About this Site

Contents

Skip to main content

- Explore Badges
- Build Badges

FAQ

- Syllabus and Grading FAQ
- Git and GitHub

Resources

- Glossary
- General Tips and Resources
- How to Study in this class
- GitHub Interface reference
- 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 Spring 2023 with Professor Brown.

This class meets TTh 12:30-1:45 in Morrill Hall 323.

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

course calendar

> **Tip**
>
> subscribe to that calendar in your favorite calendar application

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

All class notes can be downloaded in multiple formats, including as a notebook. Some pages of the syllabus and resources are

Try it Yourself

Notes will have exercises marked like this

Question from Class

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

Further reading

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

Hint

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

Click here!

Special tips will be formatted like this

Check your Comprehension

Questions to use to check your comprehension will looklike this

Contribute

Chances to earn community badges will sometimes be marked like this

Computer Systems and Programming Tools

About this course

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

About this syllabus

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

Tools and Resources

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

All of the tools and resources below are either:

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

BrightSpace

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

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

Not

Seein
loggi
being

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

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

Course Website

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

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

GitHub

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

You will also need the gh CLI. It will help with authentication and allow you to work with other parts of github besides the core git operations.

> **Important**
>
> You need to install this on Mac

Programming Environment

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

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

Requirements:

- Python with scientific computing packages (numpy, scipy, jupyter, pandas, seaborn, sklearn)
- a C compiler
- Git
- A bash shell

Skip to main content

- 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 used to the tools there will be a grade free zone for the first few weeks.

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, tools, and components of systems.
2. 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
3. Identify the computational pipeline from hardware to high level programming language

Skip to main content

5. Describe the context under which essential components of computing systems were developed and explain the impact of that context on the systems.

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, 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, it will still be graded and logged. After that, we will not see it.

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

To earn a D you must complete:

- 24 experience badges

To earn a C you must complete:

- 24 experience badges
- 18 review badges

To earn a B you must complete:

- 24 experience badges
- your choice:
  - 18 practice badges
  - 12 review + 12 practice

For an A you must complete:

Skip to main content

- your choice:
  - 18 practice badges + 9 explore badges
  - 18 review badges + 3 build badges
  - 6 review badges + 12 practice badges + 6 explore badges + 1 build badges
  - 12 review badges + 6 practice badges+ 3 explore badges + 2 build badges

You can also mix and match to get +/-. For example (all examples assume 24 experience badges)

- A-: 18 practice + 6 explore
- A-: 6 review + 12 practice + 9 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.

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.

At the end 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

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

Skip to main content

You can make up an experience badge by:

- preparing for class
- reading the posted notes
- completing the activity from the notes
- producing an “experience report” OR attending office hours

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 adcised 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.

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

Skip to main content

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

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

Explore Badges

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

Details and more ideas are on the explore page.

You can earn an explore badge by:

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

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

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

You should create one PR per badge

Build Badges

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

You can earn a build badge by:

Skip to main content

- 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
- 5 community = 1 review
- 7 community = 1 practice.
- 5 community badges + 1 review = 1 practice.
- 10 community = + step 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 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

> **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)
- 24 experience + 18 practice + 10 community = B+ (modifier)
- 23 experience + 18 practice + 13 community = B+ (modifier, replace 1 experience)
- 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 🎁 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

> **Not**
>
> We d
> assig
> The c
> be th

Skip to main content

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.

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 reviewed and 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 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

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 stages:

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

Explore badges are feedback-limited. You will not get feedback on subsequent explore badge proposals until you earn the first one. Once you have one earned, then you can have up to two in progress and two in revision at any given time.

Build Badges

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

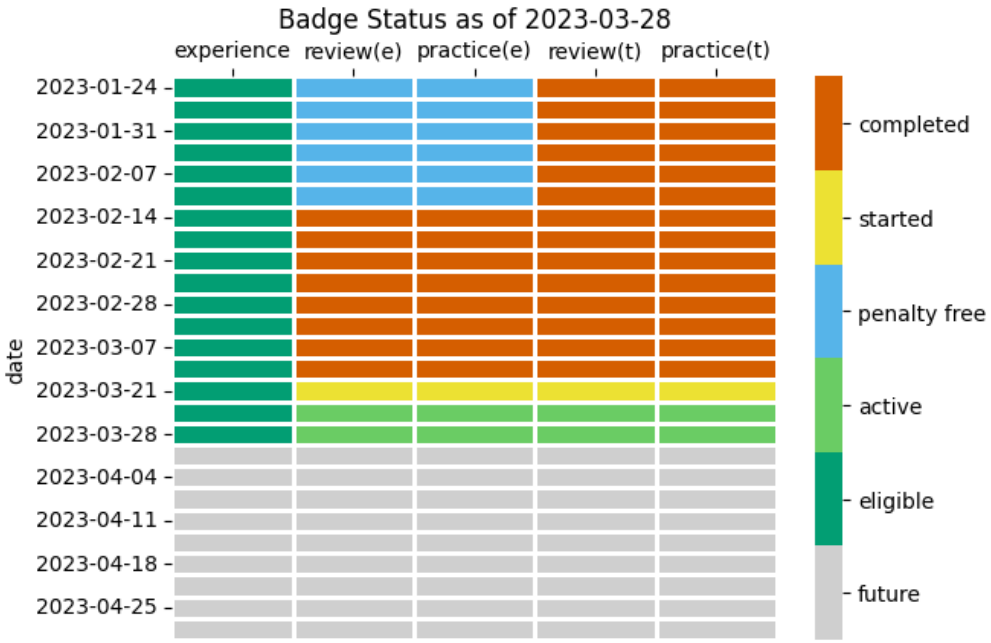
(experimental) Badge Visualizations

Badge status

> **Warning**
>
> this is not guaranteed to stay up to date with current, but the listed date is accurate. You can see the code that generates the figure behind the “click to show” button. It is Python code that you should be able to run anywhere with the required libraries installed.

▶ Show code cell source

```
Text(0.5, 1.0, 'Badge Status as of 2023-03-28')
```



The (e) columns are what will be enforced, the (t) columns is ideal, see the deadlines section of the syllabus for more on the statuses for more detail.

The following table shows, as of '2023-03-28', the number of badges with each status.

▶ Show code cell source

	experience	review	practice	review_target	practice_target
eligible	16.0				
penalty free		6.0	6.0		
active		1.0	1.0	1.0	1.0
started		1.0	1.0	1.0	1.0
completed		8.0	8.0	14.0	14.0

This means that, as of '2023-03-28' you are fully caught up if you have:

- 16 experience badges submitted.
- 14 total review and practice badges completed.
- 1 additional review and/or practice badges started and in progress.

Notes:

- If you do not have at least 13 experience badges earned, you should visit office hours to get caught up.
- There are are 6 review and practice badges that are from the penalty free zone, so they never expire.
- Most of your experience and completed badges should be earned, but there is no deadline to fix issues with them.

Skip to main content

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 Feb 21 participant E as experience Feb 23 participant M as main note over P: complete prepare work
 between feb 21 and 23 note over E: run experience badge workflow
 at the end of class feb 23 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-2023-02-21 checkout prepare-2023-02-21 commit id: "gitunderstanding.md" branch experience-2023-02-23 checkout experience-2023-02-23 commit id: "initexp" merge prepare-2023-02-21 commit id: "fillinexp" commit id: "revisions" tag:"approved" checkout main merge experience-2023-02-23
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 start[do one task] commit[commit work to the branch] moretasks[complete the other tasks] ccommit[commit them to the branch] reqreview[request a review] started[[Started
 due within one week
 of posted date]] completed[[Completed
due 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 --

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

Schedule

Overview

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

How does this class work?

one week

We'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.

- 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

Content from above will be expanded and slotted into specific classes as we go. This will always be a place you can get reminders of what you need to do next and/or what you missed if you miss a class as an overview. More Details will be in other parts of the site, linked to here.

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

be found on the AEC tutoring page.

- **Academic Skills Development** resources helps students plan work, manage time, and study more effectively. In Fall FIXME, all Academic Skills and Strategies programming are offered both online and in-person. UCS160: Success in Higher Education is a one-credit course on developing a more effective approach to studying. Academic Consultations are 30-minute, 1 to 1 appointments that students can schedule on Starfish with Dr. David Hayes to address individual academic issues. Study Your Way to Success is a self-guided web portal connecting students to tips and strategies on studying and time management related topics. For more information on these programs, visit the Academic Skills Page or contact Dr. Hayes directly at davidhayes@uri.edu.
- The **Undergraduate Writing Center** provides free writing support to students in any class, at any stage of the writing process: from understanding an assignment and brainstorming ideas, to developing, organizing, and revising a draft. Fall 2020 services are offered through two online options: 1) real-time synchronous appointments with a peer consultant (25- and 50-minute slots, available Sunday - Friday), and 2) written asynchronous consultations with a 24-hour turn-around response time (available Monday - Friday). Synchronous appointments are video-based, with audio, chat, document-sharing, and live captioning capabilities, to meet a range of accessibility needs. View the synchronous and asynchronous schedules and book online, visit uri.mywconline.com.

General 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 Services for Students Statement:

Your access in this course is important. Please send me your Disability Services for Students (DSS) 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 DSS, please contact them to engage in a confidential conversation about the process for requesting reasonable accommodations in the classroom. DSS can be reached by calling: 401-874-2098, visiting: web.uri.edu/disability, or emailing: dss@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

Skip to main content

- 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

URI COVID-19 Statement

The University is committed to delivering its educational mission while protecting the health and safety of our community. While the university has worked to create a healthy learning environment for all, it is up to all of us to ensure our campus stays that way.

As members of the URI community, students are required to comply with standards of conduct and take precautions to keep themselves and others safe. Visit web.uri.edu/coronavirus/ for the latest information about the URI COVID-19 response.

- Universal indoor masking is required by all community members, on all campuses, regardless of vaccination status. If the universal mask mandate is discontinued during the semester, students who have an approved exemption and are not fully vaccinated will need to continue to wear a mask indoors and maintain physical distance.
- Students who are experiencing symptoms of illness should not come to class. Please stay in your home/room and notify URI Health Services via phone at 401-874-2246.
- If you are already on campus and start to feel ill, go home/back to your room and self-isolate. Notify URI Health Services via phone immediately at 401-874-2246.

If you are unable to attend class, please notify me at brownsarahm@uri.edu. We will work together to ensure that you are able to successfully complete the course.

Office Hours & Communication

Announcements

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

Warning

For the first 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
```

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

Skip to main content

Help Hours

```
/tmp/ipykernel_1855/2146052215.py:1: FutureWarning: this method is deprecated in favour of `Styler.hide(axis=
  help_df.style.hide_index()
```

day	time	location	host
TBA	TBA	TBA	Dr. Brown
Monday/Wednesday	10:00-12:00	Zoom	Mark
Tuesday/Thursday	5:00-7:00	Zoom	Marcin

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

- use e-mail for general inquiries or notifications
- 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

Should you e-mail your work?

No, request a pull request review or make an issue if you are stuck

1. Welcome and Introduction

1.1. Introductions

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!

Are emoji fun or do they make me Old? *no penalty, this is for fun & to practice*

- [ ] fun
- [ ] not cool

questions can also 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

What is the topic of this course?

- [ ] hardware
- [ ] programming
- [x] computer systems and programming tools

or open ended

What is one thing you want the TAs and I to know about you?

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?

Skip to main content

This information will help me prepare

1.2.2. 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. Getting started with KWL charts

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.

Today we did the following:

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

1.4. 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.

1.5. 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).

1.5.1. How it fits into your CS degree

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

Skip to main content

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.

Tip

know
wher
under

1.6. Course Admin

1.6.1. Programming is Collaborative

There are two very common types of collaboration

- code review (workign 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.

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.

team formation survey

Important

Remember to fill out the team formation survey

1.6.2. 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

1.6.3. Get Credit for Today's class

1. Run your Experience Reflection (incalss) action on your kwl repo
2. today's evidence is your KWL repo existing and having the commits as above

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart.
3. review git and github vocabulary (include link in your badge PR)

1.8. Prepare for Next Class

1. Find the glossary page for the course website. Preview 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. Use the discussions to ask for help

1.9. More Practice

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart.
3. review git and github vocabulary (include link in your badge PR)
4. Read more about version control in general and add a “version control” row to your KWL chart with all 3 columns filled in.

1.10. Questions After Today’s Class

> **Note**
>
> I will add the rest later, today I had a one time conflict.

1.10.1. How to directly merge all suggestions without clicking commit suggestion?

Unfortunately, that is not an option on a PR review, but in general, you will not make a lot of changes in a review. We will learn other ways to do this

1.10.2. What is the importance of github in the real world? Is it so people can collaborate on code together, or maybe its somewhere to share your code and help others/inspire other code writers or a combination of both?

GitHub facilitates both collaboration and social access of code. It can also host websites (like this one) (and my lab site). It is deisgned to faciliate things that developers need.

1.10.3. I couldn’t find the button to turn a comment into a suggestion. My screen looked almost identical to the instructor’s, but the suggestion button wasn’t there.

You may have clicked the blue + icon on a deleted line instead of an added line.

1.10.4. Easy way to remember everything we learned about pull requests?

1.10.5. How do I add collaborators to my repository?

In the kwl repo you don't have the permissions. But otherwise, on the GitHub Settings tab.

1.10.6. How does the grading work in this class?

You earn badges by completing assigned activites.

1.10.7. Is attendance mandatory

Mostly, yes. You can make up a missed class, but it will always be easier to participate in class.

1.10.8. How do these git commands work in the terminal?

So far we used the terms, but have not seen the command directly yet. We will see them on Tuesday 1/31.

1.10.9. What are checks used for in the pull request tab?

Checks can run tests or other quality checks on the code. For example, they can check that the code follows good style or that contributions do not change the test coverage.

1.10.10. Will we go more in depth in creating forks and features that git has?

Yes we will go in depth on git features. We may not spend a lot of time on forks though (which are a GitHub feature). We will cover git in a way that does not cover every possible command, but focuses on *How* the most important ones work so that you have the foundation to understand the other parts (and new features that are introduced) quickly.

1.10.11. How are these systems used in work outside college?

We will learn more about this later when we discuss the Stack overflow developer survey.

1.10.12. why does the branch have more commits then the main. I think i understand, but it could be clarified

This is a good question. The branch had more commits than main because we told GitHub to make the new commit on a *new* branch. New branches have all of the history of the branch used to create them, plus any commits made to that branch.

2. Course Logistics and Learning

2.1. Syllabus Review

- Read the navigation on the left carefully

Skip to main content

2.1.1. Scavenger Hunt

> **Note**
>
> The goal here is to make sure you know where to find basic things, not that you have memorized every bit of information about the course

Where can you find when office hours are?

Where can you find the detailed list of what to prepare for today's class?

Where is the regrading policy?

Something went wrong in an assignment repo on GitHub, what should you check before asking for help?

2.2. How does this work?

2.2.1. In class:

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

2.2.2. Outside of class:

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

(practice extending will vary depending on what grade you are working toward)

2.2.3. Grade Tracking

We will use a GitHub project to track your grade. Create a project on the course organization that is named `<username> grade tacking` where `<username>` is your GitHub username. We will help you populate it.

> **Important**

In your repository, edit the: `/.github/workflows/getassignment.yml` file to remove either the two lines about practice or the two lines about review since you can only earn one or the other of these two types of badge per date.

> **Warning**
>
> This is different from in class

delete the `/.github/workflows/track.yml` file, we will add items to the project a different way.

2.3. 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.

> **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.

Systems programming is how to look at the file system, the operating system, etc.

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 how and why they are the way they are. We will therefore study in a broad way.

We will look at blogs, surveys of developers, and actually examine the systems themselves.

2.4. Mental Models and Learning

2.4.1. What is it like to know something really well?

> 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.

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

2.5. Why do we need this for computer systems?

2.5.1. Systems are designed by programmers

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

> **Important**
>
> Some of this was not discussed in class

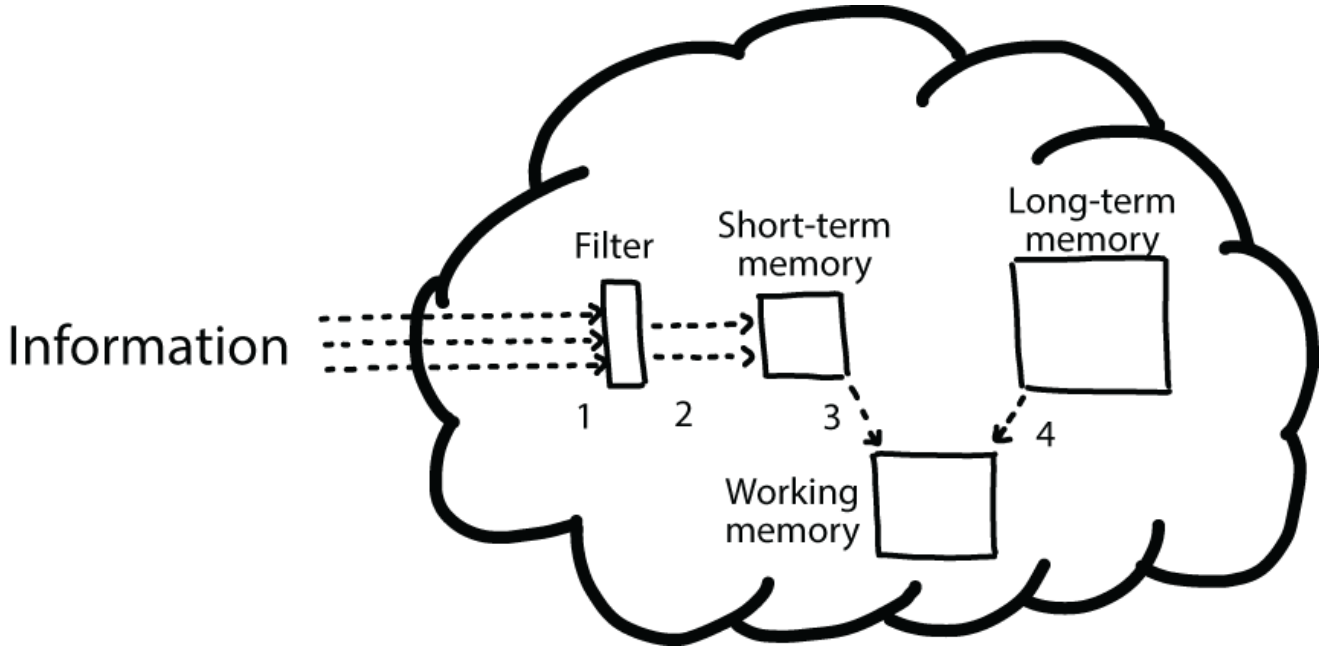


Fig. 2.1 An overview of the three cognitive processes that this book covers: STM, LTM, and working memory. The arrows labeled 1 represent information coming into your brain. The arrows labeled 2 indicate the information that proceeds into your STM. Arrow 3 represents information traveling from the STM into the working memory, where it's combined with information from the LTM (arrow 4). Working memory is where the information is processed while you think about it.

2.5.2. Context Matters

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

2.5.3. Optimal is relative

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

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

2.6. Admin

> **Warning**
>
> I created your issues for you, but GitHub’s server is in a different time zone so the issue titles have the wrong date. You can change it, or not, your choice

Remember you can create branches to work on each badge from the issue

2.7. 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
3. fill in the first two columns of your KWL chart
4. complete the syllabus quiz. If you get less than 100%, submit an FAQ for the course website in your KWL repo in a file named syllabus-faq.md about something that confused you with your best guess at the correct answer. If you get 100%, make a note in your badge PR.

2.8. Prepare for Next Class

1. Find the glossary page for the course website. Preview 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. Use the discussions to ask for help

2.9. More Practice

1. review notes after they are posted, both rendered and the raw markdown include links to each in your badge PR
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`. Use mermaid syntax, to draw your map. GitHub can render it for you including while you work using the preview button.
4. complete the syllabus quiz. If you get less than 100%, submit an FAQ for the course website in your KWL repo in a file named syllabus-faq.md about something that confused you with your best guess at the correct answer. If you get 100%, make a note

2.10. Experience Report Evidence

If you missed class today, there is no separate evidence beyond updates to your repo.

2.11. Questions After Today's Class

> **Important**
>
> Comming soon!

This list is content related and timely questions only, many of the syllabus questions were added to the syllabus faq page so they're in a more logical place to find them when you might need them later.

2.11.1. Can you use other files besides .yml files as scripts?

In other places, yes, but no on GitHub Actions.

2.11.2. why did we delete the bottom half of the track.yml file?

I had put two copies of it in there thinking it had to work in two different ways, but we do not need the second one.

2.11.3. Can we go over the badge creation and submission process again

Yes, we will

2.11.4. Which shell is best for windows?

This is a really big question that I cannot answer authoritatively, but I can say that bash is the most common on unix, which a lot of servers use, so we will use that. To use bash on Windows, I recommend Git for Windows (GitBash) or Windows subsystem for linux

2.11.5. Also, if pre-class work needs to be attached to experience badges, is the experience (in class) action supposed to be selected before each class to attach the requested work, or will it all be attached at the end of class?

Attach at the end. We will learn more ways to manage things so this gets easier as we go. You can do the work in advance on its own branch and then make a PR from that prepare branch to the experience branch. This PR you can merge without approval.

2.11.6. Will you notify us when a badge is able to be completed or will we have to know oursleves.

You will get announcement

Skip to main content

2.12. What should I do if I fall behind in class?

Raise your hand, go talk to Marcin (or waive for him to come to you) or send a message on prismia. It can even be as short as “help”. Also, check prismia because mostly, I will be posting the steps there too

2.12.1. Where do I see what I need to submit?

At the bottom of the notes, the pages in the activity section of this website and the issues that will be created on your kwl repo will all have the same information.

2.12.2. When can there be office hours so that I can make sure I am doing things correctly?

TA office hours are now posted! Also, as we go through and give feedback, we will be able to tell you if you did it right or not and help you. That’s a good thing about what we have done so far: All of it is fully visible to us and we can help fix it.

2.12.3. Do students like how this class is graded?

This is the first semester with these badges. The other system students liked some parts and found some parts made it too easy to fall behind and hard to know if work was all done. The badges were designed to make it easier to keep track of your work, while still offering choices.

I use a similar system in CSC/DSP310, students who pay attention to how flexible it is, tend to like it. Students who stay confused and do not ask for help tend to think that my other course is more strict than it actually is and do not like it.

2.12.4. Do we ahve to do all the prepare class takss to get the experience badge or just some of them.

You need to do all of them, but my goal is that prepare for class tasks should take about 30 minutes per class session. Sometimes they may be longer (espeically setup/install ones, for reasons beyond my control). When I can anticipate that one may be a longer one, I will post it early with a marker that it advanced notice, like “start thinking about x” or similar. I will then post it again on the class before like, “Make sure that x is done”.

2.12.5. Is it possible to implement github actions to automate some components of version control from your local machine?

Sort of! we will use the GitHub CLI to authenticate next class and it can also help you write scripts to do git actions and to interact with parts of GitHub. You can also use it within GitHub actions.

2.12.6. Questions we will address later

- Does github has a compiler to run code?
- How do we setup a workflow?

Make an issue or PR

Add your question directly to the course website as an issue or a PR.

To make a PR, use the “suggest edit” button behind the icon at the top. It will have you make a fork which is a copy of the repo that lives on your own account instead of the organization and then you can submit a PR>

Doing this will not always be worth a community badge, but I hit a challenge in the way I had planned to, so I’m giving you an extra opportunity.

---

[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.

3. Bash intro & git offline

3.1. Open a Terminal

Windows | **Windows with WSL** | **MacOS**

Use Git for Windows aka GitBash. This program emulates bash for you.

My terminal reminds me that using bash is not Apple’s preference

```
Last login: Sun Jan 29 11:50:33 on ttys017

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.
```

3.2. Getting Organized.

To view where we are, we **p**rint **w**orking **d**irectory using `pwd`.

```
pwd
```

As output we see:

```
/Users/brownsarahm
```

It prints out the *absolute* path, that begins with a `/` above, we used a relative path, from the home directory.

Skip to main content

which is the directions from one location to another.

```
cd Documents/inclass/
```

Here I can list the files.

```
ls
```

```
fa22            prog4dssp23
```

Next we'll make a folder for this course

```
mkdir systems
```

Note that making the new folder does not move us into it.

```
pwd
```

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

For that,we use `cd` again.

```
cd systems/
```

This is an emptry folder, we can confirm with:

```
ls
```

```

```

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

```
cd ..
```

`..` is a special file that points to a specific relative path.

```

```

```
cds systems/
```

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

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

Skip to main content

```
cd systems/
pwd
```

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

```
cd ..
pwd
```

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

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

```
cd
pwd
```

```
/Users/brownsarahm
```

Then we can go back.

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

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

We can use two levels up at once like this:

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

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

```
cd inclass/systems/
```

3.3. Authenticating with GitHub

We have two choices to Download a repository:

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

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

Skip to main content

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
2. GitBash built in authentication

Windows (GitBash) | **Windows (WSL)** | **MacOS X** | **If nothing else works**

Skip straight to clone, then follow the prompts, choosing to authenticate in Browser.

3.3.2. Cloning a repository

We will create a local copy by cloning

```
git clone https://github.com/introcompsys/github-in-class-brownsarahm-1.git
```

```
Cloning into 'github-in-class-brownsarahm-1'...
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (8/8), done.
remote: Compressing objects: 100% (4/4), done.
remote: Total 8 (delta 0), reused 4 (delta 0), pack-reused 0
Receiving objects: 100% (8/8), done.
```

Confirm it worked with:

```
ls
```

```
github-in-class-brownsarahm-1
```

We see the new folder that matches our repo name

3.4. What is in a repo?

We can enter that folder

```
cd github-in-class-brownsarahm-1/
```

When we compare the local directory to GitHub

```
ls
```

```
README.md
```

Skip to main content

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. Review today’s class

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.
4. Reorganize 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 (will be in notes) in a new file, `terminal.md` in your kwl repo. 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, make notes.

```
### Terminal File moving reflection
1. Did this get easier toward the end?
1. What if anything did you get stuck on
1. When do you think that using the terminal will be better than using your GUI file explorer?
```

3.6. Prepare for Next Class

1. Make a list of questions you have about using the terminal
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, come to class on Thursday and read this after.

3.7. More Practice

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.
4. Reorganize 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 (will be in notes) in a new file

`terminal.md` in your kwl repo. Start with a file explorer open, but then try to close it and use only command line tools to explore and make your choices. If you get stuck, look up additional commands to do acomplish your goals.

