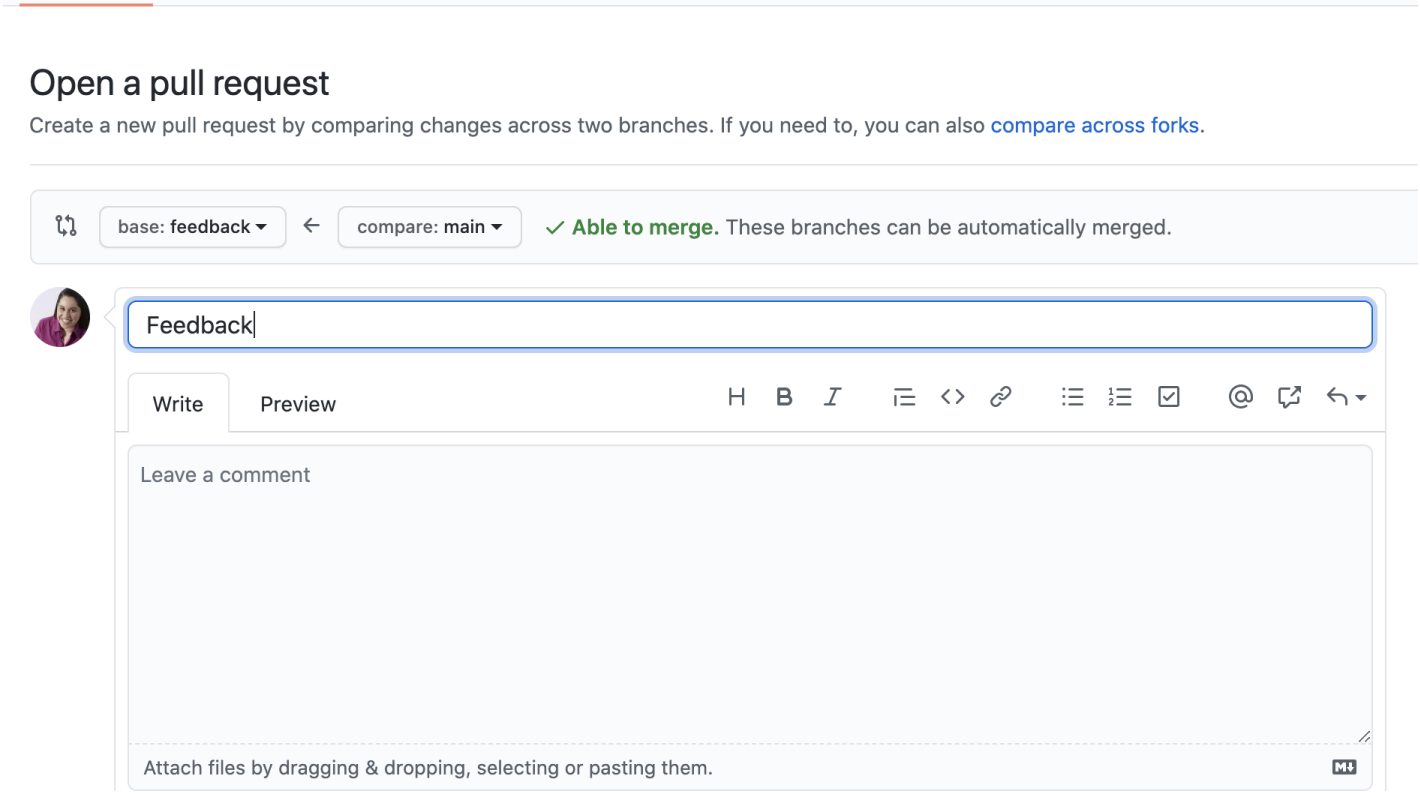
Now, you need to recreate your Pull Request, click where it says pull request.



It will say there isn't anything to compare, but this is because it's trying to use `feedback` to update `main`. We want to use `main` to update `feedback` for this PR. So we have to swap them. Change base from `main` to `feedback` by clicking on it and choosing `feedback` from the list.



Then the change the compare `feedback` on the right to `main`. Once you do that the page will change to the "Open a Pull



Make the title “Feedback” put a note in the body and then click the green “Create Pull Request” button.

Now you’re done!

If you have trouble, create an issue and tag `@@rhodyprog4ds/fall20instructors` for help.

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 the
```

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

Glossary

Skip to main content

> **Tip**
>
> We will build a glossary as the semester goes on. When you encounter a term you do not know, create an issue to ask for help, or contribute a PR after you find the answer.

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

bitwise operator

an operation that happens on a bit string (sequence of 1s and 0s). They are typically faster than operations on whole integers.

bitwise operator

an operation that happens on a bit string (sequence of 1s and 0s). They are typically faster than operations on whole integers.

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.

directory

a collection of files typically created for organizational purposes

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.

fixed point number

the concept that the decimal point does not move in the number (the example in the notes where if we split up a bit in the middle and one half was for the decimal and the other half was for the whole number. Cannot represent as many numbers as a floating point number.

.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

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.

git objects

something (a file, directory) that is used in git; has a hash associated with it

GitHub

a hosting service for git repositories

Skip to main content

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

the branch that is currently being 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

hash function

the actual function that does the hashing of the input (a key, an object, etc.)

hashing

putting an input through a function and getting a different output for every input (the output is called a hash; used in hash tables and when git hashes commits).

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.

integreated 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.

Linker

a program that links together the object files and libraries to output an executable file.

path

the “location” of a file or folder(directory) in a computer

pull (changes from a repository)

download changes from a remote repository and update the local repository with these changes.

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

repository

a project folder with tracking information in it in the form of a .git file

ROM (Read-Only Memory)

Memory that only gets read by the CPU and is used for instructions

Skip to main content

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)

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.

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

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 boo
programmi
Brain As of
by clicking
contents s

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

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

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

the third column is when is how long ago/when that commit was made

README file

be eligible for a build badge.

The bottom of the right panel has information about the languages in the project

Language/Shell Specific References

- bash
- C
- Python

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

Skip to main content

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.

> **Not**
>
> A fun
> debu

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:

```
CSC310
  |- notes
  |- portfolio-username
  |- 02-accessing-data-username
  |- ...
```

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 `rhodyprog4ds` organization as the owner, not your personal acount. Since your account is not the owner, they do not show on your profile.

Your assignment repositories are all private during the semester. At the end, you may take ownership of your portfolio[^pttrans] if you would like.

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.

> **Warning**
>
> Don't try to work on a repository that does not end in your username; those are the template repositories for the course and you don't have edit permission on them.

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