About this Site

Contents

Syllabus

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

Notes

- 1. Welcome, Introduction, and Setup
- 2. How does this course work?
- 3. Working offline
- 4. How do branches work?
- 5. When do git and bash give me an advantage?
- 6. What *is* a commit?
- 7. How do programmers communicate about code?
- 8. How do I choose a Programming Language for a project?
- 9. Why are these tools like this?
- 10. What *is* git?
- 11. How does git make a commit?
- 12. Why did we learn the plubming commands?
- 13. What is a commit number?
- 14. How can I automate things with bash?
- 15. What is an IDE?

Activities

- KWL Chart
- Team Repo
- Review Badges
- Prepare for the next class

Skip to main content

- KWL File List
- Explore Badges
- Build Badges

FAQ

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

Resources

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

Welcome to the course website for Computer Systems and Programming Tools in Spring 2024 with Professor Brown.

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

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

Navigating the Sections

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

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

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

Skip to main content

Reading each page

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

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

```
command output
important line, emphasized
```

Try it Yourself

Notes will have exercises marked like this

Question from Class

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

Further reading

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

Hint

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

Click here!

Special tips will be formatted like this

Check your Comprehension

Questions to use to check your comprehension will looklike this

Contribute

Chances to earn community badges will sometimes be marked like this

Skip to main content

Computer Systems and Programming Tools

About this course

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

About this syllabus

This syllabus is a *living* document. You can get notification of changes from GitHub by "watching" the repository You can view the date of changes and exactly what changes were made on the Github commit history page.

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

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

No, because the syallabus changes

About your instructor

Name: Dr. Sarah M Brown Office hours: listed on communication page

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

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

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

Land Acknowledgement

> **Important**
>
> The University of Rhode Island land acknowledgment is a statement written by members of the University community in close partnership with members of the Narragansett Tribe. For more information see the university land acknowledgement page

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

Tools and Resources

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

All of the tools and resources below are either:

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

BrightSpace

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

Prismia chat

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

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

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

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

Skip to main content

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
- A web browser compatible with Jupyter Notebooks
- nano text editor (comes with GitBash and default on MacOS)
- one IDE with git support (default or via extension)

Skip to main content

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, programming tools, and components of systems.
2. Apply appropriate programming workflows using context-relevant tools that enable adherance to best practices for effective code, developer time efficiency, and collaboration.
3. Differentiate the different classes of tools used in computer science in terms of their features, roles, and how they interact

Skip to main content

4. Identify how information flows across levels of abstraction.
5. Discuss implications of design choices across levels of abstraction
6. Describe the social context in which essential components of computing systems were developed and explain the impact of that context on the systems.
7. Differentiate between social conventions and technical requirements in programming contexts.

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

Principles of Grading

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

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

- Earning 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.
- 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 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.

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

Skip to main content

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 submitted.

What happens if you merged a PR without feedback?

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

Important

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

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

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

Nothing to worry about, we will review it again in week three after you get a chance to build the right habits and learn vocabulary. There will also be a lab 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 form of engagment is tracked with a particular type of badge that you can earn as you accumulate evidence of your learning and engagment.

- experience: guided in class activities, with reflection
- lab: accountability to basics through 1:1 conversation with a member of the instructional team
- review: just the basics
- practice: a little bit more indepdendent
- explore: posing your own directions of inquiry
- build: in depth application of course topics

Below are the recommended paths to different grade targets.

To earn a D, complete:

- 22 experience badges
- 13 lab check outs

To earn a C, complete:

Skip to main content

- 13 lab check outs
- 18 review badges

To earn a B, complete:

- 22 experience badges
- 13 lab check outs
- one of the following:
  - 18 practice badges
  - 12 review + 12 practice

For an A you must complete:

- 22 experience badges
- 13 lab check outs
- one of the following:
  - 18 practice badges + 6 explore badges
  - 18 review badges + 3 build badges
  - 6 review badges + 12 practice badges + 4 explore badges + 1 build badges
  - 12 review badges + 6 practice badges+ 2 explore badges + 2 build badges

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

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

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

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

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

Skip to main content

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

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

Can you do any combination of badges?