```
### Terminal File moving reflection
1. Did this get easier toward the end?
1. Use the `history` to see which commands you used and how many times each, make a table below.
1. Did you have to look up how to do anything we had not done in class?
1. When do you think that using the terminal will be better than using your GUI file explorer?
1. What questions/challenges/ relfections do you have after this?
1. What kinds of things might you want to write a bash script for given what you know in bash so far? come up
```

3.8. Experience Report Evidence

To show you completed this activity, show the output of the following commands in `evidence-2023-01-31.md` in the `experiences folder:

```
cd path/you/choce/systems
```

where you replace the `path/you/chose` accordingly

```
ls
cd github-inclass-<yourghusername>
```

with your gh username

```
ls -a
```

3.9. Questions After Today's Class

3.9.1. Questions we will answer next class

- How do changes that I do to the repo offline update on the github website?
- Does github connect to a file automatically or do i have to connect it manual
- What other command options/flag examples like ls -a?
- When we make changes to our cloned repository, do we need to make commits/pull requests within gitbash, or the equivalent, in order for the changes to showup on the browser github?

3.9.2. Questions I need clarification from you to be able to answer

probably go to office hours

- I would like more clarification on how to navigate the experience reflection badge.
- Why is nothing working lol. I tried downloading the github thing from the command line and nothing worked
- After downloading homebrew, the "gh auth login" still doesn't work.

Skip to main content

3.9.3. Why is SSH less secure than the HTTPS standard?

SSH keys are secure, but other shell things are not. Using SSH keys is acutally preferred, but you cannot do other types of command line password operations with GitHub. git alone, does support HTTPS connection with passwords, but GitHub no longer does. It only allows you to use PAT (a temporary password), SSH keys, or using the GitHub CLI tool to use your browser. We will use SSH keys later.

3.9.4. Why would we want to create files from Git Bash terminal as opposed to File Explorer or CMD Prompt

bash is not operating specific, so using bash commands can work on more systems including most servers, while CMD commands can only work on Windows (and basically no servers). File explorer is fine one file at a time, but you cannot automate it.

3.9.5. How do you move folders into other folders in terminal once you've created them?

The `mv` command does this. see its documentation

3.9.6. Is there an easy way to remember shortcuts for terminal?

Lots of practice.

3.9.7. is GitHub desktop used less often than git via command line?

I would say yes for a few reasons:

- I think only novices and nonprogrammers use GitHub desktop extensively.
- It has fewer features.
- It's not typically avialable on a server.
- It only works with GitHub, not other servers

The GitHub specific CLI, I do not have a good sense of how often it is used.

4. How can I work with branches offline?

4.1. Review

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

```
pwd
```

```
/Users/brownsarahm
```

Skip to main content

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

```
cd Do
```

```
Documents/ Downloads/
```

Let's go back to the github inclass repo.

```
cd Documents/inclass/systems/github-in-class-brownsarahm-1/
```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-in-class-brownsarahm-1
```

Also recall our one file

```
ls
```

```
README.md
```

and how to see hidden files.

```
ls -a
```

```
.               ..              .git            .github         README.md
```

4.2. How do I know what git knows?

`git status` is your friend.

```
git status
```

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

nothing to commit, working tree clean
```

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

Skip to main content

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.

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-in-class-brownsarahm-1
 * [new branch]      2-create-an-about-file -> origin/2-create-an-about-file
```

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

```
git status
```

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

nothing to commit, working tree clean
```

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

Next, we switch to that branch.

```
git checkout 2-create-an-about-file
```

```
branch '2-create-an-about-file' set up to track 'origin/2-create-an-about-file'.
Switched to a new branch '2-create-an-about-file'
```

and verify what happened

```
git status
```

```
On branch 2-create-an-about-file
Your branch is up to date with 'origin/2-create-an-about-file'.

nothing to commit, working tree clean
```

4.4. Creating a file on the terminal

The `touch` command creates an empty file.

```
touch about.md
```

Skip to main content

```
:class: cell_input
ls
```

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

and check how git has changed too?

```
:class: cell_input
git status
```

```
:class: cell_output
On branch 2-create-an-about-file
Your branch is up to date with 'origin/2-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)
```

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.

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.

```
git status
```

```
On branch 2-create-an-about-file
Your branch is up to date with 'origin/2-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`.

```
git commit -m "create empty about"
```

```
[2-create-an-about-file 57de0cd] create empty about
 1 file changed, 0 insertions(+), 0 deletions(-)
 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

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 2-create-an-about-file
Your branch is ahead of 'origin/2-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.

```
ls
```

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

We can send them to github with `git push`

```
git push
```

```
Enumerating objects: 4, done
```

Skip to main content

```
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 313 bytes | 313.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-in-class-brownsarahm-1.git
   3f54148..57de0cd  2-create-an-about-file -> 2-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

Remember our directory has other things in it:

```
ls -a
```

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

4.5. What does push do?

`git push` only sends the `.git` directory. To see that, we can make a file

```
touch temp
```

but not add or commit it

```
git push
```

```
Everything up-to-date
```

when we push git tells us there is nothing to send, because git doesn't know about that file.

Even though our computer can see that file with `ls`

```
ls
```

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

git only looks at the working directory and compares it to the `.git` directory's snapshots of our work when we use `git status` and `git add` other commands work with the `.git` directory only

We can delete that file, since we do not need it.

Skip to main content

4.6. Getting changes from GitHub

In your browser, make a pull request and then merge it for the changes that we pushed. It will be automatically linked to the issue, since we created the branch from the issue. Linked issues get closed when the PR is merged.

Now, switch to main on your local copy

```
git checkout main
```

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

It tells us that it is up to date, because it is up to date with what our computer knows about GitHub. If we used `git fetch` then `git status` it would tell us we were behind. Then we would have to apply the changes to our working directory after we fetched them. We can fetch and apply them at the same time with `git pull`.

```
git pull
```

```
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), 639 bytes | 639.00 KiB/s, done.
From https://github.com/introcompsys/github-in-class-brownsarahm-1
   3f54148..0169e39  main       -> origin/main
Updating 3f54148..0169e39
Fast-forward
 about.md | 0
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 about.md
```

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 3f54148…0169e39”) then it tells the type of update (“Fast-forward”) and describes a summary of the changes made to each file (created `about.md` empty).

We can see that this updates th working directory too:

```
ls
```

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

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 fill-in-about
```

and git tells us what it has done.

4.7. Editing a file on the terminal

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.

Navigate with your keyboard arrows, add your name to the file. 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.

Now, we check in with git again.

```
git status
```

```
On branch fill-in-about
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

```
git add .
```

we’ll add, since we want these changes

```
git status
```

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

and commit

```
git commit -m "add my name"
```

```
[fill-in-about c28b4ad] add my name
 1 file changed, 1 insertion(+)
```

and push

```
git push
```

```
fatal: The current branch fill-in-about has no upstream branch.
To push the current branch and set the remote as upstream, use

    git push --set-upstream origin fill-in-about
```

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.

However, git helps us out and tells us how to add one. So we do as advised.

```
git push --set-upstream origin fill-in-about
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 265 bytes | 265.00 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 'fill-in-about' on GitHub by visiting:
remote:      https://github.com/introcompsys/github-in-class-brownsarahm-1/pull/new/fill-in-about
remote:
To https://github.com/introcompsys/github-in-class-brownsarahm-1.git
 * [new branch]      fill-in-about -> fill-in-about
branch 'fill-in-about' set up to track 'origin/fill-in-about'.
```

This time the returned message from the remote includes the link to the page to make a PR. Visit that and make and merge the PR.

Note though, that this does not change your local copy.

```
git status
```

```
On branch fill-in-about
Your branch is up to date with 'origin/fill-in-about'.

nothing to commit, working tree clean
```

Even the main branch does not change

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

but it does not *know* that it is behind until we fetch or pull again.

4.8. Review today's class

1. Review the notes
2. Find your team's page on GitHub. It is named like `Spring2023-group-#` join the discussion that I started on your page.
3. Complete the classmate issue in your inclass repo from today. Find a partner from within your assigned team by posting on your team's page. Link to your commits on your badge issue.
4. Try using git using your favorite IDE **or** GitHub Desktop. You can either do the other tasks for this badge, work on a different badge, or add & commit some random files in your inclass repo. Answer the questions below in `gitcompare.md`.

Questions:

```
## Reflection

1. What tool's git integration did you use?
1. Compare and contrast using git on the terminal and through the tool you used.  When would each be better/v
1. Did using a more visual representation help you understand better?
1. Describe the staging area (what happens after git add) in your own words.
2. what step is the hardest for you to remember? what do you think might help you?
```

4.9. Prepare for Next Class

1. Examine a large project you have done or by finding an open source project on GitHub. Answer the reflection questions in `software.md` in your kwl repo. (will be in notes)
2. map out how you think about data moving through a small program and bring it with you to class (no need to submit)

```
## Software Reflection

1. link to public repo if applicable or title of your 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?
1. Try to figure out (remember) how the project works. What types of things, without running the code can you
```

4.10. More Practice

1. Review the notes
2. Find your team's page on GitHub. It is named like `Spring2023-group-#` join the discussion that I started on your page.
3. Download the course website repo via terminal. Append the commands used to a terminalwork.md
4. 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 your gitcommit.md. Compare what happens based on what you can see on GitHub and what you can see with git status.

6. Try using git using your favorite IDE **and** GitHub Desktop. You can either do the other tasks for this badge, work on a different badge, or add & commit some random files in your inclass repo. Answer the questions below in `gitcompare3ways.md`.

Questions:

```
## Reflection

1. What IDE did you use?
1. Was the IDE or GitHub better for you? Why?
1. Compare and contrast using git on the terminal and through your IDE. When would each be better/worse?
1. Did using a more visual representation help you understand better?
1. Describe the staging area (what happens after git add) in your own words. Provide an analogy for it using
2. What programming concepts is the staging area similar to?
2. what step is the hardest for you to remember? what do you think might help you?
```

4.11. Experience Report Evidence

For today's class evidence

4.12. Questions After Today's Class

4.12.1. How does the computer hold the information for my github account after I shut it off?

It stores your account information in files on your system.

4.12.2. How does the terminal know that our upstream server is github?

We cloned the repo from github. This can also be configured manually. We can even add more then one available remote in a single repo. Each branch will have a specific, single upstream, but two branches in the same repo can have different upstreams.

4.12.3. Could I have edited my experience reflection in terminal if I wanted?

Yes!

4.12.4. What is the best way to get comfortible with the terminal?

Lots of practice. Start with the recently posted badges, they're designed to give you good practice.

4.12.5. What is a .md file?

Markdown

4.12.6. Are there any disadvantages to using nano as opposed to something like VIM?

4.12.7. Is there going to be somewhere we can see all of git hubs syntax for the terminal?

I passed out a printed cheatsheet from GitHub with a good amount of the sytax. If you did not get one, let me know.

4.12.8. If I close the terminal is everything we did saved?

Yes, everything we did today in the terminal was directly manipulating files. The log of what commands we ran is even likely saved, but not necessarily in an easy to find and use file.

4.12.9. Is it possible for me to perform everything that I need to with git through the terminal?

git is a command line tool, by defition. The git program is **only** available on the terminal. Other programs like GitHub Desktop or VSCode can do git operations, but typically only a subset. There is also only a subset of git operations available on GitHub.com.

There are however, some GitHub things that there is no command line tool for.

4.12.10. How do I create different versions of my code on GitHub?

branches!

4.12.11. Is `origin` the name used by git terminal to reference github's servers?

In our repos we have seen so far origin has always poointed to a specific repo on GitHub. `origin` is just the default name, so, there could be no `origin` and the repo still have the `remote` be a GitHub repo.

We will see soon that we can work with multiple remotes, for example with a GitHub fork. In that case, there would be two different remotes tht each point to a GitHub repo.

4.12.12. How to commit multiple files at once?

add more than one file to the staging area before commiting.

4.12.13. Is there a way to work with other things (issues, actions, etc) in the terminal?

For GitHub, yes!

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)

Skip to main content

5.1. Important references

Use these for checking facts and resources.

- bash
- git

5.2. Setup

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

```
cd Documents/inclass/systems/github-in-class-brownsarahm-1/
```

and make sure we are on main:

```
git status
```

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

nothing to commit, working tree clean
```

Then we will use fetch to see if we are really up to date

```
git fetch
```

We will also check for open PRs in the browser:

Note this is optional, and only works with the `gh` cli is installed

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

```
Opening github.com/introcompsys/github-in-class-brownsarahm-1 in your browser.
```

and merge thm there.

To get to your main branch.

```
git checkout main
```

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

then fetch again because we now **know** there are changes on GitHub

Skip to main content

```
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), 628 bytes | 628.00 KiB/s, done.
From https://github.com/introcompsys/github-in-class-brownsarahm-1
   0169e39..4b89dff  main       -> origin/main
```

```
ls
```

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

Notice that it updated the .git, but not our working directory.

```
cat about.md
```

```

```

So we can check again

```
git status
```

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

nothing to commit, working tree clean
```

now we see it knows we are behind and tells us how to update

```
git pull
```

```
Updating 0169e39..4b89dff
Fast-forward
 about.md | 1 +
 1 file changed, 1 insertion(+)
```

5.3. Branches review

We can get a list of the branches we have locally

```
git branch
```

```
  2-create-an-about-file
  fill-in-about
* main
```

or see them with their upstream information

```
git branch -vv
```

```
  2-create-an-about-file 57de0cd [origin/2-create-an-about-file] create empty about
  fill-in-about          c28b4ad [origin/fill-in-about] add my name
* main                   4b89dff [origin/main] Merge pull request #5 from introcompsys/fill-in-about
```

5.3.1. Different ways to create a branch and switch

This doesn't work for a branch that does not exist

```
git checkout my_branch_checkedout
```

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

This creates and switches to.

```
git checkout -b my_branch_checkedoutb
```

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

we'll go back to main before the next step

```
git checkout main
```

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

this is not a real command

```
git branch create my_branch_created
```

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

This is a 2 step way to create and switch.

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

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

5.4. Organizing a project (working with files)

Skip to main content

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

We'll go back to main first

```
git checkout main
```

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

then create a new branch from main.

```
git checkout -b organization
```

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

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

```

```

```
git status
```

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

5.5. Files, Redirects, git restore

```
cat README.md
```

```
# GitHub Practice
```

Skip to main content

Echo allows us to send a message to std out. which is a special file.

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

```
age=35
```

We can add contents to files with `echo` and `>>`

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

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
age=35
```

We can redirect other commands too:

```
git status >> curgit
```

we see this created a new file

```
ls
```

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

and we can look at its contents too

```
cat curgit
```

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

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

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

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

```
rm curgit
```

we'll delete this test file.

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

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

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

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

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

Without adding first, git cannot make a commit.

So we will add first then commit.

```
git add .
git commit -m 'start organizing'
```

```
[organization ef45e77] start organizing
 15 files changed, 1 insertion(+)
 create mode 100644 API.md
 create mode 100644 CONTRIBUTING.md
 create mode 100644 LICENSE.md
 create mode 100644 _config.yml
```

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

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

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

We check the file now

```
cat README.md
```

```
age=35
```

It wrote over. This would be bad, we lost content, but this is what git is for!

It is *very very* easy to undo work since our last commit. This is good for time when you have something you have an idea and you do not know if it is going to work, so you mak a commit before you try it. Then you can try it out. If it doesn't work you can undo and go back to the place where you made the commit.

To do this, we will first check in with git

```
git status
```

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

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

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

```
git restore README.md
```

```
git status
```

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

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
age=35
```

Back how we wanted it!

Now we will add some text to the readme

```
 echo "|file | contents |
> > | ------| ------- |
> > | abstract_base_class.py | core abstract classes for the project |
> > | helper_functions.py | utitly funtions that are called by many classes |
> > | important_classes.py | classes that inherit from the abc |
> > | alternative_classes.py | classes that inherit from the abc |
> > | LICENSE.md | the info on how the code can be reused|
> > | CONTRIBUTING.md | instructions for how people can contribute to the project|
> > | setup.py | file with function with instructions for pip |
> > | 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.

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
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 |
> | _config.yml | jupyterbook config for documentation |
> | _toc.yml | jupyter book toc file for documentation |
> | philosophy.md | overview of how the code is organized for docs |
> | example.md | myst notebook example of using the code |
> | scratch.ipynb | jupyter notebook from dev |
```

```
ls
```

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

5.6. Getting organized

First, we'll make a directory

```
mkdir docs
```

next we will move a file there

```
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`

```
ls
```

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

```
touch _toc.yml
```

5.6.1. 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 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/
```

```
_config.yml     _toc.yml        overview.md
```

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

Skip to main content

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_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 achieve a move, but changing a differnt part of the path.

```
ls
```

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

Now we make a new folder:

```
mkdir tests
```

and move all of the test files there:

```
mv test_* tests/
```

```

```

```
ls
```

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

Not

this is
impor
organ
can u
yours

5.7. Recap

Why do I need a terminal

1. replication/automation
2. it’s always there and doesn’t change
3. it’s faster one you know it (also see above)

> So, is the shell the feature that interacts with the operating system and then the terminal is the gui that interacts with the shell?

Skip to main content

> **Important**
>
> if your push gets rejected, read the hints, it probably has the answer. We will come back to that error though

5.8. Review today's class

1. Read today's notes
2. Update your KWL chart with the new items and any learned items.
3. add `branches.md` to your KWL repo and describe how branches work, in your own words. Include one question you have about branches or one scenario you think they could help you with.

5.9. Prepare for Next Class

1. Read through the grading section and all of your feedback as it arrives.
2. Bring git questions or scenarios you want to be able to solve to class on Thursday
3. Update your `.github/workflows/experiencereflection.yml` file as follows: replace the two lines `team-reviewers: |` and `instructors` with `reviewers: <ta-gh-name>` where ta-gh-name is whichever TA is in your group. You can see your group on the organiation teams page named like "Spring 2023 Group X". Make a PR and ask that TA for a review.
4. Answer the following in a comment on your prepare issue. Tag @brownsarahm on the issue

```
# Plan for success

__Target Grade:__ (A,B,..)

## Plan to get there:
- 24 experience badges
- (other badges you plan)

<!-- If you plan any build badges,uncomment the line below and list some ideas, topics from the course websi
<!-- ## Builds -->

<!-- If you plan any explore badges, create a schedule and propose one topic for your first explore badge --
```

5.10. More Practice

1. Read today's notes
2. Update your KWL chart with the new items and any learned items.
3. Learn about GitHub forks (you can also use other resources)
4. 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.

5.11. Experience Report Evidence

Save your history with:

```
history > activity-2023-02-07.md
```

then append your git status, and the contents of your github-in-class and github-in-class/docs with `--------------` to help visually separat the parts.

```
echo "--------------" >> activity-2023-02-07.md
git status >> activity-2023-02-07.md
echo "--------------" >> activity-2023-02-07.md
ls >> activity-2023-02-07.md
echo "--------------" >> activity-2023-02-07.md
ls docs/ >> activity-2023-02-07.md
```

then edit that file (on terminal, any text editor, or an IDE) to make sure it only includes things from this activity.

5.12. Questions After Today's Class

5.12.1. how would we move every file we have into a new folder

```
mv old_folder/* new_folder/
```

5.12.2. Can we create a pull request through `bash`?

Strictly speaking `bash` does not do *any* git operations or any GitHub operation, or any other version control operations for other version control systems. `bash` is a particular scripting language that we are using on our terminal. We could use a different one, and git can still be used.

Pull requests are not a git feature, so they are not done by the git program at all. They are done differnt ways for different projects and hosting plan. For example, the git source code is mirrored (copied) on GitHub, but that is not where they actually do the work of maintaining it. They accept "pull requests" via e-mail.

5.12.3. Can we merge using the terminal?

Yes, we will see `git merge` soon.

5.12.4. I would like to know more about regular expressions as this is something I have seen but don't really know anything about.

We will use a few examples of them, but we will not go too in depth with them. We'll focus on the places they can be used and basic regex for the most common cases. However, this is an *excellent* topic for an explore or build badge.

5.12.5. What other symbols can be used to move files like *?

The wildcard `*` can be used not only for moving files, but in lots of places. It is a special character in regular expressions and glob matching. We will see some different ones, but this is also a good topic for an explore badge.

5.12.6. Will we use vim in the terminal?

Yes, we will see pipes soon.

5.12.7. When you said that it only changes file path, do you mean its storage placement in the hard drive stays the same?

Yes, exactly.

5.12.8. Will we be learning out to open and use a .py file?

Indirectly. We will do a tiny bit of editing and running code in python and C, but not too much. We will learn more about what files are, how they work in general, and the role of interpreters and compilers so that you will be able to use the terminal for any language you encounter.

5.12.9. To make sure, mv can rename and move files?

Yes, it can do achieve both of those outcomes, because in the file system they *are the same thing.* “moving a file” does not have to actually copy and rewrite the file in a different part of the hard drive, but instead it changes the path of the file. We typically think about the path as having two parts: the location and the name, but technicall there is a single *path* to the whole file. `mv` can change any part of the path.

5.12.10. How can we delete branches?

There is a delete option on `git branch`

5.12.11. What exactly are we doing with our teams that we were assigned?

Instructions will be given as you need to respond to them, for now just reply to discussions on the team page.

5.12.12. Should I start using terminal as a replacement of finder to move files/folders?

If you are comfortable with it. It will help you get more comfortable faster.

5.12.13. what is a .yml file?

it is YAML which is typically used to store settings and configurations. GitHub uses it for the workflows, and a lot of other tools I interact with use it for configurations. It stores key-value pairs.

5.12.14. Will all the files we used in the terminal today will be on the github website?

After we push them, yes.

5.12.15. Is there an official list of all git commands?

Skip to main content

cheatsheet is also good.

6. What if I edit the file in two places?

One of the biggest advantages of git is that it is distributed. This gives us great flexibility, but also allows us to make things complicated.

Today we will see how to fix this.

6.1. Get set up

We left off in the middle of some work on tuesday

```
git status
```

```
On branch organization
Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        modified:   README.md
        deleted:    _config.yml
        deleted:    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 thinks we deleted files because we moved them into an untracked folder.

I'll add just the one folder

```
git add docs/
git status
```

```
On branch organization
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:    overview.md
        deleted:    test_alt.py
        deleted:    test_help.py
        deleted:    test_imp.py
        deleted:    tests_abc.py
```

```
        tests/
```

Now it thinks I deleted and created new files Now if we dd all

```
git add .
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   README.md
        renamed:    _config.yml -> docs/_config.yml
        renamed:    overview.md -> docs/_toc.yml
        renamed:    test_alt.py -> docs/overview.md
        renamed:    test_help.py -> tests/test_abc.py
        renamed:    test_imp.py -> tests/test_alt.py
        renamed:    tests_abc.py -> tests/test_help.py
        new file:   tests/test_imp.py
```

It sees that we moved files. Sine these were empty, it cannot match them all up very well.

Now we can commit

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

```
[organization 812245d] start organizing
 8 files changed, 19 insertions(+)
 rename _config.yml => docs/_config.yml (100%)
 rename overview.md => docs/_toc.yml (100%)
 rename test_alt.py => docs/overview.md (100%)
 rename test_help.py => tests/test_abc.py (100%)
 rename test_imp.py => tests/test_alt.py (100%)
 rename tests_abc.py => tests/test_help.py (100%)
 create mode 100644 tests/test_imp.py
```

and confirm we’re all set

```
git status
```

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

then push

```
git push
```

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

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

Skip to main content

we have to create the upstream.

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

```
Enumerating objects: 11, done.
Counting objects: 100% (11/11), done.
Delta compression using up to 8 threads
Compressing objects: 100% (7/7), done.
Writing objects: 100% (9/9), 1.42 KiB | 1.42 MiB/s, done.
Total 9 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
remote:
remote: Create a pull request for 'organization' on GitHub by visiting:
remote:      https://github.com/introcompsys/github-in-class-brownsarahm-1/pull/new/organization
remote:
To https://github.com/introcompsys/github-in-class-brownsarahm-1.git
 * [new branch]      organization -> organization
branch 'organization' set up to track 'origin/organization'.
```

Then we'll make a PR for that branch, but leave it open.

6.2. Editing main when a PR is open.

First we switch to main

```
git checkout main
```

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

and make sure it is up to date

```
git pull
```

```
Already up to date.
```

Next we will edit the README to add somethign to the first line

```
nano README.md
```

on the main branch it starts like :

```
# GitHub Practice

Name: Sarah Brown
```

and we edit to:

```
Name: Sarah Brown
```

Then we checkin

```
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:   README.md

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

add and commit

```
git add .
git commit -m 'update title'
```

```
[main f2844b2] update title
 1 file changed, 1 insertion(+), 1 deletion(-)
```

and push to GitHub

```
git push
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 356 bytes | 356.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-in-class-brownsarahm-1.git
   4b89dff..f2844b2  main -> main
```

When we check in our PR, we are still all set. git has checked the changes mad and they are consistent or do not conflict. This means that git knows how to merge this branch without any help from a person to tell it what is important.

6.3. Making a conflict

```
nano README.md
```

again we will edit the same file, but this time we will edit in a way that conflicts with how we changed it on the `organization` branch.

We will add text on line 4.

```
Name: Sarah Brown
major: EE
```

and check with git

```
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:   README.md

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

and add commit, and push

```
git add .
git commit -m 'add major'
```

```
[main cffcf05] add major
 1 file changed, 1 insertion(+)
```

```
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), 366 bytes | 366.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-in-class-brownsarahm-1.git
   f2844b2..cffcf05  main -> main
```

If we look on the organization branch

```
git checkout organization
```

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
age=35
```

Skip to main content

```
> | 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 |
```

we see the file is different.

6.4. Resolving Merge Conflicts on GitHub

We can see it on the PR now and GitHub allows us to resolve it in the browser.

GitHub docs are the best reference on what we did here.

we resolved the issue in the browser then merged the conflict.

6.5. Making and resolving a conflict locally

We will go back to main and get up to date

```
git checkout main
```

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

```
git pull
```

```
remote: Enumerating objects: 7, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 751 bytes | 187.00 KiB/s, done.
From https://github.com/introcompsys/github-in-class-brownsarahm-1
   812245d..56f29ca  organization -> origin/organization
Already up to date.
```

Imp
This i
misse
resol

Now we will edit the about file one way on our local copy and a different way on GitHub without pulling or pushing

```
nano about.md
```

Once the two versions are different, we pull the contents

Skip to main content

```
git  pull
```

```
remote: Enumerating objects: 8, done.
remote: Counting objects: 100% (7/7), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 4 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (4/4), 1.23 KiB | 314.00 KiB/s, done.
From https://github.com/introcompsys/github-in-class-brownsarahm-1
   cffcf05..8e2fe11  main       -> origin/main
Updating cffcf05..8e2fe11
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
```

… and it fails. We need to commit our local changes first or git cannot even do its work to compare.

It will not overwrite our working directory with un committed changes.

```
git add .
git commit -m 'add jacket'
```

```
[main 9cb0323] add jacket
 1 file changed, 2 insertions(+)
```

```
git status
```

```
On branch main
Your branch and 'origin/main' have diverged,
and have 1 and 5 different commits each, respectively.
  (use "git pull" to merge the remote branch into yours)

nothing to commit, working tree clean
```

Now it tells us w have a conflict, or the two branches that are suppose to be the same have diverged.

When we do what it says now,

```
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.
```

Skip to main content

```
git pull --rebase
```

```
Auto-merging about.md
CONFLICT (content): Merge conflict in about.md
error: could not apply 9cb0323... add jacket
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 9cb0323... add jacket
```

this time we edit the same way, but locally.

```
nano about.md
```

```
git status
```

```
interactive rebase in progress; onto 8e2fe11
Last command done (1 command done):
   pick 9cb0323 add jacket
No commands remaining.
You are currently rebasing branch 'main' on '8e2fe11'.
  (fix conflicts and then run "git rebase --continue")
  (use "git rebase --skip" to skip this patch)
  (use "git rebase --abort" to check out the original branch)

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

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

then do as git says

```
git rebase --continue
```

```
about.md: needs merge
You must edit all merge conflicts and then
mark them as resolved using git add
```

we have to add first

```
git add about.md
git rebase --continue
```

```
[detached HEAD 6a2e1cc] add jacket
 1 file changed, 3 insertions(+)
Successfully rebased and updated refs/heads/main.
```

```
git status
```

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

nothing to commit, working tree clean
```

6.6. Review today's class

1. create `gitadvice.md` and write tips for how often to commit and how to avoid merge conflicts. Include at least 3 tips.
2. 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
3. clone your group repo.
4. work offline and add your contribution and then open a PR
5. review a class mate's PR.

6.7. Prepare for Next Class

1. Bring questions about git to class on Wednesday.
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. Read sections 1.1,1.2, and 1.3 of the pro git book this is mostly review at this point, but we are going to go into more of how git works next, so you need to make sure these concepts are all sorted out. Comment on your prepare issue if reading helped clarify confusion, made you more confused, or gave you new understanding. Either explain what you learned or ask a question you have. Tag @brownsarahm on the issue.

6.8. 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. 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. create an issue on your group repo for a resource you want to review. Make sure that your contribution does not overlap with one that another member is going to post.
3. clone your group repo.
4. work offline and add your contribution and then open a PR. Your reivew should help a classmate decide if that reference material will help them or not.
5. review a class mate's PR.

6.9. Experience Report Evidence

link to your github inclass repo's commit history.

6.10. Questions After Today's Class

6.10.1. Is there a way to configure git to update automatically or a command we can run to update it?

Manual, unfortunately.

6.10.2. Whenever a merge conflict occurs will there always be some indicator that it occured and instructions on how to fix it on git or github?

Yes! no guessing.

6.10.3.

We will see this soon.

6.10.4. If there are multiple merge conflicts, does the terminal iterate through each conflict once the prior one is fixed?

It tells you the first one's file and you have to find them all within the file.

6.10.5. how to get out of VIM when you accidently enter git commit without -m

escape, `:wq`, enter/return

if the escape button does not work, sometimes you have pressed another key that puts it in a different mode, then you can use command/control + c and then `:wq` +enter.

As a last resort, you *can* close the terminal window and open a new one. Everything important that git does edits a file on your computer, git does not use a lot of memory that you can lose by quitting. If you quit, your staging area will stay exactly how you left it. You can then navigate back to that directory and start your commit again with `-m`.

6.10.6. what does git rebase actually do?

the git docs describe this well

6.10.7. What's the importance of merging?

It's the way to combine two branches' of work

6.10.8. Questions we will answer later

- Is there any way to reverse what you did just like how you reverse your commits in before your staging area?
- How does git track all of the information between commits and merges and branches?

6.10.9. What if resolving a merge conflicts causes other issues?

By definition, it cannot.

6.10.10. Best way to prevent different/colliding branches / issues with local git and github not being the same?

This is an exercise for the review badge, so I am declining to answer here, but we will engage with you on your badges.

6.10.11. when you rename a file, how does git know that you renamed it (how does it know it's the same file even though it has a new name)?

It matches the content of the files.

6.10.12. How do we fix accidental merges?

The goal is to not have to, that's why GitHub makes you click twice to merge a pull request. It is possible though. It's on the FAQ: https://introcompsys.github.io/spring2023/faq/github.html#help-i-accidentally-merged-the-feedback-pull-request-before-my-assignment-was-graded. This example is taken from my Data Science class, where they commit to main and I grade a PR from main into Feedback. If they accidentally merge, they need to leave their main branch alone and hard reset their Feedback branch. In other cases you may want to reset the main branch or a develop branch. The procedure is the same, only the branch names change.

6.10.13. the prepare work before class and the work we do in class are for the same grade?

Yes, the prepare for class is a part of the experience badge for the next class after it is posted along with in class. This is because the prepare for class work is due before the next class. Increasingly, class sessions will rely on you having completed those items.

7. Why are these tools like this?

Today we're going to do a bit more practical stuff, but we are also going to start to get into the philosophy of how things are organized.

Understanding the context and principles will help you remember the what and help you understand when you shoudl do things as they have always been done and when you should challenge and change things.

7.1. How do I decide how to resolve a merge conflict?

First lets make a new branch

```
git checkout -b examplebranch
```

```
Switched to a new branch 'examplebranch'
```

```
git status
```

Skip to main content

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

Now we will add some text to a file that “tells a story” of what we might have actually done.

```
echo "edit code near a bug I did not know about" > important_classes.py
```

and commit this change

```
git add .
git commit -m 'new code'
```

```
[examplebranch bb4e0d8] new code
 1 file changed, 1 insertion(+)
```

Then back on the main branch

```
git checkout main
```

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

we will pretend a bug got fixed in that same file. This is a realistic-ish scenario. You might be workign on adding a new feature then get a bug report and fix the bug on a different branch and get it merged into main so that others using the file

```
echo "fix a bug" >> important_classes.py
```

```
git add .
```

```

```

```
git commit -m 'bug fix'
```

```
[main 4fa9114] bug fix
 1 file changed, 1 insertion(+)
```

```
git status
```

```
On branch main
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

```

```

```
git checkout examplebranch
```

```
Switched to branch 'examplebranch'
```

```
git merge main
```

```
Auto-merging important_classes.py
CONFLICT (content): Merge conflict in important_classes.py
Automatic merge failed; fix conflicts and then commit the result.
```

```
cat important_classes.py
```

```
<<<<<<< HEAD
edit code near a bug I did not know about
=======
fix a bug
>>>>>>> main
```

```
nano important_classes.py
```

```
git status
```

```
On branch examplebranch
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   important_classes.py

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

7.2. How do I exit vi?

```
vi
```

If you start it this way, you exit by pressing only `q` beacuse it will already have the `:` at the bottom. if you are in `~insert~` mode then you have to press `esc` first and then `:` then you may want `w` to save first before you `q`uit

7.3. Why are we studying developer tools?

Skip to main content

> 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

7.4. Unix Philosophy

wiki

- 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.

```
echo "hello"
```

```
hello
```

```
echo "hello" > greeting.md
```

```

```

```
echo $PATH
```

```
/Users/brownsarahm/.rbenv/shims:/Library/Frameworks/Python.framework/Versions/3.8/bin:/opt/anaconda3/bin:/opt
```

```
award --help
```

```
Usage: award [OPTIONS] BADGE_NAME

  award a badge to a student and output the signature as a receipt

  parameters - badge_name : string     the name of the badge
  formatted like type.YYYY-MM-DD for      type in {experience, review,
  practice,community} or type.keyword for type     in {explore, build,
  community}

Options:
  -s, --student-gh-name TEXT
  -p, --gradebook-path TEXT
```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-in-class-brownsarahm-1
```

7.5. 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.

These are not mutually exclusive we will use them all together, and trade off.

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.

7.5.1. 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).

7.5.2. Components

> Take it apart, asess 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.

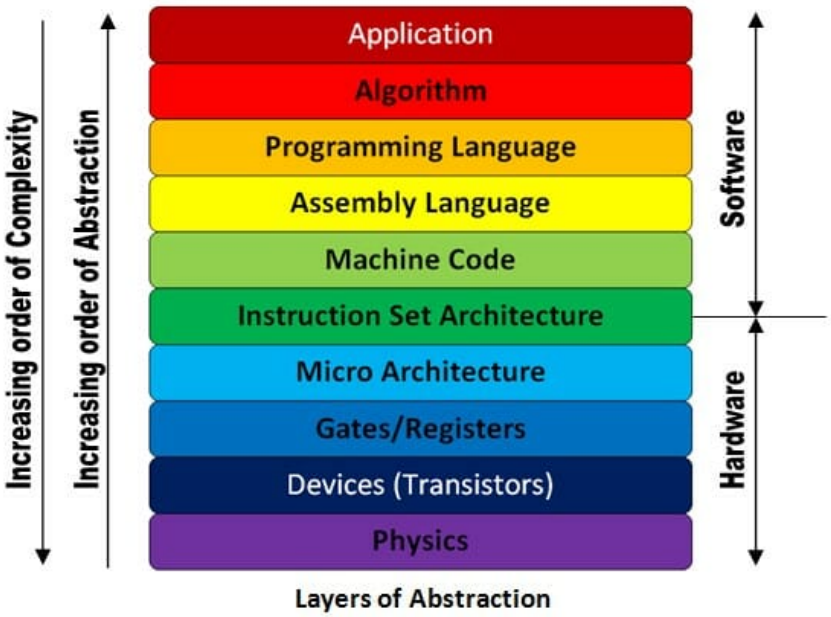
This idea, however that we can use another tool to understand each component is an important one. This is also a way to again, take care and study each piece even within a software-alone system without worrying about the hardware.

7.5.3. 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

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.

Let's take a simple example, let's say we are talking about a simple search program that we wrote that finds xx. We can say that you put in a part of a file name and it shows you all the ones with a similar name. That description is at the application level it gives the high level behavior, but not the step by step of how it does it. Let's say we implemented it using bubble search then searches by … That's the algorithm level, this is still abstract and could be implemented in different ways, but we know the steps and we can use this to know some things about how fast it will be, what types of result swill make it slower, etc. At the programming level language then we know which language it was done in and we see more details. At this level, we can see the specific data structures and controls structures. These implementation details can also impact performance in terms of space(memory) or time. Still at this level, we do not need to know how the actual hardware works, but we see it in increasing detail. At each level we have different types of operations. At the application we might have input, press enter, get results. At the algorithm we have check the value, compare. At the programming language level we need more specifics too, like assign or append.

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.

Machine code translates to binary from assembly.

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

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.

Note

For example, Bayesian statistics is a philosophy that treats probability as subjective uncertainty instead of as frequency. This has some interpretative differences, but most importantly it means that we always need an extra factor (multiplied term) in our calculations. This makes all of the math **much** more complex. For many decades Bayesian statistics was not practical for anything but the simplest models. However, with improvements to computers, that opened new options at the algorithm level. The first large scale application of this type of statistics was by Microsoft after their researchers built a Bayesian player model for player matching in Halo.

7.6. 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.
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?
```

7.7. Prepare for Next Class

1. 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.
2. If you like to read about things before trying them, skim the jupyterbook docs.
3. Think about and be prepared to reply to questions in class about your past experiences with documentation, both using it and

Skip to main content

7.8. More Practice

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 and why.
3. 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 the effective? why or why not? When do they work
- what are you most interested in trying that might be different?
```

7.9. Experience Report Evidence

complete the review or practice badge an link that to your experience report.

Also, update your `experiencereport.yml` to match the following but with `<ta-gh-name>` changed to be your group's TA's actual gh user name.

```
name: Experience Report (makeup)
on:
  workflow_dispatch:
    inputs:
      date:
        description: 'missed class date'
        required: true
        type: string


jobs:
  createPullRequest:
    runs-on: ubuntu-latest
    steps:
      - uses: actions/checkout@v3

      - name: Make changes to pull request
      # TODO: modify to create template file
        run: |
          exptitle="experience-${{ inputs.date }}.md"
          cp .templates/experience-report.md experiences/$exptitle
      - name: Create Pull Request
        id: cpr
        uses: peter-evans/create-pull-request@v4
        with:
          token: ${{ secrets.GITHUB_TOKEN }}
          commit-message: initialize experience report
          committer: GitHub <noreply@github.com>
          author: ${{ github.actor }} <${{ github.actor }}@users.noreply.github.com>
          signoff: false
          branch: experience
          branch-suffix: timestamp
          title: 'Experience Report ${{ inputs.date }}'
          body: |
            Checklist:
            - [ ] Merge prepare work into this PR
            - [ ] Link prepare issue to this PR
            - [ ] Complete experience report
            - [ ] Add activity completion evidence per notes
          reviewers: <ta-gh-here>
```

7.10.1. Questions that are good explore badge topics

- Why did windows deviate so heavily from these principles?
- Are there new text editors for the terminal? 2015 - present?
- what is the difference between Unix and Linux?

7.10.2. Does learning a little bit about the other layers in layers of abstractions help create better software or applications?

I think so, in some cases at least. A lot of the time the other layers are hidden because the abstractions work well. Sometimes though you hit a limit of some sort and knowledge of other layers canhelp you realize that or even overcome it.

7.10.3. Are there options to Nano that we can use, and in what case would it be worth using something else.

For this class, I recommend nano because it is more friendly. vim and emacs are more popular and powerful, but less friendly.

They can be worth t when you have more complex tasks to do

7.10.4. When is it better to work in the terminal as opposed to GitHub desktop?

Once you get used to the terminal, it saves *a lot* of time over using the GUI. Also, you can have better ergonomics using only the keyboard and less mouse use.

The biggest limitation of GitHub desktop is that it only works for GitHub repositories. The command line git program can be used with different hosts.

Also, on the terminal you can automate and use other commands in combination with your git actions.

7.10.5. Is assembly language inefficient to be programming in?

It is maybe a less efficient use of your time, but otherwise not strictly speaking. However, to get good performance from your code, you have to manually make it efficient. In a compiled languge, when you compile it, it can be optimized for you.

7.10.6. Are there different langauges for machine code?

Its not in a language per se, there are different kinds and different specifications, but none of it is human readable like the programming languages you have used to date.

7.10.7. What does assembly code look like?

We will see more soon but it consists of simple, low level instructions and addresses to apply them to.

8. How do programmers communicate about code?

8.1. Wrap up

```
git status
```

```
On branch examplebranch
You have unmerged paths.
  (fix conflicts and run "git commit")
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)
        both modified:   important_classes.py

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

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

```
cat important_classes.py \
```

```
>
new code that interates bug fix and new feature
```

```
git add important_classes.py
```

```

```

```
git status
```

```
On branch examplebranch
All conflicts fixed but you are still merging.
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   important_classes.py

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

```
git commit -m 'resolve conflict with bug fix and feature'
```

```
[examplebranch 12bbbad] resolve conflict with bug fix and feature
```

```
git status
```

```
  (use "git add <file>..." to include in what will be committed)
        greeting.md

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

```
git checkout main
```

```
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 2 commits.
  (use "git push" to publish your local commits)
```

```
cd ..
```

```

```

```
git status
```

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

8.2. Why Documentation

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

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

In particular documentation tools are really good examples of:

- pattern matching
- modularity
- automation
- building

8.3. *conceptual topics*

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

Skip to main content

practical skills

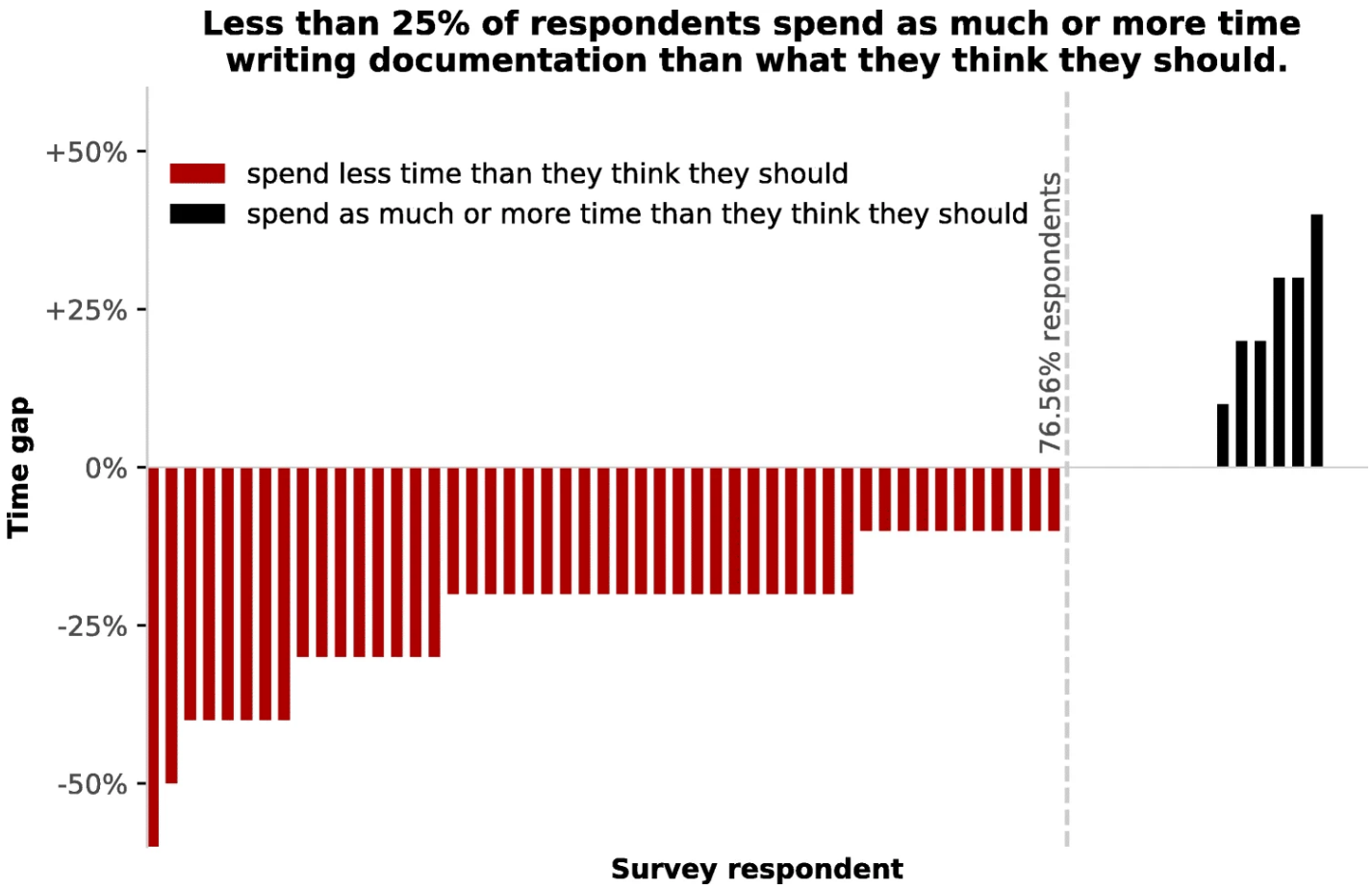
8.4. What is documentation

documentation types table

from ethnography of docuemtnation data science

8.4.1. Why is documentation so important?

we should probably spend more time on it



via source

8.5. So, how do we do it?

Documenation Tools

write the docs

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

Skip to main content

Jupyterbook wraps sphinx and uses markdown instead of restructured text. We're goign to use this.

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

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

Your book template can be found at

    tiny-book/

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

You can make it with any name:

```
jupyter-book create example
```

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

Your book template can be found at

    example/

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

```
ls
```

Each one makes a directory

```
example                       tiny-book
github-in-class-brownsarahm-1
```

```
cd tiny-book/
ls -a
```

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

8.6.1. starting a git repo locally

```
git init .
```

```
hint: Using 'master' as the name for the initial branch. This default branch name
```

```
hint:   git config --global init.defaultBranch <name>
hint:
hint: Names commonly chosen instead of 'master' are 'main', 'trunk' and
hint: 'development'. The just-created branch can be renamed via this command:
hint:
hint:   git branch -m <name>
Initialized empty Git repository in /Users/brownsarahm/Documents/inclass/systems/tiny-book/.git/
```

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

8.6.2. 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
```

```

```

```
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

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

and we will commit the template.

```
git add .
```

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

```
[main (root-commit) 267e9ae] init jupyterbook
 9 files changed, 342 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
```

```
git commit --help
```

8.6.3. 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`)

- config defaults
- toc file formatting rules

Some files contain the content. 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.

Further Reading

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

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

```
cat _config.yml
```

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

Not

CRLF

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

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

git do
is ado

```
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
```

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

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

```
cat _toc.yml
```

```
# 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
```

```
jupyter-book build .
```

```
Running Jupyter-Book v0.13.1
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 for /Users/brownsarahm/Documents/inclass/systems/tiny-book/references.bib in bibtex cache... not fou
parsing bibtex file /Users/brownsarahm/Documents/inclass/systems/tiny-book/references.bib... parsed 5 entries
myst v0.15.2: MdParserConfig(renderer='sphinx', commonmark_only=False, enable_extensions=['colon_fence', 'do
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
Executing: markdown-notebooks in: /Users/brownsarahm/Documents/inclass/systems/tiny-book
```

```
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
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
Or paste this line directly into your browser bar:
    file:///Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html/index.html

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

Try it yourself

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

```
ls
```

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

```
ls _build/
```

```
html           jupyter_execute
```

```
ls _build/html/
```

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

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

8.7. Ignoring Built files

```
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)
```

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

Git helps us with this with the .gitignore

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

```
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 "ignore build"
```

```
[main be4ae7c] ignore build
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

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

For today, create an empty github repo shared with me.

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.

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

```
git remote
```

```
origin
```

```
git status
```

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

```
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), 16.47 KiB | 5.49 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/tiny-book-brownsarahm-1.git
 * [new branch]      main -> main
branch 'main' set up to track 'origin/main'.
```

```
git push --help
```

```

```

```
ls
```

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

Glossary Links for community badge

See the issue I created and more generally, any issue that I tag community is a chance

8.9. Review today's class

1. Make your kwl repo into a jupyter book. Review the notes carefully for what files are required to make `jupyter-book build` run. Ignore your build directory.
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

3. Learn about the documentation ecosystem in a language that you know besides Python. In `docs.md` include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources.

8.10. Prepare for Next Class

1. Try exploring your a repo manually and bring more questions
2. Make sure that you have submitted and gotten feedback on your plan for the course. (Feb 7 prepare for class)

8.11. More Practice

1. Make your kwl repo into a jupyter book. Review the notes carefully for what files are required to make `jupyter-book build` run. Ignore your build directory.
2. Add one of the following features to your kwl repo:
    - a glossary both the terms and linknig to their use
    - substitutions
    - figure
3. Learn about the documentation ecosystem in another language that you know. In `docs.md` include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources.

8.12. Experience Report Evidence

Link to your tinybook repo.

8.13. Questions After Today's Class

8.13.1. How does the jupyter-book automatically translate html?

It runs a parser on the markdown and adds those into templates based on the settings.
Markdown is designed to be translated to HTML. Working with templates is like at the file level, but similar to using string formatting in programming(C++, Python, javascript, Rust).

more in depth exploration here is a good explore badge

8.13.2. what is a .yml file?

Be sure to read the questions after every class, this has been asked and answered. I also added yml to the glossary.

8.13.3. Is it a good habit to pretty much git status every other line just to keep track of where I am at?

class, but even when I am working, I use it a lot. It is also good to get confirmation that git is where you think it is.

8.13.4. Is there a cheatsheet for jupyter-book or just the documentation?

The documentation's `RESOURCES` section has a cheatsheet for Myst-Markdown and a configuration reference which are what I use a lot.

8.13.5. What is the difference between Jupyter Book, Lab, and Notebook?

Jupyter Notebook is a single stream of computational analysis. Jupyter Lab is a more IDE like interface for doing compuational analyses. Both are part of project jupyter and on GitHub

Jupyter book is for publishing book like documents as websites and to other forms designed to be compatible with jupyter notebooks, but is a part of a separate executable books project. It is specialized for cases where there is computation in the code. See their gallery for examples. I use it for CSC310 that has code and plots in the notes

8.13.6. Is documentation a requirement when creating a programming language?

If you want people to be able to use your language then pretty much. That said, not all languages are open source and have easily accessbile, official documentation. Python, Rust, Asteroid (developed here at URI), Ruby and Stan among many others do. On the other hand the C++ language does not have any official documentation. There is a C++ standard that you can purchase, but no official documentation.

8.13.7. Can jupyterbook convert other languages to html or only python?

It doesn't convert plain python to html, it can run jupyter notebooks. Jupyter notebooks can run many different kernels. Jupyter-book is an opinionated distribution of sphinx which can also be used to document other languages like C++

8.13.8. this seemed like a tedious way of creating a repository is there maybe a shorter way?

The only thing that was for making the repository was the `git init .` step and then to link to GitHub, `git remote add`.

8.13.9. Should I use Jupyterbook for creating websites showcasing some of my programming projects?

You totally can. You could also use sphinx which is more customizable. A sphinx gallery might be of interest.

> **Build Badge**
>
> You could build a profile website using a tool like this or other jamstack tool that showcases projects for a build badge.

8.13.10. What happends to files in .gitignore once the repo is pushed?

Absolutely nothing. They exist in your working directory but they are not in the `.git` directory. These files are not tracked by git locally and not backed up by being copied to a server.

Skip to main content

8.13.11. does jupyterbook have a hosting service or should we use something else like github pages?

Jupyterbook only provides the builder, but they do provide instructions for hosting with mulitple services.

8.13.12. What if later I do not want to ignore some files anymore?

The `.gitignore` file is a plain text file (try `cat .gitignore`). You can edit it anytime.

9. What *is* git?

9.1. Admin

9.1.1. get credit for this class toward your major

If you have not already [get the “curriculum modification” form from the college]](https://web.uri.edu/artsci/academics/student-resources/forms/). Fill it out in the following way:

- substitution of: CSC392 Intro to Computer Systems For: 300 level elective
- no rationale
- send the form to Professor Dipippo, ldipippo@uri.edu

9.1.2. Progress

- find an issue from me on your KWL repo with titled “progress as of 2023-02-18 HH:MM:SS” ask questions there.
- merge PR from me titled “udpate experience report action”

> **Important**
>
> read comments carefully, if we correct something for you a few times and you continue to frequently make the same error, we will stop approving work with that error**

9.1.3. Logistics Clarification

badge info

Warning

The bottom of the page had rendering issues during class, but is fixed now.

See the bug I had today.

CDN for `mermaid.js` as imported through the sphinx extension that allows the jupyter book (course website) to render mermaid diagrams is failing. I put in an issue to report the bug. After class, I also figured out how to fix it and submitted a PR as well. I tested the jupyter book build offline with the version of the package I edited, and it works. So, I have attempted making my fork version of the package the version that GitHub installs.

- commit with change to requirements.tx
- where it uses requirements.txt and builds the book

Not

I shar
bug I
down
worki
scena
depe
error
had is
a mo
phas
comp

9.2. Git is a File system

Important

git book is the official reference on git.

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

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

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

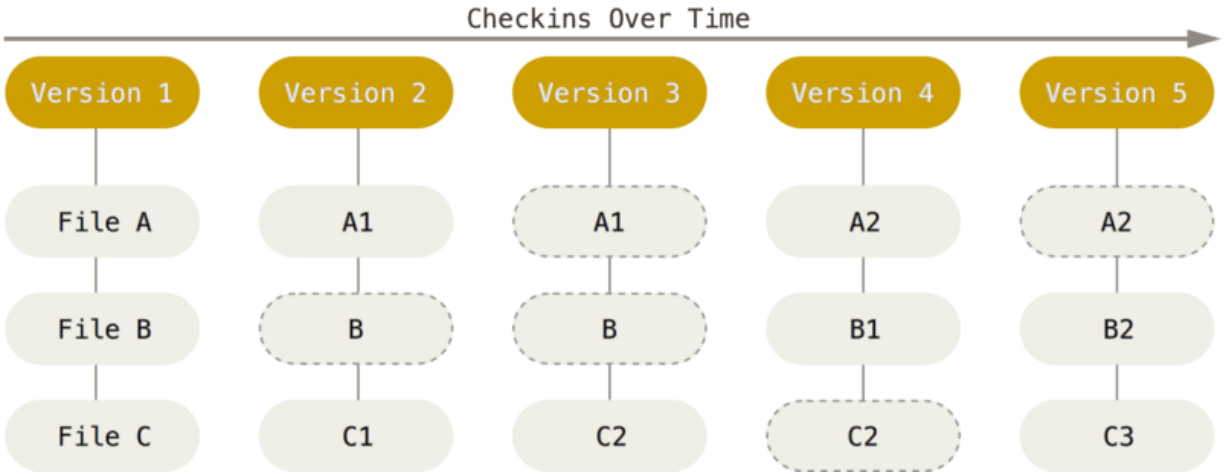
what types of programming have you seen that use key- value pairs?

```
Python 3.8.3 (default, Jul  2 2020, 11:26:31)
[Clang 10.0.0 ] :: Anaconda, Inc. on darwin
Type "help", "copyright", "credits" or "license" for more information.
>>> md = {'key1':1}
>>> md[key1]
Traceback (most recent call last):
  File "<stdin>", line 1, in <module>
NameError: name 'key1' is not defined
>>> md['key1']
1
>>> exit()
```

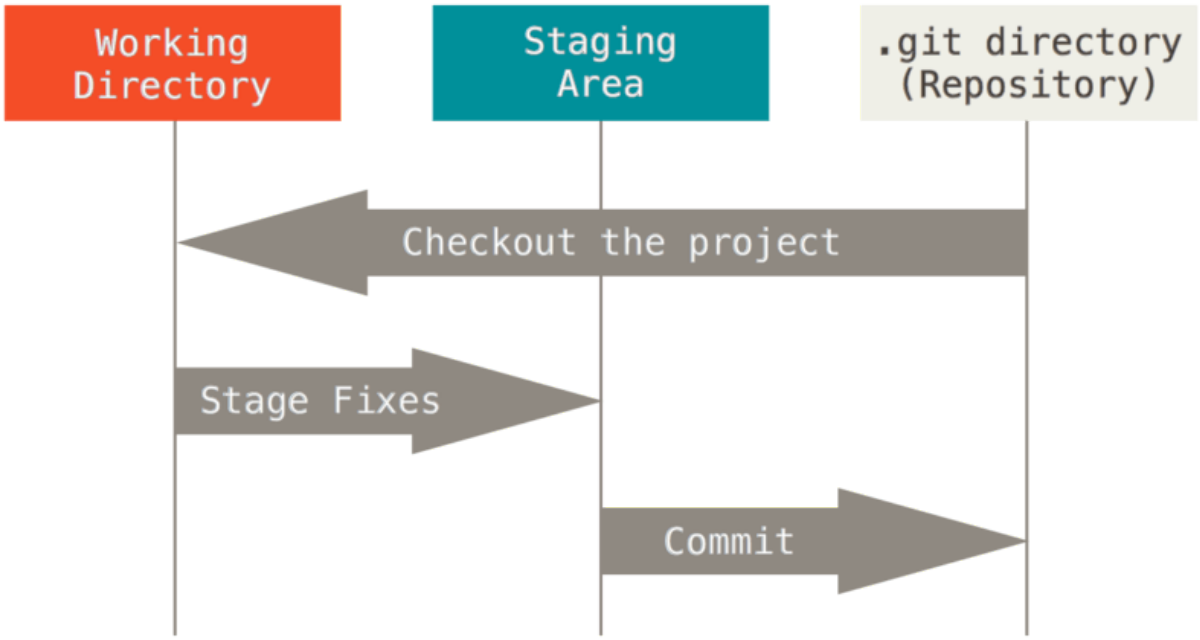
What this means is that you can insert any kind of content into a Git repository, for which Git will hand you back a unique key you can use later to retrieve that content.

9.3. Git is a Version Control System

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



it uses 3 stages:



9.4. 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.

9.5. Git is distributed

What does that mean?

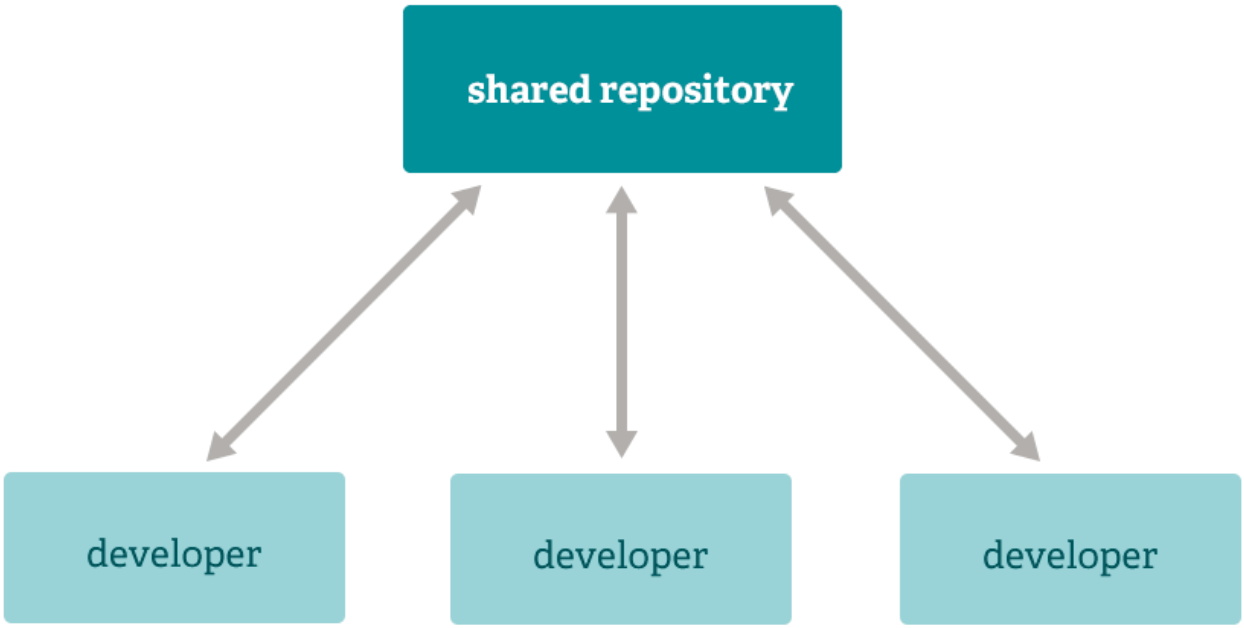
Git runs locally. It can run in many places, and has commands to help sync across remotes, but git does not require one copy of the repository to be the “official” copy and the others to be subordinate. git just sees repositories.

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.

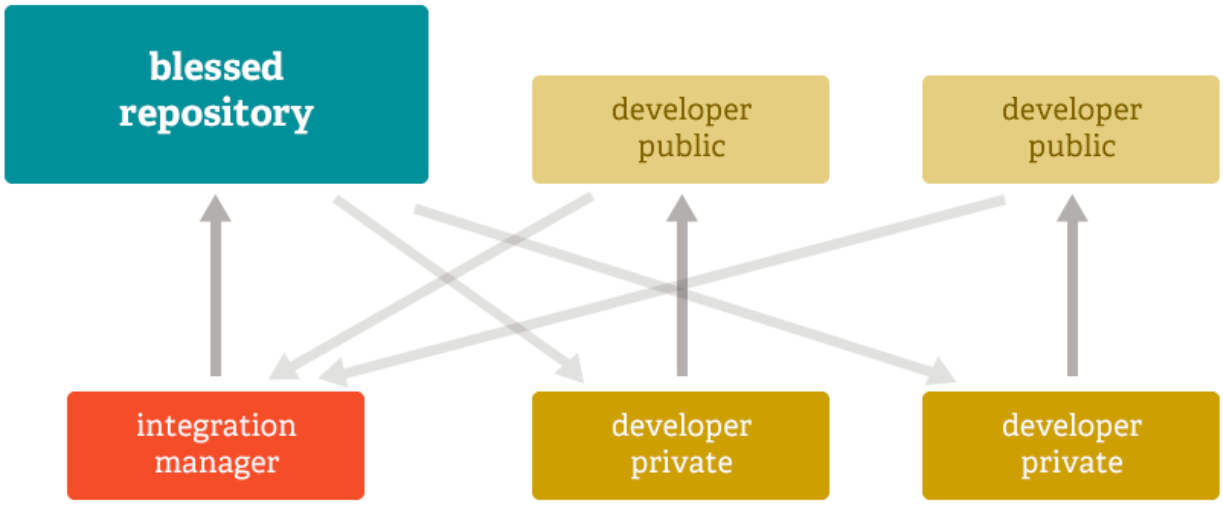
It’s about the workflows, or the ways we socially *use* the tool.

Some example workflows

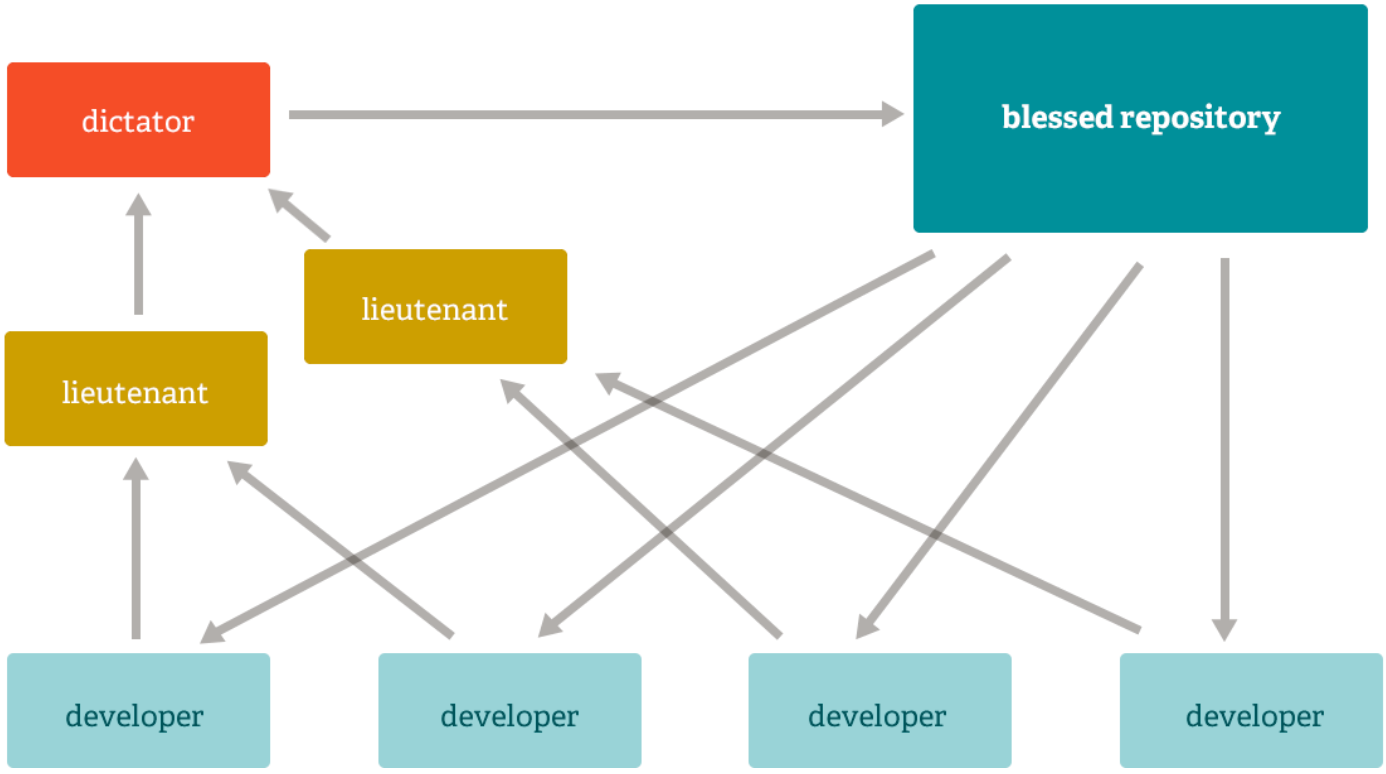
9.5.1. Subversion WOrkflow



9.5.2. Integration Manager



9.5.3. dictator and lieutenants



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

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

```
cd ../github-in-class-brownsarahm-1/
```

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.

Skip to main content

```
objects//0d/d61172a7dad0ac78ed6ca1788130a607983973
objects//66/63ed3256515f3b45209c380c76bc2c47225175
objects//57/de0cd87ec14926273fd4760b94c0ea6f5b87d3
objects//03/29e78f9f8cc84f80808b907b044295f1f82594
objects//6a/2e1cc65204d0c05acc477e7d5e704f989a68dc
objects//56/f29cad4c33dc9e5c194e6f774024bca756b6ee
objects//34/6853b8cf891a2783619506221965ffd35ed99d
objects//5f/3e4cc9381a15bf59b928af1381cb5f2518cedb
objects//9d/b284a866b196a79fd633c439b1dce783740137
objects//9c/b0323be29dab2db451e9975fa2e65b37190ff1
objects//d9/328fbe3ec302b590077eee7ad599b7dc246685
objects//bb/4e0d87b6dc7d6c48e5515f6a52cc05948ca9e3
objects//df/d1d047143ec437b431560e2ba8e7eb0c0d514f
objects//a2/fdcfaa06f1c56dac80df0cef908423b425ee4a
objects//bc/dc409a2e646854230114127ab8ad2f336668db
objects//f2/844b2a9c31f83e589a73c1c1b8bd7696c8cf64
objects//e3/cd28cf87c8f6c40529b4687cda50776970c829
objects//cf/fcf05fdee460b85057ae3716ce1f10ac205a22
objects//c1/b4b79b8bdd8e4496ea42bedcf143d22aea7e03
objects//4b/89dffc186f530de628673b199676a097d6d13c
objects//pack/pack-f37aaf1cb9275265234c59965db2dc9400655294.idx
objects//pack/pack-f37aaf1cb9275265234c59965db2dc9400655294.pack
objects//7c/3f99d89aafd49b67e50baf15adae32a3390ca5
objects//45/fcb1dd311e5e45af759cb3627dca5f47f58f04
objects//10/d85a79160a778436092df1706244394ba8376e
objects//81/2245d1177f889dd3ea2e20da6ac104085d25a3
objects//00/96423976e05d13728aa27490d6dd2d2840cc84
objects//98/96f7a7000a7b9d2fdb12047a141524358286c3
objects//5e/373ecaa68320f4fb5cfec60a0d46dd68084693
objects//5b/0dbbf0e8590575ff3fe8e9df38d140ce2e2948
objects//01/69e39a61bde682786d77bcde127fde0ff35d10
objects//6c/96898a9a0a0d5c5f1a38311a919b9b950aa05c
objects//64/0b6939d6dcfb9256e4f070f292f75c32bc8f59
objects//b8/f98b0f8093179f9374bcadd9b6ebe9a5bf274f
objects//ef/45e7786b3fab07e387ed8dda483d536d6fde1f
objects//e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
objects//f7/9afa4312d6903db0c6ad33aaca79a93dadd0c3
objects//e8/2ebb4066f0a6ae070b2f68d13bfbad2f9a6835
objects//fa/eef5ba81b606bf348f1a4e72683b0f8d71b5e4
objects//c5/d4dbaf3808a86af355490f4251ee020d8b7057
objects//c2/8b4ad71b16469c755dba11a1021703a3703212
objects//f8/c5f11ce099748ff751d5535fec38871895a153
objects//70/0ecda4eea6263b886e3ce5faa48ade2079f1d0
objects//23/754587e413b5e75b96665b21b7e9a6404c4026
objects//4f/a9114632f26ec590eec7e91712596085d7c442
objects//12/bbbadd5e636eb345c891e044d4231912085fe4
objects//76/175a40431fa6596e046795f46590208bb04232
objects//47/0d497ce7c645a4646ae7f95999846988546440
objects//78/b2aa7505031c06c109ff8bb2b0a8a1173c0652
objects//8e/2fe11b4ed5ac39b22f680a4c627fd88a6628fd
```

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

```
cd ..
ls
```

```
API.md                  docs
CONTRIBUTING.md         greeting.md
LICENSE.md              helper_functions.py
README.md               important_classes.py
about.md                setup.py
abstract_base_class.py  tests
alternative_classes.py
```

Skip to main content

9.7. Git HEAD

First lets get a a clean working directory

> **Note**
>
> MY first `git status` was not a clearn up to date working tree. I took the steps to get there

```
git status
```

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

nothing to commit, working tree clean
```

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.

```
cd .git
ls
```

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

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 HEAD
```

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

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

```
git status
```

```
fatal: this operation must be run in a work tree
```

Inside the `.git` directory `git` commands do not work.

```
cd ..
```

```
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 swithc and look agian.

```
git checkout 2-create-an-about-file
```

```
Switched to branch '2-create-an-about-file'
Your branch is up to date with 'origin/2-create-an-about-file'.
```

```
git status
```

```
On branch 2-create-an-about-file
Your branch is up to date with 'origin/2-create-an-about-file'.

nothing to commit, working tree clean
```

```
cat .git/HEAD
```

```
ref: refs/heads/2-create-an-about-file
```

As expected.

9.8. Branches are pointers

```
cat .git/refs/heads/2-create-an-about-file
```

```
57de0cd87ec14926273fd4760b94c0ea6f5b87d3
```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-in-class-brownsarahm-1
```

```
cd .git
```

```

```

```
/Users/brownsarahm/Documents/inclass/systems/github-in-class-brownsarahm-1/.git
```

```
git status
```

```
fatal: this operation must be run in a work tree
```

```
cd ..
```

```

```

```
pwd
```

```
/Users/brownsarahm/Documents/inclass/systems/github-in-class-brownsarahm-1
```

```
cat .git/refs/heads/2-create-an-about-file
```

```
57de0cd87ec14926273fd4760b94c0ea6f5b87d3
```

```
git checkout main
```

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

```
cat .git/HEAD
```

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

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

```
4fa9114632f26ec590eec7e91712596085d7c442
```

```
git status
```

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

nothing to commit, working tree clean
```

Skip to main content

```
Switched to branch 'organization'
Your branch is behind 'origin/organization' by 3 commits, and can be fast-forwarded.
  (use "git pull" to update your local branch)
```

```
cat .git/HEAD
```

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

```
git checkout main
```

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

```
git log
```

```
commit 4fa9114632f26ec590eec7e91712596085d7c442 (HEAD -> main, origin/main, origin/HEAD)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 14 12:48:11 2023 -0500

    bug fix

commit 6a2e1cc65204d0c05acc477e7d5e704f989a68dc
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:26:28 2023 -0500

    add jacket

commit 8e2fe11b4ed5ac39b22f680a4c627fd88a6628fd
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:22:43 2023 -0500

    Update about.md

commit bcdc409a2e646854230114127ab8ad2f336668db
Merge: cffcf05 56f29ca
Author: Sarah Brown <brownsarahm@uri.edu>
```

```
git log
```

```
commit 4fa9114632f26ec590eec7e91712596085d7c442 (HEAD -> main, origin/main, origin/HEAD)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 14 12:48:11 2023 -0500

    bug fix

commit 6a2e1cc65204d0c05acc477e7d5e704f989a68dc
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:26:28 2023 -0500

    add jacket

commit 8e2fe11b4ed5ac39b22f680a4c627fd88a6628fd
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:22:43 2023 -0500
```

```
commit bcdc409a2e646854230114127ab8ad2f336668db
Merge: cffcf05 56f29ca
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:20:59 2023 -0500

    Merge pull request #6 from introcompsys/organization

    Organization

commit 56f29cad4c33dc9e5c194e6f774024bca756b6ee (origin/organization)
Merge: 812245d cffcf05
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:18:04 2023 -0500

    Merge branch 'main' into organization

commit cffcf05fdee460b85057ae3716ce1f10ac205a22
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:09:30 2023 -0500

    add major

commit f2844b2a9c31f83e589a73c1c1b8bd7696c8cf64
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:04:12 2023 -0500

    update title

commit 812245d1177f889dd3ea2e20da6ac104085d25a3 (organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 12:59:00 2023 -0500

    start organizing
```

```
ls .git
```

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

```
cat .git/config
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-in-class-brownsarahm-1.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "2-create-an-about-file"]
        remote = origin
        merge = refs/heads/2-create-an-about-file
[branch "fill-in-about"]
        remote = origin
        merge = refs/heads/fill-in-about
[branch "organization"]
        remote = origin
```

9.9. Review today's class

1. Review the notes
2. Update your kwl chart with what you have learned or new questions
3. Practice with git log and redirects to write the commit history of your main branch for your kwl chart to a file gitlog.txt and commit that file to your kwl repo.
4. 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

9.10. Prepare for Next Class

1. In a gitunderstanding.md list 3-5 items from the following categories (a) things you have had trouble with in git in the past and how they relate to your new understanding (b) things that your understanding has changed based on today's class © things about git you still have questions about
2. Follow up on your progress issue and plan for the course

9.11. More Practice

1. Review the notes
2. Update your kwl chart with what you have learned or new questions
3. Practice with git log and redirects to write the commit history of your main branch for your kwl chart to a file gitlog.txt and commit that file to your kwl repo.
4. 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
5. Contribute either a cheatsheet item, or additional resource/reference to your group repo.
6. Complete one peer review of a team mate's contribution

```
## 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
```

9.12. Experience Report Evidence

9.13. Questions After Today's Class

9.13.1. Is the head file shared across all branches and updated every time we checkout a new branch or does each branch have its own head file that stores its own path?

The head file is like a variable that `git` writes to every time it checks out a new branch.

9.13.2. Will checking the commit hash locally and in the remote version of the branch play a role in going back in history if you messed something up?

That is exactly what we will do next

9.13.3. Can look go directly to the git/refs/main file instead of also looking the /HEAD before?

Yes, these are both files that we looked at, we can look at them in any order.

9.13.4. Just a little bit more clarification on what commands Porcelain and Plumbing have.

We'll use porcelain commands next class, that should help make the definitions more clear.

9.13.5. Are the names dictator and lieutenant really used?

It's the name of the workflow, but probably if a team is using it they use other terms like team lead or something.

9.13.6. What is a snapshot?

A "snapshot" is all of the details of the project (repo) at a given time. The term is derived from photography, a picture (aka snapshot) captures a moment in time.

the git parable uses a story to explain git.

this GitHub blog post might also be helpful.

9.13.7. when I set rebase to true, should i set it back to false?

Depends if you think you will ever want to not default to true.

9.13.8. How did git come to such prominence?

It is actually, truly, SO MUCH BETTER than what we had before. It is complicated, but so is every other version control system. *versioning* is hard no matter how you do it.

In addition to what we discussed today git is extremely fast.

It also helps that git is open source. Notice that git uses an integration manager type workflow.

9.13.9. what is the difference between the HEAD and ORIG_HEAD in the main branch?

These files are not in the working directory or *on* any branch. These tell us what branch is currently checked out locally and what

```
cat .git/ORIG_HEAD
```

```
bb4e0d87b6dc7d6c48e5515f6a52cc05948ca9e3
```

Notice that this gives a commit directly, not another branch. When we do `git status` we compare this to the local main to find out if we are up to date or not.

9.13.10. Do we have to link the prepare issue for this class day to this PR after it is done?

You should link the prepare issue from February 16 to today's (Feburary 21) experience badge PR.

9.13.11. Is there a limit to how many branches that one repo can have?

No.

9.13.12. Will what we learned today help to correct mistakes we make while using Git locally?

Yes.

9.13.13. will local commit hashes always match with the git servers?

Unless there is some hacking going on, yes.

10. How does git *really* work?

Today we will dig into how git really works. This will be a deep dive and provide a lot of details about git. It will conceptually reinforce important concepts and practically give you some ideas about how you might fix things when things go wrong.

Next week, we will built on this more on the practical side, but these **concepts** are very important for making sense of the more practical aspects of fixing things in git.

This deep dive in git is to help you build a correct, flexbile understanding of git so that you can use it independently and efficiently. The plumbing commands do not need to be a part of your daily use of git, but they are the way that we can dig in and see what *actually* happens when git creates a commit.

```
cd systems/ github-in-class-brownsarahm-1/
```

Important

git stores important content in *files* that it uses like variables.

Skip to main content

```
cat .git/HEAD
```

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

holds the current branch and

```
cat .git/config
```

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-in-class-brownsarahm-1.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "2-create-an-about-file"]
        remote = origin
        merge = refs/heads/2-create-an-about-file
[branch "fill-in-about"]
        remote = origin
        merge = refs/heads/fill-in-about
[branch "organization"]
        remote = origin
```

stores information about the different branches and remotes.

There are many:

```
ls .git
```

```
COMMIT_EDITMSG  description     info            refs
HEAD            hooks           logs
config          index           objects
```

Important

`.gitignore` is a file in th working direcotry that contains alist of files and patterns to not track.

We can see it is a hidden file in the working directory with `ls -a`

```
cd ../tiny-book/
ls -a
```

```
.                   _config.yml         markdown.md
..                  _toc.yml            notebooks.ipynb
.git                intro.md            references.bib
```

and what it contains:

```
cat .gitignore
```

```
_build
```

10.1. Creating a repo from scratch

We will start in the top level course directory.

```
cd ..
ls
```

Mine looks like this:

```
example                       tiny-book
github-in-class-brownsarahm-1
```

Yours should also have your kwl repo, group repo, etc.

We can create an empty repo from scratch using `git init <path>`

Last time we used an existing directory like `git init .`

Today we will create a new directory:

```
git init test
```

we get this message again, see context from last week

```
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/
```

It creates a folder and gives us a warning about branch names. If you have a new install you will not see this, because new versions of git have this by default.

We change into the new directory

and then rename the branch

```
git branch -m main
```

To clarify 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)
```

Noticee are no commits, and no origin.

```
ls .git
```

```
HEAD            description     info            refs
config          hooks           objects
```

we've looked at most of these, but we have not been able to see the objects before. We will work with those now.

10.2. Searching the file system

We can use the bash command `find` to search the file *system* note that this does not search the **contents** of the files, just the names.

```
find .git/objects/
```

```
.git/objects/
.git/objects//pack
.git/objects//info
```

we have a few items in that directory and the directory itself.

We can limit by type, to only files with the `-type` option set to `f`

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

```
find: --type: unknown primary or operator
```

I made a typo, so I removed the extra `-`

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

```

```

And we have no results. We have no objects yet. Because this is an empty repo

10.3. Git Objects

There are 3 types:

- blob objects: the content of your files (data)
- tree objects: stores file names and groups files together (organization)
- Commit Objects: stores information about the sha values of the snapshots

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

10.3.1. Hashing objects

Let's create our first one. 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 database 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. Rememebr, **everything** is a file. When we use things like `echo` it writes to th stdout file.

```
echo "test content"
```

```
test content
```

which shows on our trminal. We can us a pipe to connect the stdout of on command to the stdin of the next.

```
echo "test content" | git hash-object -w --stdin
```

we get back the hash:

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

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`

and we can check if it wrote to the database

Imp

pipes
we're
uses,
Pipes
comm

Skip to main content

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

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we see a file that it was supposed to have!

10.3.2. Viewing git objects

We can try with `cat`

```
cat .git/objects/d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
```

```
xK??OR04f(I-.QH??+I?+?K?
```

This is binary output that we cannot understand. Fortunately, git provides a utility. We can use `cat-file` to use the object by referencing at least 4 characters that are unique from the full hash, not the file name. (`70460` will not work)

```
git cat-file -p d670
```

`cat-file` requires *an* option `-p` is for pretty print

```
test content
```

This is the content that we put in, as expected.

10.3.3. Hashing a file

let's create a file

```
echo "version 1" > test.txt
```

and store it in our database, by hashing it

```
git hash-object -w test.txt
```

```
83baae61804e65cc73a7201a7252750c76066a30
```

we can look at what we have.

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

Skip to main content

```
.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 typ of fils with `-t` and `git cat-file`

```
git cat-file -t 83ba
```

```
blob
```

```
git cat-file -t d670
```

```
blob
```

Notice, however, that we only have one file in the working directory.

```
ls
```

```
test.txt
```

> **Note**
>
> the workign directory and the git repo are not strictly the same thing, and can be different like this. Mostly they will stay in closer relationship that we currently have unless we use plumbling commands, but it is good to build a solid understanding of how the `.git` directory relates to your working directory.

So far, even though we have hashed the object, git still thinks the file is untracked, because it is not in the tree and there are no commits that point to that part of the tree.

```
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)
```

We can write a tree

```
git write-tree
```

Skip to main content

and look at the tree

```
git cat-file -p 4b82
```

```

```

but it is empty

```
git cat-file -t 4b82
```

```
tree
```

becuse `write-tree` works from the index.

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

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

it still made an object tough, because we told it to.

classDiagram class d67046{ test content (blob) } class 83baae{ Version 1 (blob) } class 4b825d{ (tree) }

10.4. Updating the Index

Now, we can add our file as it is to the index.

```
git update-index --add --cacheinfo 100644 \
 83baae61804e65cc73a7201a7252750c76066a30 test.txt
```

the `\` lets us wrap onto a second line.

- this the plumbing command `git update-index` updates (or in this case creates an index, 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

```
git status
```

Skip to main content

```
No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt
```

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
```

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

Skip to main content

```
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

classDiagram class d67046{ test content (blob) } class 83baae{ Version 1 (blob) } class 4b825d{ (tree) } class 0c1e73{ Version 1 Verson 2 (blob) }

```
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
```

```
ls
```

```
test.txt
```

```
cat test.txt
```

```
version 1
version 2
```

```
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
```

Now we can write a tre again and it will have content.

```
git write-tree
```

Lets examine the tree, first check the type

```
git cat-file -t d8329
```

```
tree
```

and now we can look at its contents

```
git cat-file -p d8329
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

Now this is the status of our repo:

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.

10.5. 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
```

```
188a75ef66b6a85be0ab68d8575ec27808881dfc
```

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 when I re-ran this while typing the notes I got a different hash (`d450567fec96cbd8dd514313db9bcb96ad7664b0`) even though I have the same name and e-mail because the time changed.

We can also look at its type

```
git cat-file -t 188a
```

```
commit
```

and we can look at the content

```
git cat-file -p 188a
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1677177139 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1677177139 -0500
```

Now we check the final status of our repo

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//18/8a75ef66b6a85be0ab68d8575ec27808881dfc
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Important

Check that you also hav 6 objects and 5 of them should match mine, the one you should not have is the `188a75e` one but you should have a different one.

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 188a75{ tree d8329f author name commiter time } d8329f -|> 83baae 188a75 --|> d8329f

10.5.1. There are many git objects

Remember how many objects thre weere in the github inclass repo

```
cd ../github-in-class-brownsarahm-1/
```

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

```
.git/objects//0d/d61172a7dad0ac78ed6ca1788130a607983973
.git/objects//66/63ed3256515f3b45209c380c76bc2c47225175
.git/objects//57/de0cd87ec14926273fd4760b94c0ea6f5b87d3
.git/objects//03/29e78f9f8cc84f80808b907b044295f1f82594
.git/objects//6a/2e1cc65204d0c05acc477e7d5e704f989a68dc
.git/objects//56/f29cad4c33dc9e5c194e6f774024bca756b6ee
.git/objects//34/6853b8cf891a2783619506221965ffd35ed99d
.git/objects//5f/3e4cc9381a15bf59b928af1381cb5f2518cedb
.git/objects//9d/b284a866b196a79fd633c439b1dce783740137
.git/objects//9c/b0323be29dab2db451e9975fa2e65b37190ff1
.git/objects//d9/328fbe3ec302b590077eee7ad599b7dc246685
.git/objects//bb/4e0d87b6dc7d6c48e5515f6a52cc05948ca9e3
.git/objects//df/d1d047143ec437b431560e2ba8e7eb0c0d514f
.git/objects//a2/fdcfaa06f1c56dac80df0cef908423b425ee4a
.git/objects//bc/dc409a2e646854230114127ab8ad2f336668db
.git/objects//f2/844b2a9c31f83e589a73c1c1b8bd7696c8cf64
.git/objects//e3/cd28cf87c8f6c40529b4687cda50776970c829
.git/objects//cf/fcf05fdee460b85057ae3716ce1f10ac205a22
.git/objects//c1/b4b79b8bdd8e4496ea42bedcf143d22aea7e03
.git/objects//4b/89dffc186f530de628673b199676a097d6d13c
.git/objects//pack/pack-f37aaf1cb9275265234c59965db2dc9400655294.idx
.git/objects//pack/pack-f37aaf1cb9275265234c59965db2dc9400655294.pack
.git/objects//7c/3f99d89aafd49b67e50baf15adae32a3390ca5
.git/objects//45/fcb1dd311e5e45af759cb3627dca5f47f58f04
.git/objects//10/d85a79160a778436092df1706244394ba8376e
.git/objects//81/2245d1177f889dd3ea2e20da6ac104085d25a3
```

Skip to main content

```
.git/objects//5e/373ecaa68320f4fb5cfec60a0d46dd68084693
.git/objects//5b/0dbbf0e8590575ff3fe8e9df38d140ce2e2948
.git/objects//01/69e39a61bde682786d77bcde127fde0ff35d10
.git/objects//6c/96898a9a0a0d5c5f1a38311a919b9b950aa05c
.git/objects//64/0b6939d6dcfb9256e4f070f292f75c32bc8f59
.git/objects//b8/f98b0f8093179f9374bcadd9b6ebe9a5bf274f
.git/objects//ef/45e7786b3fab07e387ed8dda483d536d6fde1f
.git/objects//e6/9de29bb2d1d6434b8b29ae775ad8c2e48c5391
.git/objects//f7/9afa4312d6903db0c6ad33aaca79a93dadd0c3
.git/objects//e8/2ebb4066f0a6ae070b2f68d13bfbad2f9a6835
.git/objects//fa/eef5ba81b606bf348f1a4e72683b0f8d71b5e4
.git/objects//c5/d4dbaf3808a86af355490f4251ee020d8b7057
.git/objects//c2/8b4ad71b16469c755dba11a1021703a3703212
.git/objects//f8/c5f11ce099748ff751d5535fec38871895a153
.git/objects//70/0ecda4eea6263b886e3ce5faa48ade2079f1d0
.git/objects//23/754587e413b5e75b96665b21b7e9a6404c4026
.git/objects//4f/a9114632f26ec590eec7e91712596085d7c442
.git/objects//12/bbbadd5e636eb345c891e044d4231912085fe4
.git/objects//76/175a40431fa6596e046795f46590208bb04232
.git/objects//47/0d497ce7c645a4646ae7f95999846988546440
.git/objects//78/b2aa7505031c06c109ff8bb2b0a8a1173c0652
.git/objects//8e/2fe11b4ed5ac39b22f680a4c627fd88a6628fd
```

Back to the test repo

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//18/8a75ef66b6a85be0ab68d8575ec27808881dfc
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

and we have no change in the status

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

This is because git status works off the HEAD file and we have not updated that or set our branch to point to our commit yet. We will start there on Tuesday.

10.6. Review today's class

Skip to main content

2. 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 commands. Each row should have one git plumbing command and at least one of the corresponding git porcelain command(s). Include three rows: git add, git commit, and git status.
3. 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.

```
[text to display](url/of/link)
```

10.7. Prepare for Next Class

1. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file in the course notes (will provide prompts and tips). 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 the week after spring break, it does not have to be in the Feb 28 Experience Badge.**
2. make sure that you have a `test` git repo that matches the notes. **this is very important and if you do not have it you will not be able to follow along in class on Feb 28**

```
# IDE Thoughts

## Actions Accomplished
<!-- list what things you do: run code/ edit code/ create new files/ etc; no need to comment on what the code


## Features Used
<!-- list features of it that you use, like a file explorer, debugger, etc -->
```

10.8. More Practice

1. Review the notes
2. Read about git internals to review what we did in class in greater detail. Make gitplumbingdetail.md. 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. (view the raw version of this issue page for the git internals link above for an example)

10.9. Experience Report Evidence

> **Important**
>
> You need to have a test repo that matches this for class on tuesday.

Generate your evidence with the following in your test repo

```
find .git/objects/ -type f > testobj.md
```

then append the contents of your commit object to that file.

Move the `testobj.md` to your kwl repo in the experiences folder.

10.10. Questions After Today's Class

10.10.1. Why is Git so fast? It does lots of reading and writing from files which is a (relatively) slow process but does this all very quickly.

It is all structured files that are designed to be fast. Most are small and they are all written by git, there is no case handling or user decisions to parse.

10.10.2. What scenarios would we use this over the way we have done them before?

You would not use this way very often. It is helpful to work through it to build up an understanding of what actually happens. It basically allows us to slow down and do things one step at a time.

10.10.3. How does the tree structure properly become formatted without specifying the branch that you want it to start at or be based on?

The tree is a part of the *snapshot* of the files, it looks only at what is in the index. That is why our first one was empty. The tree is independent from branches, we will see how branches relate to this next week.

10.10.4. When files that are not staged get staged, does the file thats already staged change?

When you add a file to the staging area, it writes it to the `.git` direcotry, but it does not tell you what the hash of it is. So, it is stored, but you have no way to know how to get it back directly.

Remember, in the case we had today, we staged the already hashed version 1 of the file when our working directory already had only version 2.

In our example, we had a version of test.txt that wasn't staged and version that was committed. If we staged the second version of test.txt what would hapen to the already staged version. Since the first version is committed, we could still get back to that version. Each one of these is an object in the .git directory so it comes down to a matter of finding it in there.

10.10.5. Are hashes just a number indentifier for an object? Is there a way to change them?

git primarily uses them as an identifier, but it also uses it to verify that the content is what it should be. There is no way to change

10.10.6. What is the difference between random and deterministic?

Random means that it comes from probability, for example rolling a die or flipping a coin. Deterministic dervies from determined, it means there is a fixed outcome.

If you run a random function two times with the same input, it can give different values.

If you run a deterministic two times (or as many as you want) it will always have the same value.

10.10.7. How does the connection work between our devices and GitHub?

git does not require that at all. SO far in the test repo it is all 100% local. git uses regular web protocols HTTPS and SSH to send content to servers.

10.10.8. How is this useful on the day to day?

Understanding th different types of objects in git means that you will understand what git does and know how to accomplish different goals.

For example, once on a collaborative project, I realized that a collaborator had deleted content that we actually needed and added content we needed to keep in the same commit. Then we had made several more commits that impacted both files. Because I understood how git actually works, I was able to use the staging area and git revert carefully in order to undo the delete part of the commit and keep the adding part of that commit.

10.10.9. How common is it to use git plumbing commands?

First note that this is still not manually editing the .git directory. We are still relying on git, just smaller bits at a time. It is a thing to do in a pinch, when things are stuck or you get in a tricky thing you need to undo.

10.10.10. When does hashing occur, after any change is made to a file? Or when a git command is ran?

Only when a git command is run.

10.10.11. What happens if two hashes are the same?

this is called a collision. We will talk about it next week. git is set up to prevent this.

10.10.12. Is a tree a snapshot of the repo at the time?

it is a snapshot of everything that is staged. it is the whole files, not only the changes. git status shows us the fils changes, but the tree also points to the corresponding version of the other files.

10.10.13. if hashes generated by git are deterministic, why is it impossible to convert a hash back into the commit's contents?

It is an assymetric algorithm. We will talk more about hashes next week to see more about what this means.

10.10.14. How to access remote branches located in/at origin?

You can use `git branch -r` to viw remote branches.

10.10.15. Is using the write tree command the same as creating a new branch?

No, we have not done anything with branches so far. `write-tree` is one of the things that happens when we do `git add`

11. How do git references work?

> **Important**
>
> This picks up directly from where we left off in: How does git really work?

We start in the `test` repo:

```
cd test
ls -a
```

```
.               ..              .git            test.txt
```

We have one file, `test.txt`, but we were working more inside the `.git` directory.

Remember: git objects

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
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

Recall the `.git` directory has many other files in it.

```
ls .git
```

```
HEAD            description     index           objects
config          hooks           info            refs
```

Skip to main content

compares the working directory to the current state of the active branch

- we cansee the working directory with: `ls`
- we can see the active branch in the `HEAD` file
- what is its status?

```
git status
```

```
On branch main

No commits yet

Changes to be committed:
  (use "git rm --cached <file>..." to unstage)
        new file:   test.txt

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

we see it is “on main” this is because we set the branch to main , but since we have not written there, we have to do it directly. Notice that when we use the porcelain command for commit, it does this automatically; the porcelain commands do many things.

In our case because we made the commit manually, we did not update the branch, so it says we have no commits…

Notice, we have no commits yet even though we had written a commit. This is because the main branch does not point to any commit.

We can verify by looking at the `HEAD` file

```
cat .git/HEAD
```

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

and then viewing that file

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

```
cat: .git/refs/heads/main: No such file or directory
```

which does not even exist!

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

```
```

Skip to main content

```
ls .git/refs
```

```
heads   tags
```

we can see the objects though:

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//18/8a75ef66b6a85be0ab68d8575ec27808881dfc
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

```
git cat-file -t 188a
```

```
commit
```

This is our repo currently, but the branches are not there!

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 188a75{ tree d8329f author name commiter time } d8329f --|> 83baae 188a75 --|> d8329f

11.2. Git References

```
echo 188a75ef66b6a85be0ab68d8575ec27808881dfc > .git/refs/heads/main
```

```

```

```
git status
```

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

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

We can see that indeed we have one object that is a commit

```
git cat-file -p 188a
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1677177139 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1677177139 -0500

first commit
```

```
git cat-file -p d8329
```

```
100644 blob 83baae61804e65cc73a7201a7252750c76066a30    test.txt
```

```
git cat-file -p 83baa
```

```
version 1
```

So we now have HEAD-> main and main -> our commit -> tree –> blob.

11.3. Making another commit

First lets find the hash for the version 2 of test.txt

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//18/8a75ef66b6a85be0ab68d8575ec27808881dfc
.git/objects//4b/825dc642cb6eb9a060e54bf8d69288fbee4904
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

We can display objects to see that it is what we want.

```
git cat-file -p 0c1e
```

```
version 1
version 2
```

Then we add it to the index

```
git update-index --add --cacheinfo 100644 \
 0c1e7391ca4e59584f8b773ecdbbb9467eba1547 test.txt
```

Let's also add a second file

```
echo "new file" > new.txt
```

Tip

Revie
is cor

Skip to main content

```
git hash-object -w new.txt
```

```
fa49b077972391ad58037050f2a75f74e3671e92
```

and stage it as well

```
git update-index --add --cacheinfo 100644 \
>  fa49b077972391ad58037050f2a75f74e3671e92 new.txt
```

Now we have two files stages:

```
git status
```

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   new.txt
        modified:   test.txt


```

and in our repo

```
ls
```

```
new.txt         test.txt
```

Now we will write a tree object from all of the staged things:

```
git write-tree
```

```
163b45f0a0925b0655da232ea8a4188ccec615f5
```

and make a second comit.

```
echo "second commit" | git commit-tree 163b -p 188a
```

This command:

- `echo` sends “second commit” to stdout
- the pipe `|` connects that std out to stdin of the next command
- `git commit-tree` needs a tree (`163b`) as its input
- the `-p` option specifies the parent which is in this case our previous commit (mines is `188a`)

```
90f8d145f3b264e99832b47a662ed5d50b687e7a
```

Skip to main content

```
git status
```

```
On branch main
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   new.txt
        modified:   test.txt
```

THen we can set our main branch to point to the new commit so that it will see what we expect

```
git update-ref refs/heads/main 90f8
```

and we can also use `git log` now

```
git log
```

```
commit 90f8d145f3b264e99832b47a662ed5d50b687e7a (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Mar 2 13:15:39 2023 -0500

    second commit

commit 188a75ef66b6a85be0ab68d8575ec27808881dfc
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 23 13:32:19 2023 -0500

    first commit
```

11.4. What does all this get me?

We can create a new branch at a previous point:

```
git update-ref refs/heads/test 188a
```

```
git log
```

```
commit 90f8d145f3b264e99832b47a662ed5d50b687e7a (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Mar 2 13:15:39 2023 -0500

    second commit

commit 188a75ef66b6a85be0ab68d8575ec27808881dfc (test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 23 13:32:19 2023 -0500

    first commit
```

We see the file for each branch points to the commit

Skip to main content

```
188a75ef66b6a85be0ab68d8575ec27808881dfc
```

We can change them around as we like:

```
git update-ref refs/heads/main 188a
```

```
git log
```

```
commit 188a75ef66b6a85be0ab68d8575ec27808881dfc (HEAD -> main, test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 23 13:32:19 2023 -0500

    first commit
```

This does not lose any of our data, it just makes it harder to access, we have to know the hashes of thigns to find them if we did not have the branch structure to traverse the structure.

We can put this back

```
git update-ref refs/heads/main 90f8
git log
```

```
commit 90f8d145f3b264e99832b47a662ed5d50b687e7a (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Mar 2 13:15:39 2023 -0500

    second commit

commit 188a75ef66b6a85be0ab68d8575ec27808881dfc (test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 23 13:32:19 2023 -0500

    first commit
```

and see that the files are as expected:

```
ls
```

```
new.txt         test.txt
```

```
git status
```

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

```
git checkout 188a
```

```
Note: switching to '188a'.

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

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

  git switch -c <new-branch-name>

Or undo this operation with:

  git switch -

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

HEAD is now at 188a75e first commit
```

```
cat .git/HEAD
```

```
188a75ef66b6a85be0ab68d8575ec27808881dfc
```

This changes the head pointer directly to the commit.

```
ls
```

```
test.txt
```

and checkout also updates the working directory

```
cat test.txt
```

```
version 1
```

```
git checkout test
```

```
Switched to branch 'test'
```

```
ls
```

```
test.txt
```

```
cat test.txt
```

```
version 1
```

11.5. What does this mean?

We tend to think of comits like this:

flowchart RL subgraph A[first commit] %% commitA file1v1 %% treeA %% blob1 end subgraph B[second commit] file1v2 file2v1 file3v1 %% commitB %% treeB %% blob2 end B--> A
In reality

```
flowchart RL
subgraph A[first commit]
commitA
%% file1v1
treeA
blob1
commitA-->treeA
treeA-->blob1
end
subgraph B[second commit]
%% file1v2
%% file2v1
%% file3v1
commitB
treeB
blob1v2
blob2
blob3
commitB-->treeB
treeB-->blob1v2
treeB--->blob2
treeB-->blob3
end
%% B--> A
commitB -->commitA
```

11.6. We can move pointers around freely

```
flowchart BT
blob1
blob2
blob3
subgraph A[first commit]
    direction TB
    commitA
    %% file1v1
    treeA
    commitA-->treeA

end
subgraph B[second commit]
    direction TB
    %% file1v2
    %% file2v1
    %% file3v1
    commitB
    treeB
    commitB-->treeB
end
%% subgraph C[third commit]
```

```
%%     %% file2v1
%%     %% file3v1
%%     commitC
%%     commitC-->treeC
%% end
B--> A

%% treeC-->blob1
treeC--->blob2
treeC-->blob3
treeB-->blob1
treeB--->blob2
treeA-->blob1
commitB -->commitA
commitC-->commitB
treeA ~~~ treeB
treeB ~~~treeC
main --> commitA
branchB --> commitB
branchC --> commitC2
subgraph C2[new commit]
commitC2 -->treeC
commitC2 -->commitA

    treeC
end

%% A ~~~blob1
%% B ~~~blob2
%% C ~~~blob3
```

11.7. Review today's class

1. Read the notes and repeat the activity if needed
2. use `git cat-file` over the objects to draw a graph diagram of your current status in your test directory include your drawing in test_repo_map.md using mermaid syntax to diagram it. Name each node in your graph with 5-7 characters of the hash and the type. eg `0c913 commit`

11.8. Prepare for Next Class

1. Read about the Learn more about the SHA-1 collision attach
2. Think about different ways you know to represent numbers.

11.9. More Practice

1. Read the notes and repeat the activity if needed
2. use `git cat-file` over the objects to draw a graph diagram of your current status in your test directory include your drawing in test_repo_map.md using mermaid syntax to diagram it. Name each node in your graph with 5-7 characters of the hash and the type. eg `0c913 commit`
3. Add "version 3" to the test.txt file and hash that object
4. Add that to the staging area
5. Add the tree from the first commit to the staging area as a subdirectory with `git read-tree --prefix=back <hash>`
6. Write the new tree

Skip to main content

8. Update your diagram in test_repo_map.md after the following:

11.10. Experience Report Evidence

write your git status and object list to a file.

11.11. Questions After Today's Class

> **Important**
>
> I will get to these later

12. What is a commit number?

12.1. Admin

- spring break is like a time pause (you get an extra week on things assigned last week and this week)
- grading updates PR is made

12.2. 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 hasing function could be really simple, to read off a hash table, or it can be more complex.

For example:

Hash	content
0	Success
1	Failure

If we want to represent the status of a program running it has two possible outcomes: success or failure. We can use the following hash table and a function that takes in the content and returns the corresponding hash. Then we could pass around the 0 and 1 as a single bit of information that corresponds to the outcomes.

This lookup table hash works here.

In a more complex scenario, imagine trying to hash all of the new terms you learn in class. A table would be hard for this, because until you have seen them all, you do not know how many there will be. A more effective way to hash this, is to derive a *hashing*

A *cyrptographic* hash is additionally:

- unique
- not reversible
- similar inputs hash to very different values so they appear uncorrelated

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)

12.3. 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.

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
```

```
f3097ab13082b70f67202aab7dd9d1b35b7ceac2
```

Not

Some
passv
brows
proce
not re
(you
but es
warni

12.4. 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.

Git as originally designed to use SHA-1. SHA-1 is weak. Git switched to hardened HSA-1 in response to a collision. Learn more about the SHA-1 collision attach

> 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

```
echo "it's almost break" | git hash-object --stdin
```

```
d49aa364a349587fc438e7a738d58b8eb06b040f
```

Then we get the hash back. Let's change just one character

```
echo "it's almost brak" | git hash-object --stdin
```

```
671ece673e365c943997c861be56a48977ceff77
```

and we see that it changes a lot.

```
echo "it's almost braek" | git hash-object --stdin
```

```
185150671674540c2229dac8bda22ecba7bbc3f8
```

and again.

git uses the SHA hash primarily for uniuqeness, not privacy

It does provide some *security* assurances, because we can check the content against the hash to make sure it is what it matches.

This is a Secure Hashing Algorithm that is derived from cryptography. Because it is secure, no set of mathematical options can directly decrypt an SHA-1 hash. It is designed so that any possible content that we put in it returns a unique key. It uses a combination of bit level operations on the content to produce the unique values.

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

12.4.1. 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

```
cd ../github-in-class-brownsarahm-1/
git log
```

Skip to main content

Hin

GitHu
altern
so ye
devel
hosts
by the
that c

```
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 14 12:48:11 2023 -0500

    bug fix

commit 6a2e1cc65204d0c05acc477e7d5e704f989a68dc
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:26:28 2023 -0500

    add jacket

commit 8e2fe11b4ed5ac39b22f680a4c627fd88a6628fd
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 9 13:22:43 2023 -0500

    Update about.md
```

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
```

```
4fa9114 (HEAD -> main, origin/main, origin/HEAD) bug fix
6a2e1cc add jacket
8e2fe11 Update about.md
bcdc409 Merge pull request #6 from introcompsys/organization
56f29ca (origin/organization) Merge branch 'main' into organization
cffcf05 add major
f2844b2 update title
812245d (organization) start organizing
ef45e77 start organizing
4b89dff (my_branch_checkedoutb, my_branch) Merge pull request #5 from introcompsys/fill-in-about
c28b4ad (origin/fill-in-about, fill-in-about) add my name
0169e39 Merge pull request #4 from introcompsys/2-create-an-about-file
57de0cd (origin/2-create-an-about-file, 2-create-an-about-file) create empty about
3f54148 closes #1
4db10e5 Initial commit
```

12.5. What is a Number ?

a mathematical object used to count, measure and label

12.6. 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

In modern, western cultures our 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**.

Skip to main content

- called "Arabic" numerals in the West because Arab merchants introduced them to Europeans
- slow adoption

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.

12.6.1. Roman Numerals

is both different characters and not place based.

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

12.6.2. Decimal

To represent larger numbers than we have digits on we have a base (10) and then.

$$10 = 10 * 1 + 1 * 0$$

$$22 = 10 * 2 + 1 * 2$$

we have the ones ($10^0) place, tens$(10^1)$place, hundreds$(10^2) place etc.

12.6.3. 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

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

$$1001 => 8 * 1 + 4 * 0 + 2 * 0 + 1 * 1 = 9$$

12.6.4. 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.

use by native americans from present day CA

and

Pamean languages in Mexico

$$10 => 8 * 1 + 1 * 0 = 8$$

so 10 in octal is 8 in decimal

$$401 => 64 * 4 + 8 * 0 + 1 * 1 = 257$$

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.

As in binary we use hindu-arabic symbols, 0,1,2,3,4,5,6,7 (the first eight). Then nine is 11.

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.

12.6.5. 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 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.

```
cd ../test
```

Skip to main content

```
git log
```

```
commit 188a75ef66b6a85be0ab68d8575ec27808881dfc (HEAD -> test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 23 13:32:19 2023 -0500

    first commit
```

```
git status
```

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

```
cat .git/HEAD
```

```
ref: refs/heads/test
```

```
git checkout main
```

```
Switched to branch 'main'
```

```
cat .git/HEAD
```

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

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

```
90f8d145f3b264e99832b47a662ed5d50b687e7a
```

```

```

```
## Review today's class

```{include} ../_review/2023-03-07.md
```

12.7. Prepare for Next Class

1. Make sure you can run python code from bash and that you hve `gh` CLI installed. You will need to be able to run `gh` and python in the same terminal. This should happen for free on not-Windows or WSL. On Windows, check the GitBash settings.

Skip to main content
```

12.8. More Practice

1. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below) gittransition.md
2. 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`

```
## transition questions
1. Why make the switch?
3. What impact will the switch have on how git works?
4. Which developers will have the most work to do because of the switch?
```

12.9. Experience Report Evidence

12.10. Questions After Today's Class

13. Bash Scripting

So far we have used bash commands to navigate our file system as a way to learn about the file system itself. To do this we used commands like:

- `mv`
- `cd`
- `pwd`
- `ls`

Bash is a unix shell for the GNU operating system and it has been adopted in other contexts as well. It is the default shell on Ubuntu linux as well for example (and many others). This is why we teach it.

A Unix shell is both a command interpreter and a programming language. As a command interpreter, the shell provides the user interface to the rich set of GNU utilities. The programming language features allow these utilities to be combined.

Read the official definition of `bash` and a shell in the bash manual

Today we will start from the main course directory

```
cd systems
ls
```

```
example                         test
github-in-class-brownsarahm-1   testobj.md
kwl                             tiny-book
```

13.1. Variables in Bash

Skip to main content

```
NAME='Sarah'
```

> **Important**
>
> notice that there are **no spaces** around the `=`. spaces in bash separate commands and options, so they cannot be in variable declarations.

and use them with a `$` before the variable name.

```
echo $NAME
```

```
Sarah
```

This variable is local, in memory, to the current terminal window, so if we open a separate window and try `echo $NAME` it will not work. We can also see that it does not create any file changes.

```
ls
```

```
example                         test
github-in-class-brownsarahm-1   testobj.md
kwl                             tiny-book
```

We can, however use the variable at differnt working directories. So if we move

```
cd github-in-class-brownsarahm-1/
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 varialbe **value**

A common mistake is to put a space around the `=` sign, this is actually considered **good style** in many other languages.

```
:class: cell_input
NAME ='Sarah'
```

Skip to main content

```
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.

Removing the space works again:

```
NAME='Sarah'
```

```

```

13.2. Bash Loops

We can also make loops:

```
for name in Sarah Amoy Scott Bri
> 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 Scott Bri`)
- lists is bash are defined with no brackets and no commas `Sarah Amoy Scott Bri` is a list
- we start the loop body with `do` and close it with `done` these are like the `{` and `}` in some langauges.

```
Sarah
Amoy
Scott
Bri
```

When we get the command back with the up arrow key, it puts it all on one line, because it was one command. The `;` (semicolon) separates the “lines”

```
for name in Sarah Amoy Scott Bri Aiden; do echo $name; done
```

```
Sarah
Amoy
Scott
Bri
Aiden
```

```
ls
```

```
API.md                about.md              helper_functions.py
```

Skip to main content

README.md docs tests

13.3. Nesting commands

We can run a command to generate the list:

```
for file in $(ls)
> do
> echo $file
> done
API.md
CONTRIBUTING.md
LICENSE.md
README.md
about.md
abstract_base_class.py
alternative_classes.py
docs
helper_functions.py
important_classes.py
setup.py
tests
```

the `$()` tells bash to run that command first and then hold its output as a variable for use elsewhere

Important

I use this to make the date in the filename of your experience reports and the titles of your badge issues

Find & Comment

find where I use this in one of your action files and add a comment explaining the line. Put this commit in its own PR and request a review from @brownsarahm for a community badge!

We can modify what is in the `$()`:

```
for file in $(ls -a); do  echo $file; done
```

```
.
..
.git
.github
API.md
CONTRIBUTING.md
LICENSE.md
README.md
about.md
abstract_base_class.py
alternative_classes.py
docs
helper_functions.py
important_classes.py
setup.py
tests
```

Skip to main content

13.4. Conditionals in bash

We can also do conditional statements

```
if test -f docs
> then
> echo "file"
> fi
```

the key parts of this:

- `test` checks if a file or directory exists
- the `-f` option makes it check if the item is a *file*
- what to do if the condition is met goes after a `then` keyword
- the `fi` (backwards `if`) closes the if statment

This outputs nothing because `docs` is a directory not a file.

If we switch it, we get output:

```
if test -f API.md; then echo "file"; fi
```

```
file
```

We can put the if inside of the loop.

```
for file in $(ls); do if test -f $file;  echo $file; fi; done
```

```
bash: syntax error near unexpected token `fi'
```

I forgot the `then` so bash said it had bad syntax around `fi` this is because it was treating everything after the `if` as the condition and looking for a `then` but when it got to `fi` it knew it was too late and the `then` was not coming because the `fi` is supposed to be after `then`.

Once we put `then` in, it works:

```
for file in $(ls); do if test -f $file;then  echo $file; fi; done
```

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

Skip to main content

We can put our script into a file

```
nano filecheck.sh
```

So that the file

```
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
about.md
abstract_base_class.py
alternative_classes.py
filecheck.sh
helper_functions.py
important_classes.py
setup.py
```

```
cat filecheck.sh
```

```
for file in $(ls)
do
  if test -f $file
  then
     echo $file
  fi
done
```

13.6. `gh` CLI operations

When you are working sometimes it is helpful to be able to manipulate (or create) issues, pull requests or even releases from the command line.

```
gh issue list
```

```

Showing 1 of 1 open issue in introcompsys/github-in-class-brownsarahm-1

#3  Create a Add a classmate    about 1 month ago
```

We can also get options

```
gh issue list --help
```

```
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
```

We can filter them to only the closed ones

```
gh issue list -s closed
```

```
Showing 2 of 2 issues in introcompsys/github-in-class-brownsarahm-1 that match your search

#2  Create an about file    about 1 month ago
#1  Create a README         about 1 month ago
```

It is simlar for PRs:

```
gh pr list
```

Not

This i
work
in vs
termi
tag, a
the n
are p
all; th
than i

Skip to main content

When we use `create` is is interactive (or you can specify the options if you know them).

```
gh issue create
```

```
Creating issue in introcompsys/github-in-class-brownsarahm-1

? Title bug
? Body <Received>
? What's next? Submit
https://github.com/introcompsys/github-in-class-brownsarahm-1/issues/7
```

Now we can see our list of issues

```
gh issue list -s all
```

```
Showing 4 of 4 issues in introcompsys/github-in-class-brownsarahm-1 that match your search

#7  bug                         less than a minute ago
#3  Create a Add a classmate    about 1 month ago
#2  Create an about file        about 1 month ago
#1  Create a README             about 1 month ago
```

We can also search and filter them by piping the output to `grep` which searches the **contents** of a file (including stdin). We previously searched the file **names** with `find`. So `find` searches the paths that exist and `grep` actually reads the contents of the files, it does so faster than many other languages would be.

```
gh issue list -s all | grep "Create"
```

```
3       OPEN    Create a Add a classmate               2023-01-31 18:11:59 +0000 UTC
2       CLOSED  Create an about file            2023-02-02 18:14:19 +0000 UTC
1       CLOSED  Create a README         2023-01-31 18:18:20 +0000 UTC
```

`grep` can be used with pattern matching as well

> **Tip**
>
> Learning more about `grep` is a good explore badge topic

Here are the options

```
gh issue create --hlp
```

```
unknown flag: --hlp

Usage:  gh issue create [flags]

Flags:
```

> **Not**
>
> You c
> optio
> actio

Skip to main content

```
  -l, --label name       Add labels by name
  -m, --milestone name   Add the issue to a milestone by name
  -p, --project name     Add the issue to projects by name
      --recover string   Recover input from a failed run of create
  -t, --title string     Supply a title. Will prompt for one otherwise.
  -w, --web              Open the browser to create an issue
```

13.7. Badge Hints

Important

The goal is to put files in your kwl repo for us to review, that summarizes the *contributions* you have done in your group repo

- See the options for `gh pr list`.
- Use two strategies for what you are the author and when you are the reviewer
- Try the json option on `gh pr list` and see how it can help you
- I use bash in the your experience badge action to make a file with a date in the file name
- You can run a file from different locations

```
cd ../test
bash ../github-in-class-brownsarahm-1/filecheck.sh
```

```
new.txt
test.txt
```

```
cd ../../
bash systems/github-in-class-brownsarahm-1/filecheck.sh
```

```

```

```
ls
```

```
fa22            prog4dssp23     systems
```

13.8. Review today's class

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that can generate a file in your KWL repo with a list of all of your contributed PRs. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md

13.9. Prepare for Next Class

1. install putty on windows
2. spend 10 minutes after break before class, thinking about what you know about networking and how it works, what examples you have of it.

13.10. More Practice

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that can generate a file in your KWL repo with a list of all of your PRs and PR reviews. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md

13.11. Experience Report Evidence

13.12. Questions After Today's Class

13.12.1. Is there any structure/tools to reverse the order of if and do in a bash loop? Writing "do if" doesn't feel natural to me.

The `do` is a part of the `for`. We put the `if` in the loop body, as the first thing in there. It could have been like

```
for file in $(ls)
do
  echo checking $file
  if test -f $file
  then
     echo $file is a file
  fi
done
```

13.12.2. Would the script that we wrote into the file still work with semicolons or would adding them ruin the syntax?

This is easily testable.

13.12.3. How come we have been using git commands in stead of gh commands?

`git` commands are the core things we have been learning about so that you can use `git` no matter what host you use. `gh` commands **only** work with GitHub. They will not work on BitBucket, Gitlab, or any other git host. GitHub is popular, but since Microsoft purcahsed it, some people have left because they do not trust Microsoft.

13.12.4. What else can we use this gh command for?

Its most important uses are for issues and pull requests

It is the syntax for using a value from a variable.

13.12.6. Are the API.MD, CONTRIBUTING.MD …etc files important? They have disappeared from my main branch

No, but I do encourage you to try to get them back. Try using `git log` to see what might have happened.

13.12.7. Can the script be in any type of file or does it have to be in a certain type of file?

They must be in plain text but the file extension does not matter.

13.12.8. How is this you said fi is the oppisite if but in a contect in a if else statement what do it do? (like else if).

13.12.9. Is there any major benefit in creating an issue in a repo directly within a terminal, or is it just a matter of preference?

I use the terminal within vscode to make issues sometimes while I am working on onet thing but I get an idea for a future feature. Using the terminal is helpful because I don't always have the GitHub web page for thta repo open while I am working, but I do alwasy have a terminal in the folder where I'm working. So I can quickly take note of an idea and then get back to what I was doing. Less switching contexts helps stay focused and complete tasks faster.

13.12.10. How often would I use bash programming as a software engineer?

For small bits here and there. You might not write bash scripts often but when you do it will be thing sthat could save you a ton of time.
A lot of build processes are bash scripts, as we will see soon.

You will most likely never write a large program in bash, but you will almost definitely write or update small bash scripts on a semi-regular basis.

13.12.11. can you create classes and objects in bash?

No bash is not an object oriented language

13.12.12. what are the benefits, if any, to bash scripting in the terminal rather then in an IDE?

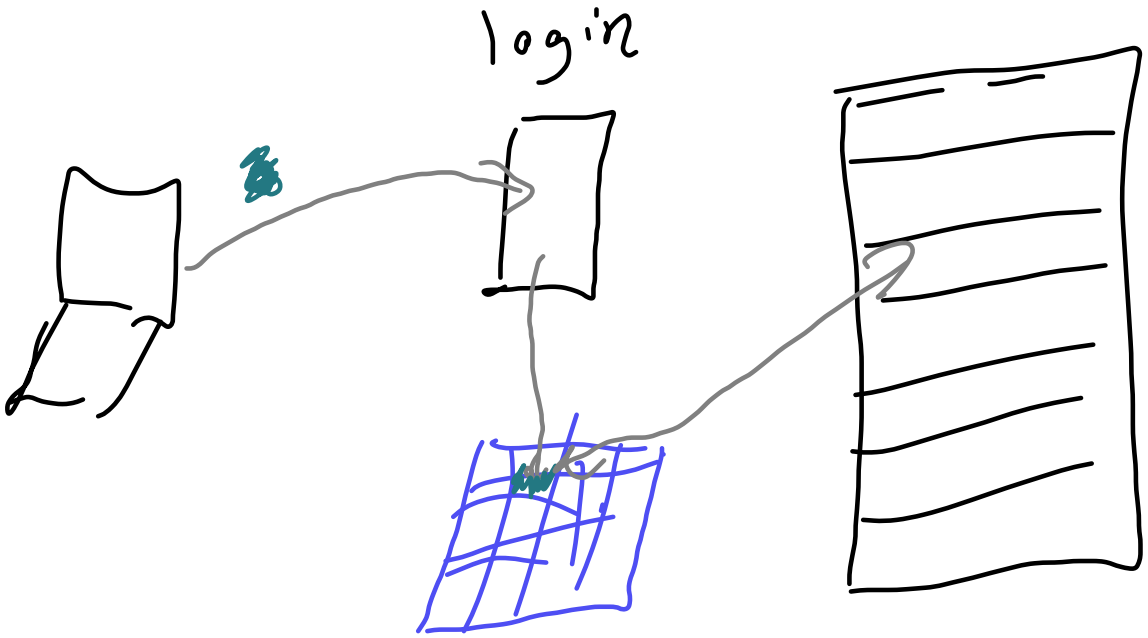
knowing how to do to it is important becuse sometimes an IDE is not available. On a regular basis though no difference.

13.12.13. What exactly does grep do, can it only be used with piping?

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 @ and I will tell you how to find your default password if you missed class (do not want to post it publicly). Comment on your experience report PR to ask for this information.

```
ssh -l 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.

```
The authenticity of host 'seawulf.uri.edu (131.128.217.210)' can't be established.
ECDSA key fingerprint is SHA256:RwhTUyjWLqwohXiRw+tYlTiJEbqX2n/drCpkIwQVCro.
Are you sure you want to continue connecting (yes/no/[fingerprint])? y
Please type 'yes', 'no' or the fingerprint: yes
Warning: Permanently added 'seawulf.uri.edu,131.128.217.210' (ECDSA) to the list of known hosts.
brownsarahm@seawulf.uri.edu's password:
You are required to change your password immediately (root enforced)
WARNING: Your password has expired.
You must change your password now and login again!
Changing password for user brownsarahm.
Changing password for brownsarahm.
(current) UNIX password:
New password:
```

Skip to main content

```
passwd: all authentication tokens updated successfully.
Connection to seawulf.uri.edu closed.
```

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

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
```

```
brownsarahm@seawulf.uri.edu's password:
Last login: Tue Mar  8 12:52:38 2022 from 172.20.133.152
```

We have logged into our home directory which is empty

```
[brownsarahm@seawulf ~]$ ls
```

```
[brownsarahm@seawulf ~]$ pwd
/home/brownsarahm
```

```
[brownsarahm@seawulf ~]$ whoami
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]
```

```
[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
```

```
[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
```

14.4. 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
```

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.

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.

Skip to main content

```
[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 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
2L      FlyBase start_codon     781297  781299  .       +       0       gene_id "FBgn0041250"; gene_symbol "G
2L      FlyBase CDS     781297  782048  .       +       0       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase CDS     782124  782181  .       +       1       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase CDS     782238  782441  .       +       0       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase CDS     782495  782821  .       +       0       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
2L      FlyBase stop_codon      782822  782824  .       +       0       gene_id "FBgn0041250"; gene_symbol "G
2L      FlyBase 3UTR    782825  782885  .       +       .       gene_id "FBgn0041250"; gene_symbol "Gr21a"; t
```

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 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 SRR307027_2.fastq
   20000 SRR307028_1.fastq
   20000 SRR307028_2.fastq
   20000 SRR307029_1.fastq
   20000 SRR307029_2.fastq
   20000 SRR307030_1.fastq
   20000 SRR307030_2.fastq
  320000 total
```

We can also search files, without loading them all into memory or displaying them, with `grep`:

```
[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
```

14.5. File permissions

Skip to main content

```
[brownsarahm@seawulf ~]$ echo "echo 'script works'" >> demo.sh
```

We can confirm that the script looks like a we expected

```
[brownsarahm@seawulf ~]$ cat demo.sh
echo 'script works'
```

One thing we could do is to run the script using `./`

```
[brownsarahm@seawulf ~]$ ./demo.sh
```

but we get a permission denied error

```
-bash: ./demo.sh: Permission denied
```

By default, files have different types of permissions: read, write, and execute for different users that can access them. To view the permissions, we can use the `-l` option of `ls`.

```
[brownsarahm@seawulf ~]$ ls -l
total 138452
-rw-r--r--. 1 brownsarahm spring2022-csc392 12534006 Apr 18  2021 bash-lesson.tar.gz
-rw-r--r--. 1 brownsarahm spring2022-csc392       20 Mar  8 13:12 demo.sh
-rw-r--r--. 1 brownsarahm spring2022-csc392 77426528 Jan 16  2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 brownsarahm spring2022-csc392   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
-rw-r--r--. 1 brownsarahm spring2022-csc392 25056938 Jan 25  2016 gene_association.fb
-rw-r--r--. 1 brownsarahm spring2022-csc392      447 Mar  8 13:07 linecounts.txt
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_2.fastq
```

For each file we get 10 characters in the first column that describe the permissions. The 3rd column is the username of the owner, the fourth is the group, then size date revised and the file name.

We are most interested in the 10 character permissions. The fist column indicates if any are directories with a `d` or a `-` for files. We have no directories, but we can create one to see this.

```
[brownsarahm@seawulf ~]$ mkdir results
```

```
[brownsarahm@seawulf ~]$ ls -l
total 138452
```

```
-rw-r--r--. 1 brownsarahm spring2022-csc392 77426528 Jan 16  2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 brownsarahm spring2022-csc392   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
-rw-r--r--. 1 brownsarahm spring2022-csc392 25056938 Jan 25  2016 gene_association.fb
-rw-r--r--. 1 brownsarahm spring2022-csc392      447 Mar  8 13:07 linecounts.txt
**drwxr-xr-x. 2 brownsarahm spring2022-csc392       10 Mar  8 13:16 results**
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_2.fastq
```

We can see in the bold line, that the first character is a d.

The next nine characters indicate permission to **R**ead, **W**rite, and e**X**ecute a file. With either the letter or a `-` for permissions not granted, they appear in three groups of three, three characters each for owner, group, anyone with access.

If we want to run the file, we *can* instead use `bash` directly, but this is limited relative to calling our script in other ways.

```
[brownsarahm@seawulf ~]$ bash demo.sh
script works
```

Instead, to add execute permission, we can use `chmod`

```
[brownsarahm@seawulf ~]$ chmod +x demo.sh
```

```
[brownsarahm@seawulf ~]$ ls -l
total 138452
-rw-r--r--. 1 brownsarahm spring2022-csc392 12534006 Apr 18  2021 bash-lesson.tar.gz
-rwxr-xr-x. 1 brownsarahm spring2022-csc392       20 Mar  8 13:12 demo.sh
-rw-r--r--. 1 brownsarahm spring2022-csc392 77426528 Jan 16  2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 brownsarahm spring2022-csc392   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
-rw-r--r--. 1 brownsarahm spring2022-csc392 25056938 Jan 25  2016 gene_association.fb
-rw-r--r--. 1 brownsarahm spring2022-csc392      447 Mar  8 13:07 linecounts.txt
drwxr-xr-x. 2 brownsarahm spring2022-csc392       10 Mar  8 13:16 results
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_2.fastq
```

Skip to main content

```
script works
```

We can add a bit more to our script to make it more interesting

```
[brownsarahm@seawulf ~]$ nano demo.sh
```

```
for VAR in *.gz
do
  echo $VAR
done


echo 'script works'
```

and note that that does not change the permission.

```
[brownsarahm@seawulf ~]$ ls -l
total 138452
-rw-r--r--. 1 brownsarahm spring2022-csc392 12534006 Apr 18  2021 bash-lesson.tar.gz
-rwxr-xr-x. 1 brownsarahm spring2022-csc392       60 Mar  8 13:27 demo.sh
-rw-r--r--. 1 brownsarahm spring2022-csc392 77426528 Jan 16  2018 dmel-all-r6.19.gtf
-rw-r--r--. 1 brownsarahm spring2022-csc392   721242 Jan 25  2016 dmel_unique_protein_isoforms_fb_2016_01.tsv
-rw-r--r--. 1 brownsarahm spring2022-csc392 25056938 Jan 25  2016 gene_association.fb
-rw-r--r--. 1 brownsarahm spring2022-csc392      447 Mar  8 13:07 linecounts.txt
drwxr-xr-x. 2 brownsarahm spring2022-csc392       10 Mar  8 13:16 results
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625262 Jan 25  2016 SRR307023_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625376 Jan 25  2016 SRR307024_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625286 Jan 25  2016 SRR307025_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625302 Jan 25  2016 SRR307026_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625312 Jan 25  2016 SRR307027_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625338 Jan 25  2016 SRR307028_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625390 Jan 25  2016 SRR307029_2.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_1.fastq
-rw-r--r--. 1 brownsarahm spring2022-csc392  1625318 Jan 25  2016 SRR307030_2.fastq
```

14.6. Review today's class

Important

This is an integrative 2x badge.

1. File permissions are represented numerically often in octal, by transforming the permissions for each level (owner, group, all) as binary into a number. Add octal-review.md to your KWL repo and answer the following.

   ```
   1. Transform the permissions [`r--`, `rw-`, `rwx`] to octal, by treating it as three bits.
   1. Transform the permission we changed our script to `rwxr-xr-x` to octal.
   1. Which of the following would prevent a file from being edited by other people 777 or 755?
   ```

2. 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. Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
- [x] correct answer marked with x
- [ ] another wong

---

- explanation for first wrong
- explanation for second wrong
- key point about correct
- explantion for third wrong

---

Next question
```

14.7. Prepare for Next Class

2. Review your notes on IDE use,make sure they are complete (from 2023-02-23 Prepare)
3. Preview the Stack Overflow Developer Survey Technology section parts that are about tools.

14.8. Practice

Important

This is an integrative 2x badge.

1. File permissions are represented numerically often in octal, by transforming the permissions for each level (owner, group, all) as binary into a number. Add octal-practice.md to your KWL repo and answer the following.

   ```
   1. Transform the permissions [`r--`, `rw-`, `rwx`] to octal, by treating it as three bits.
   1. Transform the permission we changed our script to `rwxr-xr-x` to octal.
   1. What permissions would we want (both long and in octal) would allow only the owner to run a file?
   1. Which of the following would prevent a file from being edited by other people 777 or 755?
   ```

2. 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. Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
```

```
---

- explanation for first wrong
- explanation for second wrong
- key point about correct
- explantion for third wrong

---

Next question
```

14.9. Experience Report Evidence

No specific files.

14.10. Questions After Today's Class

14.10.1. 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.

14.10.2. How would we change permissions for just one type of user with chmod instead of all users?

There are two ways. One is to se the permision you want it to end up at, for exame `chmod 744 demo.sh`. Another is to use more of the modifiers:

- "g" is for group
- "o" is for others
- "-" is for removing permissions
- "r" is for read-permission
- "w" is for write-permission
- "x" is for execute permission.
- "u" is for user / owner
- "+" is for adding permissions

So, instead of `chmod +x demo.sh` you could do `chmod u+x demo.sh`, or since we already did `chmod +x demo.sh` you could *remove* that access from the ones we do not want with `chmod go-x`.

you can see all of this and examples on the man page

14.10.3. 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.

Skip to main content

14.10.4. Is it normal to be able to change file permissions? In the case in class i think it worked because it was our own file, but we wouldnt be able to just change other peoples permissions right?

Correct you have to have write permission to be able to do this.

14.10.5. What type of data structure for storage does the server we used today follow? 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)

14.10.6. 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.

14.10.7. What are .fastq and .gz files?

.gz is a zip file, .fastq is a plain text output of a sequencing tool.

14.10.8. 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.

14.10.9. What is the point of a remote server?

More compute power.

14.10.10. Do the fiels 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.

14.10.11. 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.

14.10.12. Is access to the chmod command restricted on computers and servers?

on a file by file basis typically, eys.

Skip to main content

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.

14.10.14. Will I work with files more like this or locally as a software engineer?

Maybe not this type specifically, but you will almost surely encounter a large file at some point. You will also likely encounter a server for some reason.

14.10.15. 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.

14.10.16. How can I zip from tthe terminal?

`zip file file` see the man

14.10.17. 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.

15. What is an IDE?

15.1. Review your notes

> **Important**
>
> Do this before proceeding to the next section

Either discuss with peers in class or on the GitHub (asynch) discuss commonalities in your IDE notes.

15.1.1. In person

1. What different tasks did all of you use an IDE for?
2. What features of an IDE did you all use?
3. Which features were used but not very much?
4. Share the most helpful IDE feature you use?

15.1.2. On GitHub

There are questions on the GitHub Discussion. Update your individual IDE notes in your KWL repo with links to your post and replies.

15.2. Learn more

- What is an IDE?
- compare IDEs
- Most popular in 2021
- Most popular in 2022

15.2.1. In person

In class with your peers you can divide these up and read one and then share key points with others.

With your group, build a large list of IDE attributes or features that would be important, and make a table of how would you evaluate attribute? Which ones would you evaluate by just if it exists or not? Which ones would you evaluate in different degrees, what attributes of them would you evaluate?

Discuss with your group how you would rank them. You do not all have to agree on a final ranking, but notice the differences.

vs code is open source

Summarize what you all discussed on GitHub for your classmates. Note the ranking, with any disagreements.

15.2.2. Asynchronous

After reading the above, also read at least 3 different articles about the “best IDE” for your favorite language or for multiple languages.

Notice what IDE attributes or features the authors think is important, and how they evaluate each criterion. Which ones are evaluted as present/missing? Which ones are evaluated in more detail.

Join the discussion on GitHub summarizing what you found the most important criteria to be and if you personally agree or not.

15.3. Experience Reports

For today, whether you are in class or asynchronous use the experiecen report (makeup) action. Use ISO date format: YYYY-MM-DD for the workflow input.

15.4. Review today’s class

1. Explore the IDE you use most and add `frequentide.md` to your kwl with notes about which features it does/not have based on what you learned in the in-class activity.

Skip to main content

15.5. Prepare for Next Class

1. Read about connection protocols in general and specifically https and ssh. Wikipedia is a good source to start from, use sources to veirfy 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 Tuesdady and then submit others in your experience reflection.

15.6. More Practice

1. Explore the IDE you use most and add `frequentide.md` to your kwl with notes about which features it does/not have based on what you learned in the in-class activity.
2. Compare at least 3 IDEs for working in a single language. Your comparison should be based on first hand experience using each of the IDEs. Complete the same task in each tool. Create `favoriteide.md` to define and justify your preferred IDE. Include a ranked list of your criteria(which attributes and features) with justification/explanation of your ranking of these criteria. Then describe how each of the three IDEs meets/does not meet those criteria, and a conclusion of which IDE is the best based on your criteria.

15.7. Questions After Today's Class

Will be gathered from your experience reports.

16. Programming Languages

Today we'll explore how programming languages are cateogrized. Along the way, this will expose what core, generic features of a programming language are.

The key takeaway that I want for your is to have intution for how to choose a language for a project, not only your favorite, but what is the best for different types of projects.

> **Tip**
>
> You should do *either* the `in groups` or the `on your own` sections below for the x.x.x. For the x.x sections do all of them.

> **Important**
>
> Use the makeup workflow with an ISO formated date (YYYY-MM-DD)).

16.1. Comparing languages you know

Tak a few minutes on your own to fill in the follwing table for two languages of your own. Replace the `<langugage>` with two languages that you are familiar with. Add two additional rows. You can do this by memory, or by looking up/discussing. If you look things up, be sure to use reputable sources and include links to them.

Skip to main content

```
| ---------- | ---------- | ---------- |
| use of whitespace     | Text     | Text     |
| list/array types |    | |
| variable typing |  | |
| memory usage | | |
```

16.1.1. In groups

Share your table with some classmates and then discuss how they are similar and different.

Together, produce a list of questions for what other things you would want to know about how programming languages compare. All of you should include the collaboratively developed list in your experience report.

16.1.2. On your own

Post your table and a question about how you might compare programming languages on GitHub. Reply to at least two peers offering either advice or asking additional questions.

16.2. Learn more

- What is the study of programming languages? intro to PL

16.2.1. In groups

Discuss the reading, in particular:

- what you found most interesting
- was anything new to you?
- what do you want to remember most or learn more about?

In your experience report make a few notes (all can be the same notes// one person could write and share) about if you all agreed or each had different key points from the reading.

16.2.2. On your own

After reading this and your classmates' posts, add notes on broader patterns below your table in your experience report.

16.3. PL in Developer Survey

Read Carefully the developer survey// languages section for 2022. (the 2023 survey has not occurred yet)

Do additional reading about the languages from their official references if needed to answer the following questions.

16.3.1. With classmates

Discuss the findings to answer the following questions. This discussion should be at least 10-15 minutes including looking up

Skip to main content

- what is surprising?
- what did you expect?
- what do the popular languages have in common?
- what do the dreaded languages have in common?
- How are popular vs dreaded languages different? What features might be the cause for making a language dreaded?
- How to used languages differ from less commonly used languages? What features might be the cause for making a languge popular?
- How have things changed since 2011?

Include (shared) notes from the discussion in your experience report. Reflect (individually) on a few key points (2-3 bullet points) from the discussion in your experience report for today. Include the names of your group mates that you discussed with.

16.3.2. On your own

Include answers to the following questions in your experience report for today.

- what is surprising?
- what did you expect?
- what do the popular languages have in common?
- what do the dreaded languages have in common?
- How are popular vs dreaded languages different? What features might be the cause for making a language dreaded?
- How to used languages differ from less commonly used languages? What features might be the cause for making a languge popular?
- How have things changed since 2011?

16.4. Review today's class

1. For 2 languages from the loved vs dreaded list (one top 5; one bottom 5) read 2-3 posts about why people love/hate that language and summarize the key points on each side (meaning pros and cons for both loved and dreaded languges). Include links to all of the posts you read in a section titled `## Sources` in your markdown file. Make a bulleted list with some notes about the author's background and any limitations that might put on the scope of their opinions. (for example, a data scientists's opinion on languages is very valuable for data science, but less for app development) Add this to your kwl repo in `language_love_dread.md`.

16.5. Prepare for Next Class

1. See prep from 2023-03-23, no additional prep.

16.6. More Practice

1. Describe a type of project where it would be worth it for you to learn a language you have never used before in newlanguage.md This should be based in what types of features for the language your project would require and/or what would contribute to the long term health of the project.

Skip to main content

language (eg a toy data analysis in R): R, Julia, Clojure, Stan, Go. Try to use official documentation only to figure out a toy task to do. Answer the following questions in languagelearning.md:

```
1. What is this language designed for?
1. What Programming paradigm(s) does it support?
1. What language would make it easy to learn this language? Why?
1. What language would make it hard for someone to learn this language? Why?
1. What is its most unique feature(s)?
1. Include your small code bit (with comments!)
1. How was it trying to figure out this langauge
```

16.7. Experience Report Evidence

Answers to questions

16.8. Questions After Today's Class

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 | _ | _ | _ |
```

```
| 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.

date	file	type	zone
2023-01-26	syllabus-faq.md	review	penalty-free
2023-01-26	brain.md	practice	penalty-free
2023-01-26	syllabus-faq.md	practice	penalty-free
2023-01-31	gitoffline.md	review	full-requirements
2023-01-31	gitoffline.md	practice	full-requirements
2023-02-02	terminalwork.md	practice	full-requirements
2023-02-14	software.md	review	full-requirements
2023-02-14	methods.md	review	full-requirements
2023-02-14	methods.md	practice	full-requirements
2023-02-14	software.md	practice	full-requirements
2023-02-21	favorite_git_workflow.md	review	full-requirements
2023-02-21	workflows.md	practice	full-requirements
2023-02-21	gitunderstanding.md	prepare	full-requirements
2023-02-23	gitislike.md	practice	full-requirements
2023-02-23	gitplumbingreview.md	review	full-requirements
2023-02-23	idethoughts.md	prepare	full-requirements
2023-03-02	test_repo_map.md	practice	full-requirements
2023-03-02	test_repo_map.md	practice	full-requirements
2023-03-02	test_repo_map.md	review	full-requirements
2023-03-09	groupcontributions.sh` and its output as {index}`group_contributions-YYYY-MM-DD.md	review	full-requirements
2023-03-09	groupcontributions.sh` and its output as {index}`group_contributions-YYYY-MM-DD.md	practice	full-requirements
2023-03-21	octal-practice.md	practice	full-requirements
2023-03-			full-

Skip to main content

date	file	type	zone
2023-03-28	newlanguage.md	practice	full-requirements

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?

Skip to main content

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.

2023-01-24

related notes

Activities:

1. Review the notes after I post them.
2. Fill in the first two columns of your KWL chart.
3. review git and github vocabulary (include link in your badge PR)

2023-01-26

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
3. fill in the first two columns of your KWL chart
4. complete the syllabus quiz. If you get less than 100%, submit an FAQ for the course website in your KWL repo in a file named syllabus-faq.md about something that confused you with your best guess at the correct answer. If you get 100%, make a note in your badge PR.

2023-01-31

Activities:

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.
4. Reorganize 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 (will be in notes) in a new file, `terminal.md` in your kwl repo. 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, make notes.

```
### Terminal File moving reflection
1. Did this get easier toward the end?
1. What if anything did you get stuck on
1. When do you think that using the terminal will be better than using your GUI file explorer?
```

2023-02-02

related notes

Activities:

1. Review the notes
2. Find your team's page on GitHub. It is named like `Spring2023-group-#` join the discussion that I started on your page.
3. Complete the classmate issue in your inclass repo from today. Find a partner from within your assigned team by posting on your team's page. Link to your commits on your badge issue.
4. Try using git using your favorite IDE **or** GitHub Desktop. You can either do the other tasks for this badge, work on a different badge, or add & commit some random files in your inclass repo. Answer the questions below in `gitcompare.md`.

Questions:

```
## Reflection

1. What tool's git integration did you use?
1. Compare and contrast using git on the terminal and through the tool you used.  When would each be better/v
1. Did using a more visual representation help you understand better?
1. Describe the staging area (what happens after git add) in your own words.
2. what step is the hardest for you to remember? what do you think might help you?
```

2023-02-07

related notes

Activities:

1. Read today's notes
2. Update your KWL chart with the new items and any learned items.
3. add `branches.md` to your KWL repo and describe how branches work, in your own words. Include one question you have

Skip to main content

2023-02-09

related notes

Activities:

1. create `gitadvice.md` and write tips for how often to commit and how to avoid merge conflicts. Include at least 3 tips.
2. 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
3. clone your group repo.
4. work offline and add your contribution and then open a PR
5. review a class mate's PR.

2023-02-14

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?
```

2023-02-16

related notes

Activities:

1. Make your kwl repo into a jupyter book. Review the notes carefully for what files are required to make `jupyter-book build` run. Ignore your build directory.
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.
3. Learn about the documentation ecosystem in a language that you know besides Python. In `docs.md` include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources.

2023-02-21

related notes

1. Review the notes
2. Update your kwl chart with what you have learned or new questions
3. Practice with git log and redirects to write the commit history of your main branch for your kwl chart to a file gitlog.txt and commit that file to your kwl repo.
4. 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

2023-02-23

related notes

Activities:

1. Review the notes
2. 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 commands. Each row should have one git plumbing command and at least one of the corresponding git porcelain command(s). Include three rows: git add, git commit, and git status.
3. 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.

```
[text to display](url/of/link)
```

2023-03-02

related notes

Activities:

1. Read the notes and repeat the activity if needed
2. use `git cat-file` over the objects to draw a graph diagram of your current status in your test directory include your drawing in test_repo_map.md using mermaid syntax to diagram it. Name each node in your graph with 5-7 characters of the hash and the type. eg `0c913 commit`

2023-03-07

related notes

Activities:

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.

2023-03-09

Activities:

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that can generate a file in your KWL repo with a list of all of your contributed PRs. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md

2023-03-21

related notes

Activities:

1. File permissions are represented numerically often in octal, by transforming the permissions for each level (owner, group, all) as binary into a number. Add octal-review.md to your KWL repo and answer the following.

```
1. Transform the permissions [`r--`, `rw-`, `rwx`] to octal, by treating it as three bits.
1. Transform the permission we changed our script to `rwxr-xr-x` to octal.
1. Which of the following would prevent a file from being edited by other people 777 or 755?
```

2. create a `vocab-quiz.md` file 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. Questions 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. Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
- [x] correct answer marked with x
- [ ] another wong

---

- explanation for first wrong
- explanation for second wrong
- key point about correct
- explantion for third wrong

---

Next question
```

2023-03-23

related notes

Activities:

1. Explore the IDE you use most and add `frequentide.md` to your kwl with notes about which features it does/not have based on what you learned in the in-class activity.

Skip to main content

2023-03-28

related notes

Activities:

1. For 2 languages from the loved vs dreaded list (one top 5; one bottom 5) read 2-3 posts about why people love/hate that language and summarize the key points on each side (meaning pros and cons for both loved and dreaded languges). Include links to all of the posts you read in a section titled `## Sources` in your markdown file. Make a bulleted list with some notes about the author's background and any limitations that might put on the scope of their opinions. (for example, a data scientists's opinion on languages is very valuable for data science, but less for app development) Add this to your kwl repo in `language_love_dread.md`.

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.

2023-01-24

related notes

Activities:

1. Read the syllabus section of the course website carefully and explore the whole course website
2. Bring questions about the course to class
3. 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?
4. Post an introduction to your classmates on our discussion forum

2023-01-26

related notes

Activities:

1. Find the glossary page for the course website. Preview 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. Use the discussions to ask for help

2023-01-31

Activities:

1. Make a list of questions you have about using the terminal
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, come to class on Thursday and read this after.

2023-02-02

related notes

Activities:

1. Examine a large project you have done or by finding an open source project on GitHub. Answer the reflection questions in `software.md` in your kwl repo. (will be in notes)
2. map out how you think about data moving through a small program and bring it with you to class (no need to submit)

```
## Software Reflection

1. link to public repo if applicable or title of your 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?
1. Try to figure out (remember) how the project works. What types of things, without running the code can you
```

2023-02-07

related notes

Activities:

1. Read through the grading section and all of your feedback as it arrives.
2. Bring git questions or scenarios you want to be able to solve to class on Thursday
3. Update your `.github/workflows/experiencereflection.yml` file as follows: replace the two lines `team-reviewers: |` and `instructors` with `reviewers: <ta-gh-name>` where ta-gh-name is whichever TA is in your group. You can see your group on the organiation teams page named like “Spring 2023 Group X”. Make a PR and ask that TA for a review.
4. Answer the following in a comment on your prepare issue. Tag @brownsarahm on the issue

```
# Plan for success

__Target Grade:__ (A,B,..)

## Plan to get there:
- 24 experience badges
- (other badges you plan)

<!-- If you plan any build badges,uncomment the line below and list some ideas, topics from the course websi
<!-- ## Builds -->

<!-- If you plan any explore badges, create a schedule and propose one topic for your first explore badge -->
```

Skip to main content

2023-02-09

related notes

Activities:

1. Bring questions about git to class on Wednesday.
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. Read sections 1.1,1.2, and 1.3 of the pro git book this is mostly review at this point, but we are going to go into more of how git works next, so you need to make sure these concepts are all sorted out. Comment on your prepare issue if reading helped clarify confusion, made you more confused, or gave you new understanding. Either explain what you learned or ask a question you have. Tag @brownsarahm on the issue.

2023-02-14

related notes

Activities:

1. 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.
2. If you like to read about things before trying them, skim the jupyterbook docs.
3. Think about and be prepared to reply to questions in class about your past experiences with documentation, both using it and writing it.

2023-02-16

related notes

Activities:

1. Try exploring your a repo manually and bring more questions
2. Make sure that you have submitted and gotten feedback on your plan for the course. (Feb 7 prepare for class)

2023-02-21

related notes

Activities:

1. In a gitunderstanding.md list 3-5 items from the following categories (a) things you have had trouble with in git in the past and how they relate to your new understanding (b) things that your understanding has changed based on today's class © things about git you still have questions about
2. Follow up on your progress issue and plan for the course

Skip to main content

2023-02-23

related notes

Activities:

1. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file in the course notes (will provide prompts and tips). 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 the week after spring break, it does not have to be in the Feb 28 Experience Badge.**
2. make sure that you have a `test` git repo that matches the notes. **this is very important and if you do not have it you will not be able to follow along in class on Feb 28**

```
# IDE Thoughts

## Actions Accomplished
<!-- list what things you do: run code/ edit code/ create new files/ etc; no need to comment on what the code


## Features Used
<!-- list features of it that you use, like a file explorer, debugger, etc -->
```

2023-03-02

related notes

Activities:

1. Read about the Learn more about the SHA-1 collision attach
2. Think about different ways you know to represent numbers.

2023-03-07

related notes

Activities:

1. Make sure you can run python code from bash and that you hve `gh` CLI installed. You will need to be able to run `gh` and python in the same terminal. This should happen for free on not-Windows or WSL. On Windows, check the GitBash settings.

2023-03-09

related notes

Activities:

1. install putty on windows
2. spend 10 minutes after break before class, thinking about what you know about networking and how it works, what examples

2023-03-21

related notes

Activities:

2. Review your notes on IDE use,make sure they are complete (from 2023-02-23 Prepare)
3. Preview the Stack Overflow Developer Survey Technology section parts that are about tools.

2023-03-23

related notes

Activities:

1. Read about connection protocols in general and specifically https and ssh. Wikipedia is a good source to start from, use sources to veirfy 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 Tuesdady and then submit others in your experience reflection.

2023-03-28

related notes

Activities:

1. See prep from 2023-03-23, no additional prep.

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

2023-01-24

related notes

Activities:

2. Fill in the first two columns of your KWL chart.
3. review git and github vocabulary (include link in your badge PR)
4. Read more about version control in general and add a “version control” row to your KWL chart with all 3 columns filled in.

2023-01-26

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. 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`. Use mermaid syntax, to draw your map. GitHub can render it for you including while you work using the preview button.
4. complete the syllabus quiz. If you get less than 100%, submit an FAQ for the course website in your KWL repo in a file named syllabus-faq.md about something that confused you with your best guess at the correct answer. If you get 100%, make a note in your badge PR.

2023-01-31

related notes

Activities:

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.
4. Reorganize 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 (will be in notes) in a new file, `terminal.md` in your kwl repo. Start with a file explorer open, but then try to close it and use only command line tools to explore and make your choices. If you get stuck, look up additional commands to do acomplish your goals.

```
### Terminal File moving reflection
1. Did this get easier toward the end?
1. Use the `history` to see which commands you used and how many times each, make a table below.
1. Did you have to look up how to do anything we had not done in class?
1. When do you think that using the terminal will be better than using your GUI file explorer?
1. What questions/challenges/ relfections do you have after this?
1. What kinds of things might you want to write a bash script for given what you know in bash so far? come up
```

2023-02-02

related notes

Skip to main content

1. Review the notes
2. Find your team's page on GitHub. It is named like `Spring2023-group-#` join the discussion that I started on your page.
3. Download the course website repo via terminal. Append the commands used to a terminalwork.md
4. 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 your gitcommit.md. Compare what happens based on what you can see on GitHub and what you can see with git status.
5. Complete the classmate issue in your inclass repo from today. Find a partner from within your assigned team by posting on your team's page. Link to your commits on your badge issue.
6. Try using git using your favorite IDE **and** GitHub Desktop. You can either do the other tasks for this badge, work on a different badge, or add & commit some random files in your inclass repo. Answer the questions below in `gitcompare3ways.md`.

Questions:

```
## Reflection

1. What IDE did you use?
1. Was the IDE or GitHub better for you? Why?
1. Compare and contrast using git on the terminal and through your IDE. When would each be better/worse?
1. Did using a more visual representation help you understand better?
1. Describe the staging area (what happens after git add) in your own words. Provide an analogy for it using
2. What programming concepts is the staging area similar to?
2. what step is the hardest for you to remember? what do you think might help you?
```

2023-02-07

related notes

Activities:

1. Read today's notes
2. Update your KWL chart with the new items and any learned items.
3. Learn about GitHub forks (you can also use other resources)
4. 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.

2023-02-09

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. 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. create an issue on your group repo for a resource you want to review. Make sure that your contribution does not overlap with one that another member is going to post.

4. work offline and add your contribution and then open a PR. Your reivew should help a classmate decide if that reference material will help them or not.
5. review a class mate's PR.

2023-02-14

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 and why.
3. 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 the effective? why or why not? When do they work
- what are you most interested in trying that might be different?
```

2023-02-16

related notes

Activities:

1. Make your kwl repo into a jupyter book. Review the notes carefully for what files are required to make `jupyter-book build` run. Ignore your build directory.
2. Add one of the following features to your kwl repo:
   - a glossary both the terms and linknig to their use
   - substitutions
   - figure
3. Learn about the documentation ecosystem in another language that you know. In `docs.md` include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources.

2023-02-21

related notes

Activities:

1. Review the notes
2. Update your kwl chart with what you have learned or new questions
3. Practice with git log and redirects to write the commit history of your main branch for your kwl chart to a file gitlog.txt and commit that file to your kwl repo.

Skip to main content

5. Contribute either a cheatsheet item, or additional resource/reference to your group repo.
6. Complete one peer review of a team mate's contribution

```
## 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
```

2023-02-23

related notes

Activities:

1. Review the notes
2. Read about git internals to review what we did in class in greater detail. Make gitplumbingdetail.md. 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. (view the raw version of this issue page for the git internals link above for an example)

2023-03-02

related notes

Activities:

1. Read the notes and repeat the activity if needed
2. use `git cat-file` over the objects to draw a graph diagram of your current status in your test directory include your drawing in test_repo_map.md using mermaid syntax to diagram it. Name each node in your graph with 5-7 characters of the hash and the type. eg `0c913 commit`
3. Add "version 3" to the test.txt file and hash that object
4. Add that to the staging area
5. Add the tree from the first commit to the staging area as a subdirectory with `git read-tree --prefix=back <hash>`
6. Write the new tree
7. Make a commit with message "Commit 3" point to that tree and have your second commit as its parent.
8. Update your diagram in test_repo_map.md after the following.

2023-03-07

Activities:

1. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below) gittransition.md
2. 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`

```
## transition questions
1. Why make the switch?
3. What impact will the switch have on how git works?
4. Which developers will have the most work to do because of the switch?
```

2023-03-09

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that can generate a file in your KWL repo with a list of all of your PRs and PR reviews. Save the script as groupcontributions.sh and its output as group_contributions-YYYY-MM-DD.md

2023-03-21

related notes

Activities:

1. File permissions are represented numerically often in octal, by transforming the permissions for each level (owner, group, all) as binary into a number. Add octal-practice.md to your KWL repo and answer the following.

   ```
   1. Transform the permissions [`r--`, `rw-`, `rwx`] to octal, by treating it as three bits.
   1. Transform the permission we changed our script to `rwxr-xr-x` to octal.
   1. What permissions would we want (both long and in octal) would allow only the owner to run a file?
   1. Which of the following would prevent a file from being edited by other people 777 or 755?
   ```

2. 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. Use the following syntax:

```
Question text

- [ ] a wrong answer
- [ ] another wrong
- [x] correct answer marked with x
- [ ] another wong

---
```

```
- key point about correct
- explantion for third wrong

---

Next question
```

2023-03-23

related notes

Activities:

1. Explore the IDE you use most and add `frequentide.md` to your kwl with notes about which features it does/not have based on what you learned in the in-class activity.
2. Compare at least 3 IDEs for working in a single language. Your comparison should be based on first hand experience using each of the IDEs. Complete the same task in each tool. Create `favoriteide.md` to define and justify your preferred IDE. Include a ranked list of your criteria(which attributes and features) with justification/explanation of your ranking of these criteria. Then describe how each of the three IDEs meets/does not meet those criteria, and a conclusion of which IDE is the best based on your criteria.

2023-03-28

related notes

Activities:

1. Describe a type of project where it would be worth it for you to learn a language you have never used before in newlanguage.md This should be based in what types of features for the language your project would require and/or what would contribute to the long term health of the project.
2. Try out/learn about one of the following languages that you have not used before, do something small that is typical of that language (eg a toy data analysis in R): R, Julia, Clojure, Stan, Go. Try to use official documentation only to figure out a toy task to do. Answer the following questions in languagelearning.md:

```
1. What is this language designed for?
1. What Programming paradigm(s) does it support?
1. What language would make it easy to learn this language? Why?
1. What language would make it hard for someone to learn this language? Why?
1. What is its most unique feature(s)?
1. Include your small code bit (with comments!)
1. How was it trying to figure out this langauge
```

KWL File Information

Explore Badges

Skip to main content

> **Warning**
>
> Explore Badges are not required, but an option for higher grades. The logistics of this could be streamlined or the instructions may become more detialed during the penalty free zone.

Explore Badges can take different forms so the sections below outline some options. This page is not a cumulative list of requirements or an exhaustive list of options.

> **Tip**
>
> You might get a lot of suggestions for improvement on your first one, but if you apply that advice to future ones, they will get approved faster.

How do I propose?

Create an issue on your kwl repo, label it explore, and “assign” @brownsarahm.

In your issue, describe the question you want to answer or topic to explore and the format you want to use.

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

Skip to main content

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 or
- Review a reference or resource for a topic

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

> **Warning**
>
> This page is subject to change until the end of the penalty free zone

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:

Skip to main content

- 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.

This summary report have the following sections.

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.

Build Ideas

- 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
-

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.

Additionally, when we log them in our private gradebook, we will give you a "receipt" that is 128 characters long. You will be able to

at particular points in the course, an in class or class preparation activity will be for you to review a “progress report” that we give you 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.

Explore and build, we’ll agree to a deadline when you propose.

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

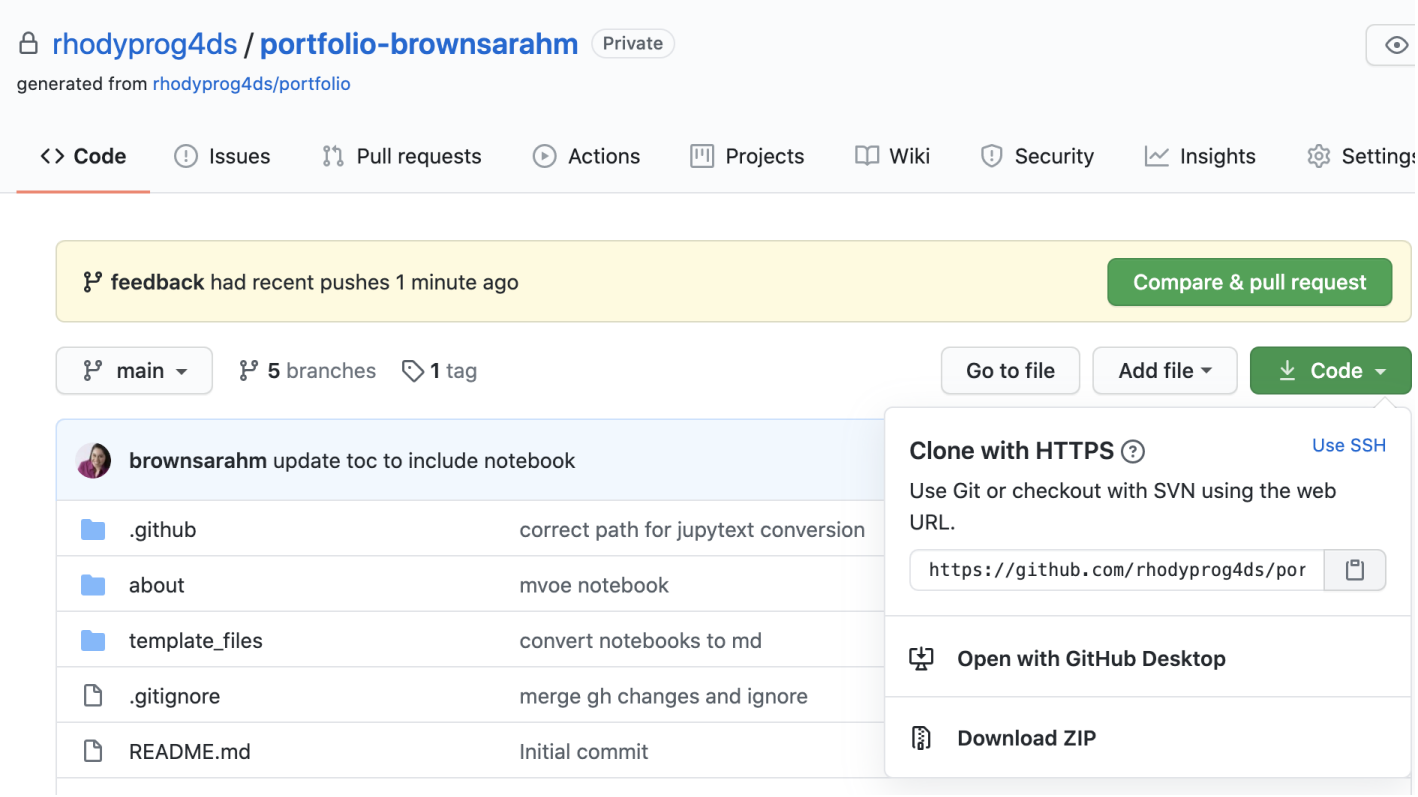
GitHub has strong rules about authentication You need to use SSH with a public/private key; HTTPS with a Personal Access Token or use the GitHub CLI auth

Help! I accidentally merged the Feedback Pull Request before my assignment was graded

That's ok. You can fix it.

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
```

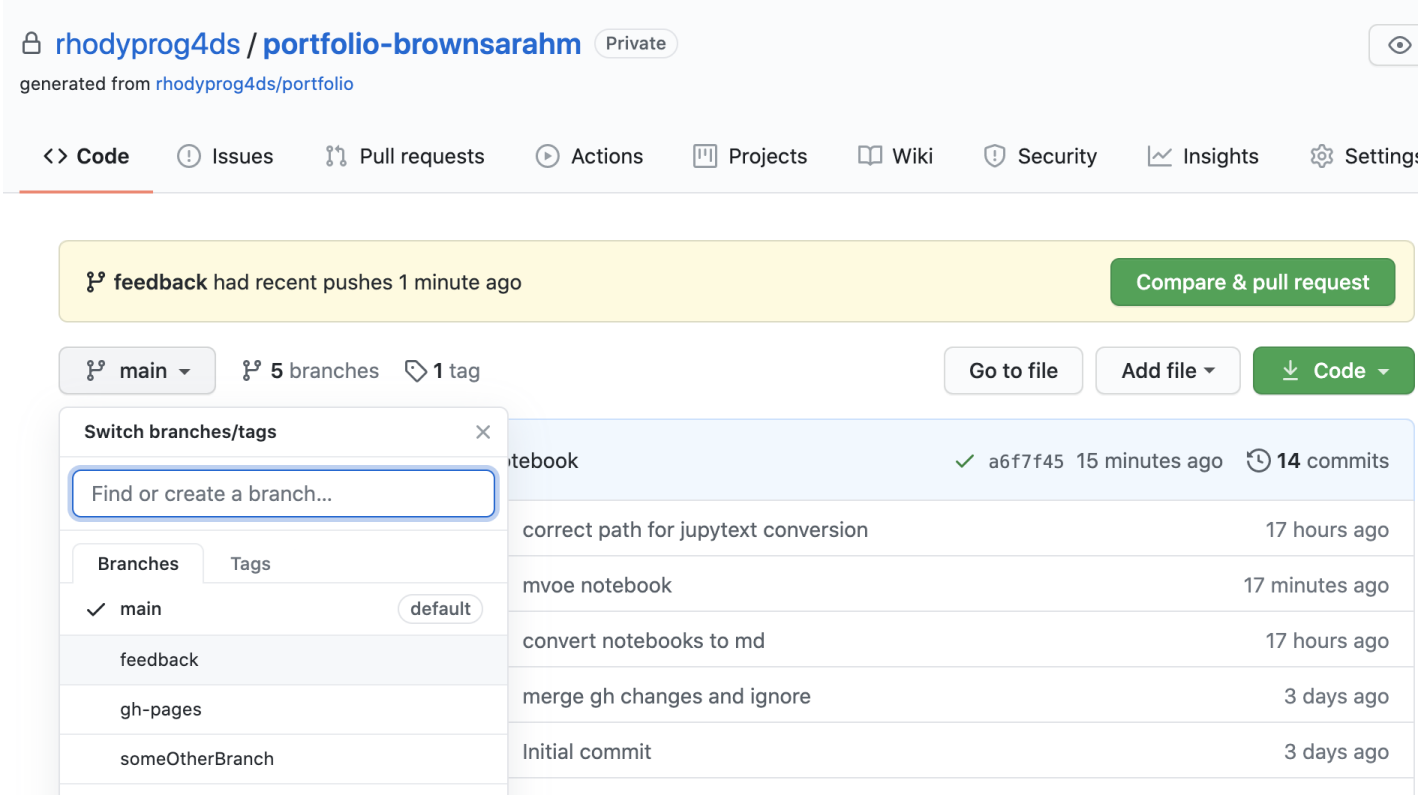
then past your url that you copied. It will look something like this, but the last part will be the current assignment repo and your username.

```
git clone https://github.com/rhodyprog4ds/portfolio-brownsarahm.git
```

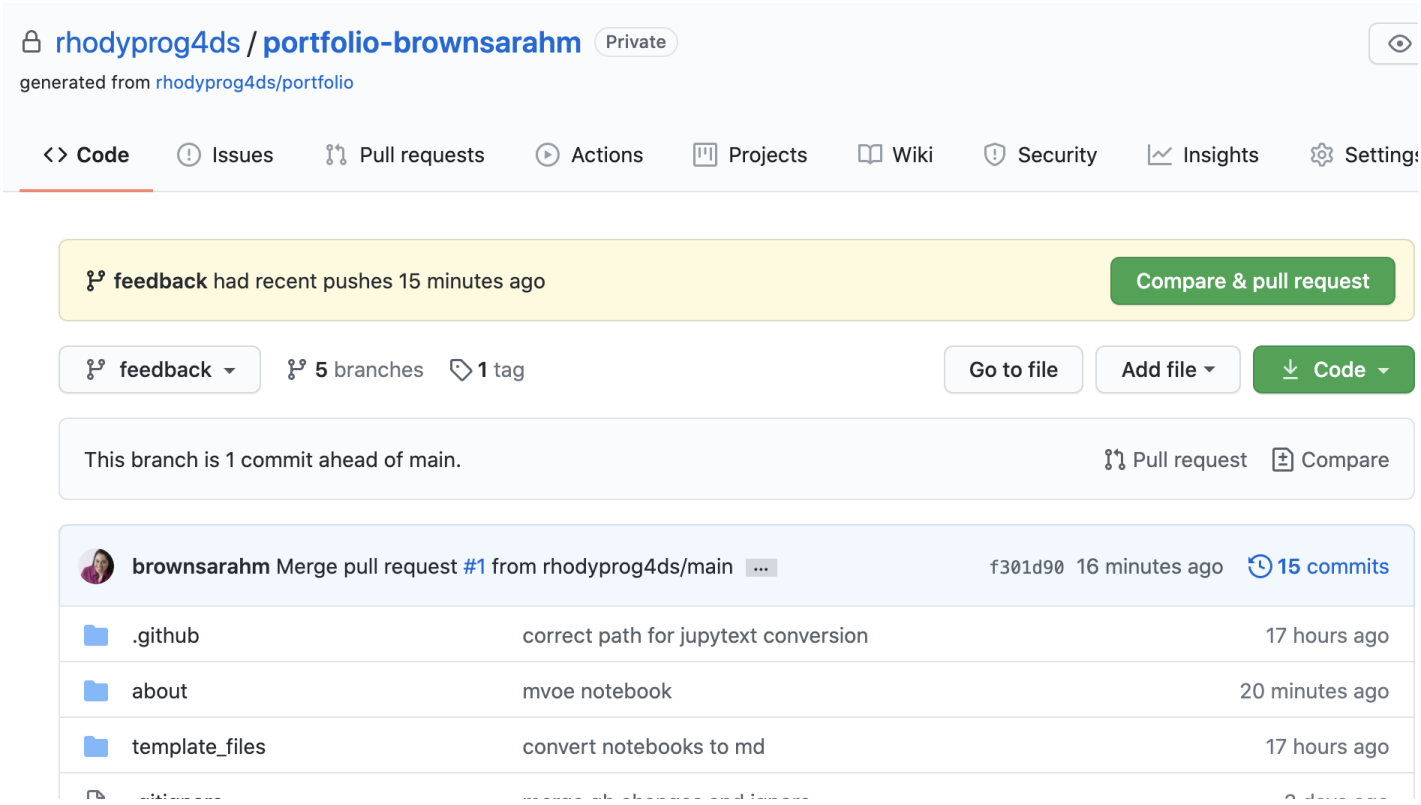
When you merged the Feedback pull request you advanced the `feedback` branch, so we need to hard reset it back to before you did any work. To do this, first check it out, by navigating into the folder for your repository (created when you cloned above) and then checking it out, and making sure it's up to date with the `remote` (the copy on GitHub)

```
cd portfolio-brownsarahm
git checkout feedback
git pull
```

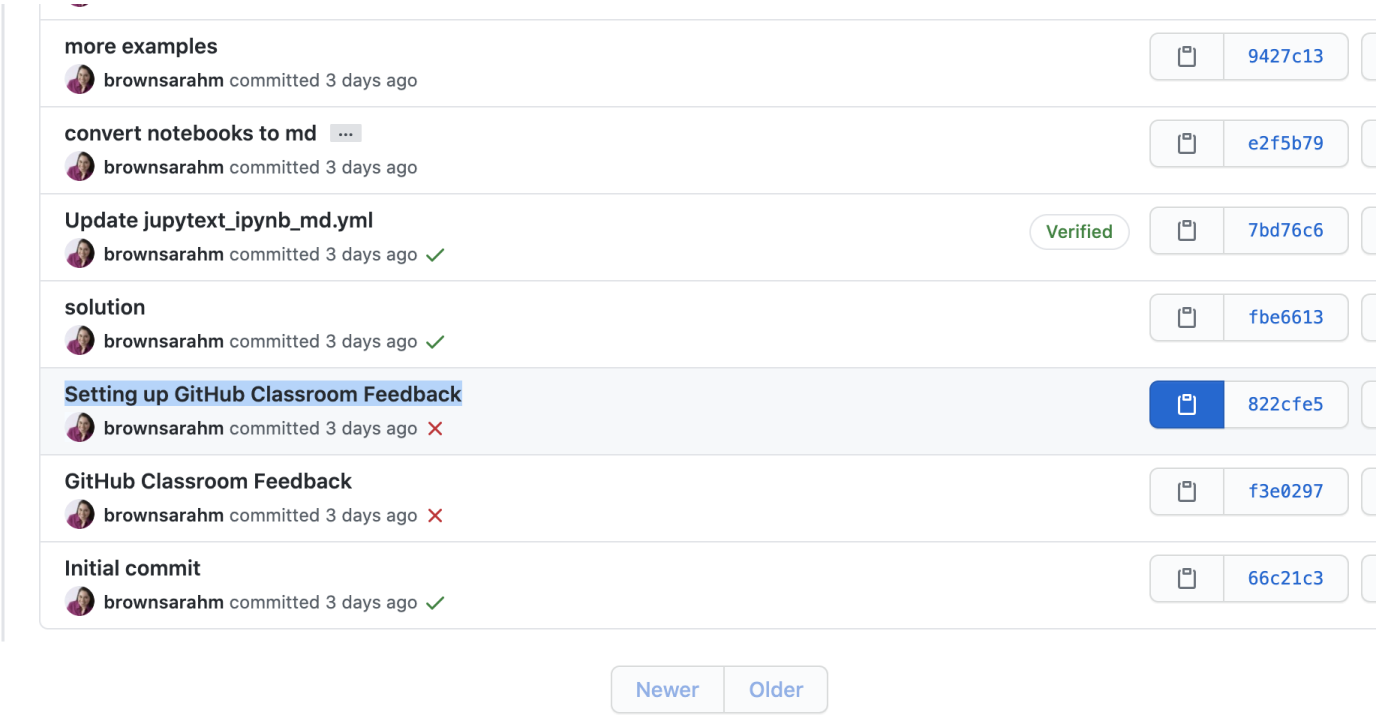
Now, you have to figure out what commit to revert to, so go back to GitHub in your browser, and swithc to the feedback branch there. Click on where it says `main` on the top right next to the branch icon and choose feedback from the list.



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.

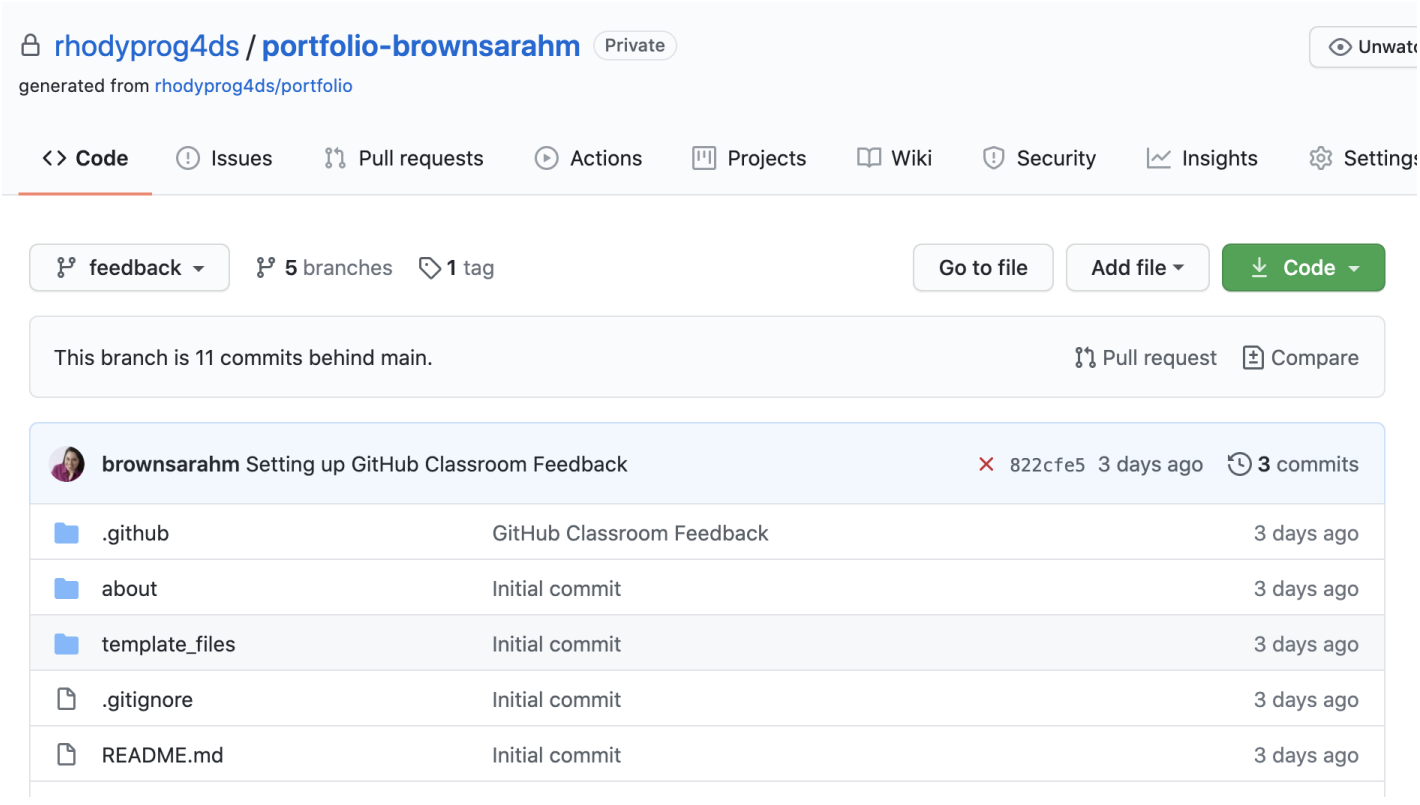
```
git push origin -f
```

and you'll get something like this to know that it worked

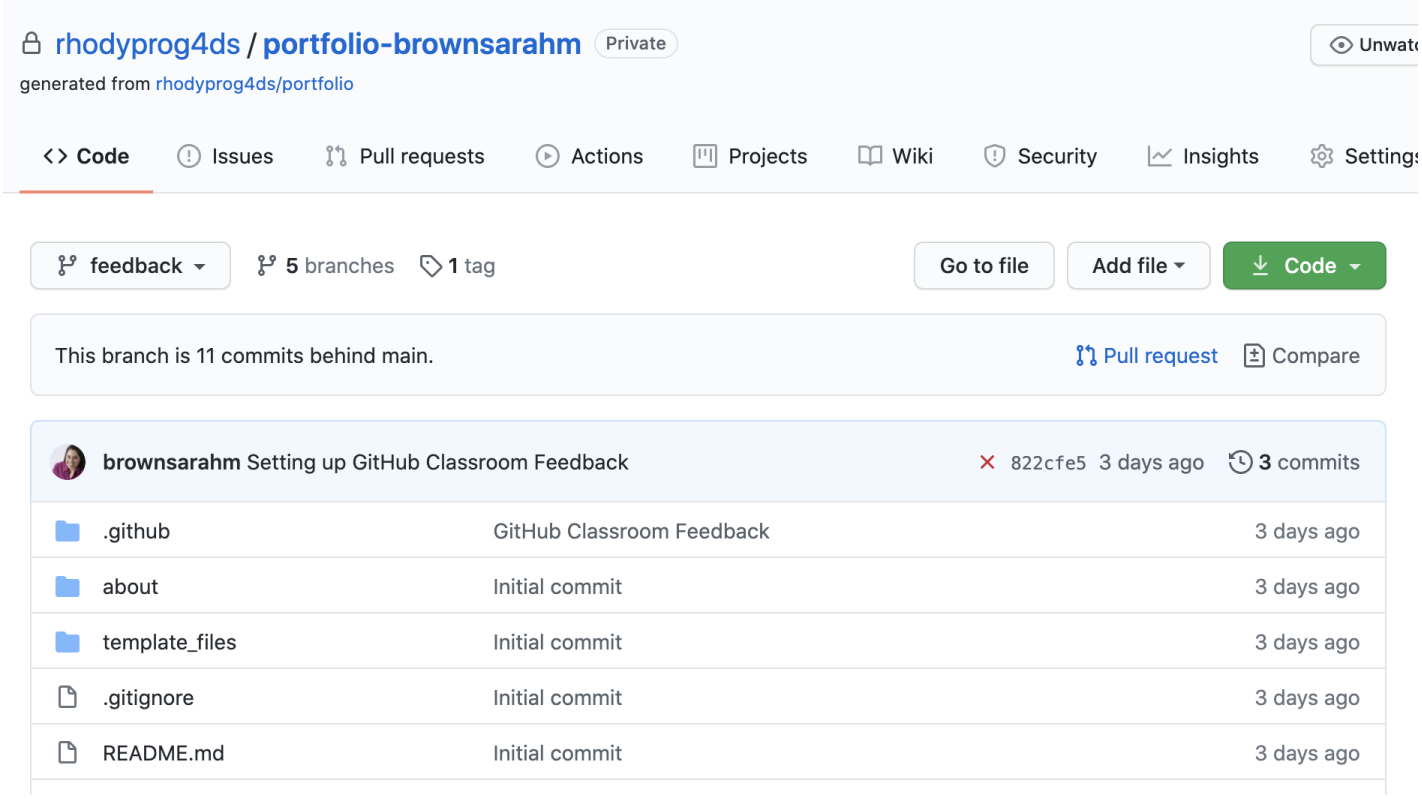
```
Total 0 (delta 0), reused 0 (delta 0)
To https://github.com/rhodyprog4ds/portfolio-brownsarahm.git
 + f301d90...822cfe5 feedback -> feedback (forced update)
```

Again, the numbers will be different and it will be your url, not mine.

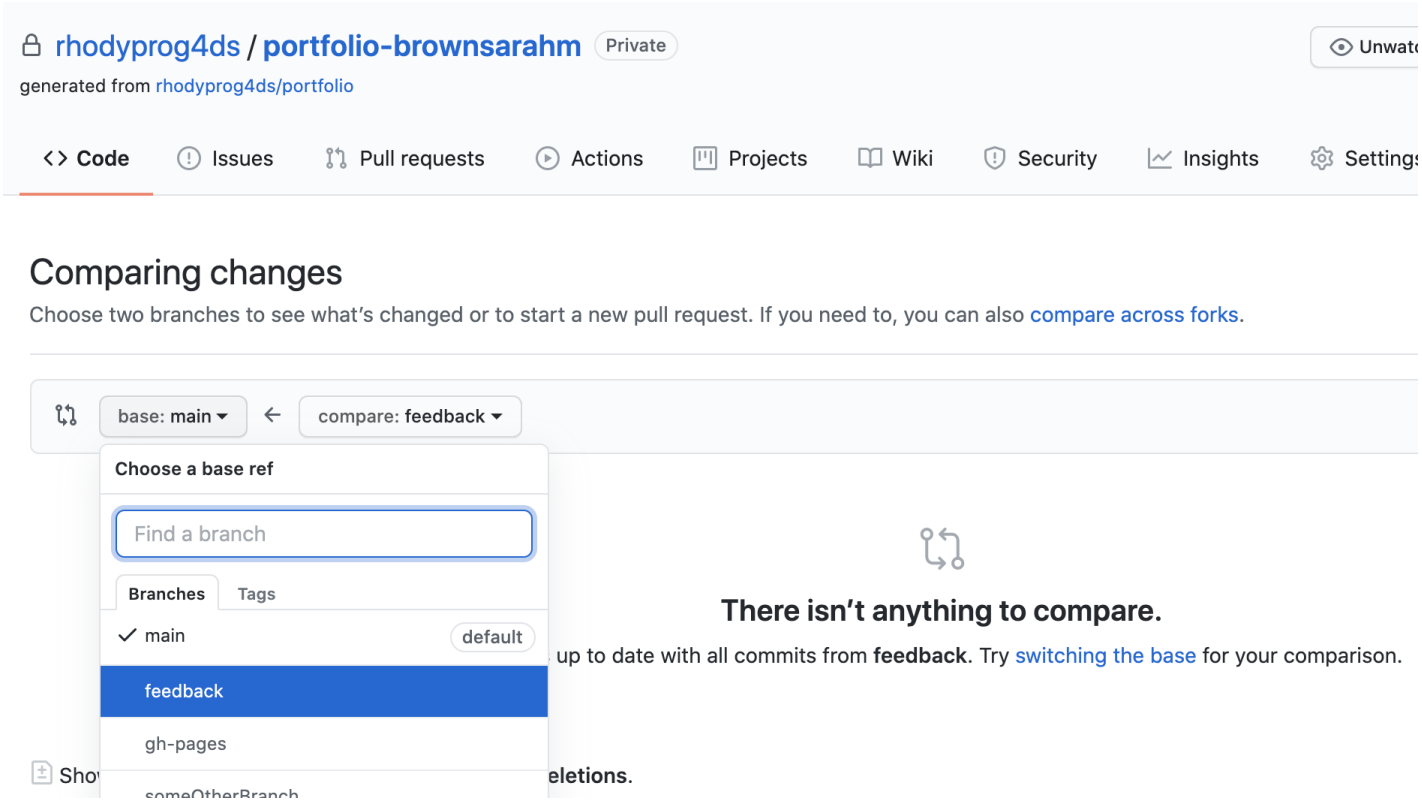
Now back on GitHub, in your browser, click on the code tab. It should look something like this now. Notice that it says, "This branch is 11 commits behind main" your number will be different but it should be 1 less than the number you had when you checked `git status`. This is because we reverted the 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").



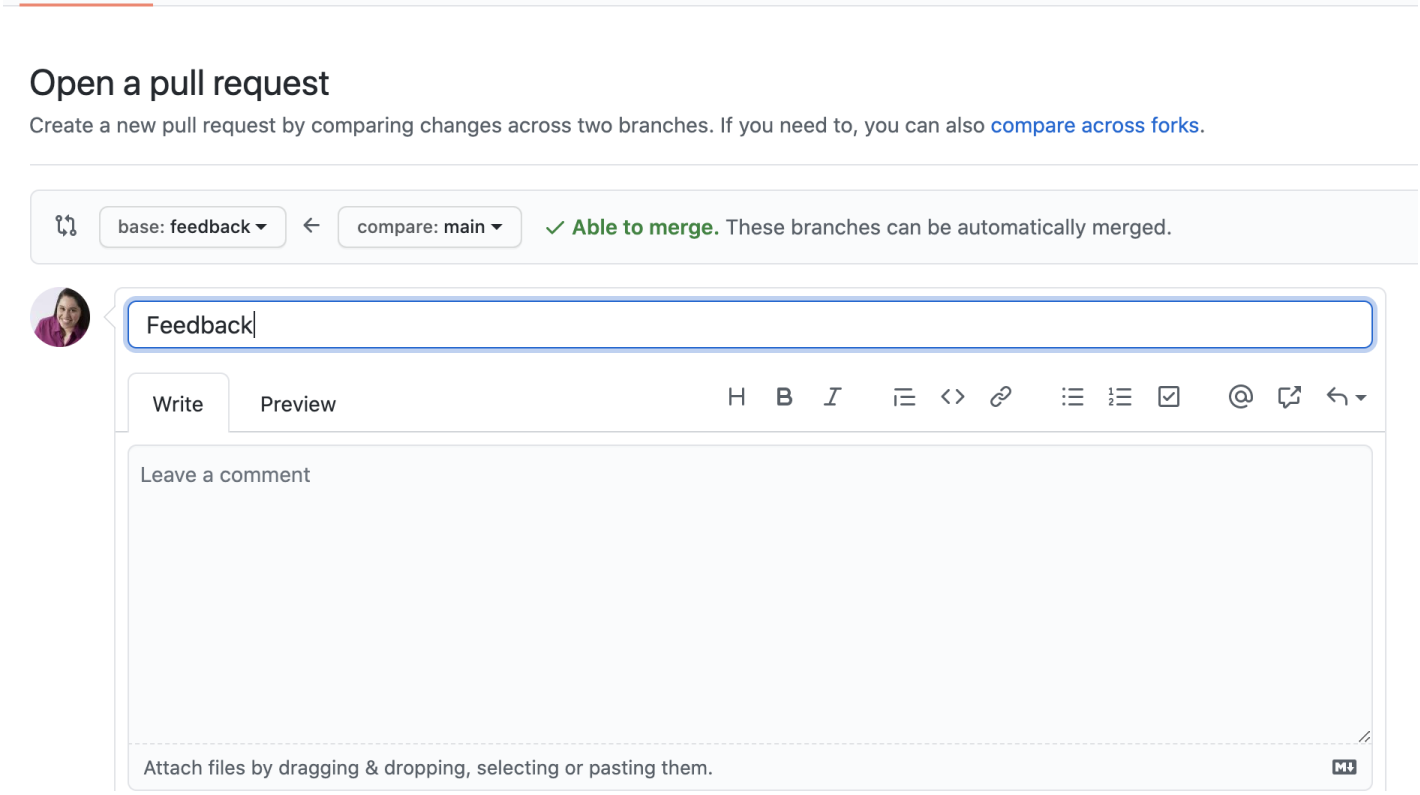
Now, you need to recreate your Pull Request, click where it says pull request.



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
```

Glossary

> **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.

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.

Skip to main content

.

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

Git Plumbing commands

low level git commands that allow the user to access the inner workings of git.

Git Workflow

a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner

HEAD

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.)

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.

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.

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

SHA 1

the hashing function that git uses to hash its functions (found to have very serious collisions (two different inputs have same hashes), so a lot of software is switching to SHA 256)

shell

a command line interface; allows for access to an operating system

ssh

allows computers to safely connect to networks (such as when we used an ssh key to clone our github repos)

templating

templating is the idea of changing the input or output of a system. For instance, the Jupyter book, instead of outputting the markdown files as markdown files, displays them as HTML pages (with the contents of the markdown file).

terminal

a program that makes shell visible for us and allows for interactions with it

tree objects

type of git object in git that helps store multiple files with their hashes (similar to directories in a file system)

YAML

: 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 you *read* code or debug environment issues. You also have to know how to effectively use these tools.
Knowing the common abstractions we use in computing and recognizing them when they look a little bit differently will help you with these more complex tasks. Understanding what is common when you move from one environment to another or to This course is designed to have you not only learn the material, but also to build skill in learning to program. Following these guidelines will help you build habits to not only be successful in this class, but also in future programming.

Why this way?

Learning requires iterative practice. In this class, you will first get ready to learn by preparing for class. Then, in class, you will get a first experience with the material. The goal is that each class is a chance to learn by engaging with the ideas, it is to be a guided inquiry. Some classes will have a bit more lecture and others will be all hands on with explanation, but the goal is that you *experience* the topics in a way that helps you remember, because being immersed in an activity helps brains remember more than passively watching something. Then you have to practice with the material

A new 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 modules, libraries, or functions introduced in that class.

If you find anything hard to understand or unclear, write it down to bring to class the next day or post an issue on the course website.

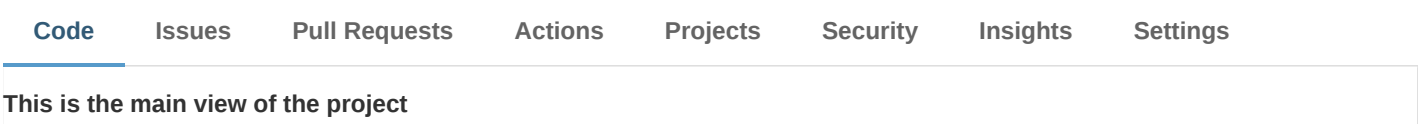
GitHub Interface reference

This is an overview of the parts of GitHubt from the view on a repository page. It has links to the relevant GitHub documentation for more detail.

Top of page

The very top menu with the logo in it has GitHub level menus that are not related to the current repository.

Repository specific page



This is the main view of the project

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.

the header in this area lists who made the last commit, the message of that commit, the short hash, date of that commit and the total number of commits to the project.

If there are actions on the repo, there will be a red x or a green check to indicate that if it failed or succeeded on that commit. ^^^ file list: a table where the first column is the name, the second column is the message of the last commit to change that file (or folder) and the third column is when is how long ago/when that commit was made

README file

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

Releases mark certain commits as important and give easy access to that version. They are related to git tags

Packages are out of scope for this course. GitHub helps you manage distributing your code to make it easier for users.

Environments are a tool for dependency management. We will cover thigns that help you know how to use this feature indirectly, but probably will not use it directly in class. This would be eligible for a build badge.

The bottom of the right panel has information about the languages in the project

Getting Help with Programming

This class will help you get better at reading errors and understanding what they might be trying to tell you. In addition here are some more general resources.

Asking Questions

One of my favorite resources that describes how to ask good questions is this blog post by Julia Evans, a developer who writes comics about the things she learns in the course of her work and publisher of wizard zines.

Describing what you have so far

Stackoverflow is a common place for programmers to post and answer questions.
As such, they have written a good guide on creating a minimal, reproducible example.

Creating a minimal reproducible example may even help you debug your own code, but if it does not, it will definitely make it easier for another person to understand what you have, what your goal is, and what's working.

Getting Organized for class

The only **required** things are in the Tools section of the syllabus, but this organizational structure will help keep you on top of what is going on.

Your username will be appended to the end of of the repository name for each of your assignments in class.

File structure

I recommend the following organization structure for the course:

Skip to main content

```
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

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