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

Experience Badges

In class

You earn an experience badge in class by:

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

Makeup

You can make up an experience badge by:

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

Tip

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

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

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

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

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

Do you earn badges for prepare for class?

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

What do you do when you miss class?

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

What if I have no questions?

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

Lab Checkouts

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

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

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

Skip to main content

Review and Practice Badges

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

You can earn review and practice badges by:

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

Where do you find assignments?

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

You should create one PR per badge

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

Important

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

Explore Badges

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

Details and more ideas are on the explore page.

You can earn an explore badge by:

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

Skip to main content

- making any requested changes
- merging the PR after approval

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

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

You should create one PR per badge

Build Badges

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

You can earn a build badge by:

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

You should create one PR per badge

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

Community Badges

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

Skip to main content

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

Community Badge values:

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

You can earn community badges by:

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

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

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

Example(nonexhaustive) uses:

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

Skip to main content

Free corrections

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

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

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

Important

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

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

IDEA

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

Ungrading Option

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

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

Skip to main content

What do you think?

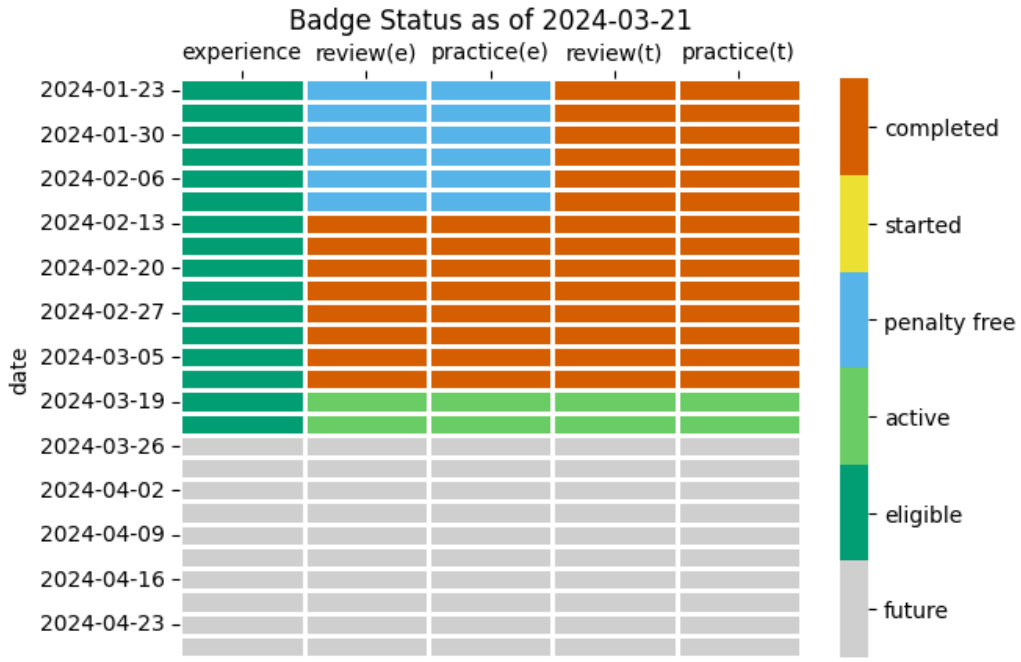
share your thoughts on this option in the discussions for the class and then

Badge Deadlines and Procedures

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

▶ Show code cell source

```
Text(0.5, 1.0, 'Badge Status as of 2024-03-21')
```



Deadlines

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

Experience badges

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

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

Skip to main content

s.

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

Review and Practice Badges

These badges have 5 stages:

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

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

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

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

Expired badges will receive a comment and be closed

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

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

Explore Badges

Explore badges have 5 stages:

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

Explore badges are feedback-limited. You will not get feedback on subsequent explore badge proposals until you earn the first

Skip to main content

time. At most, you

will receive feedback for one explore badge per week, so in order to earn six, your first one must be complete by March 18.

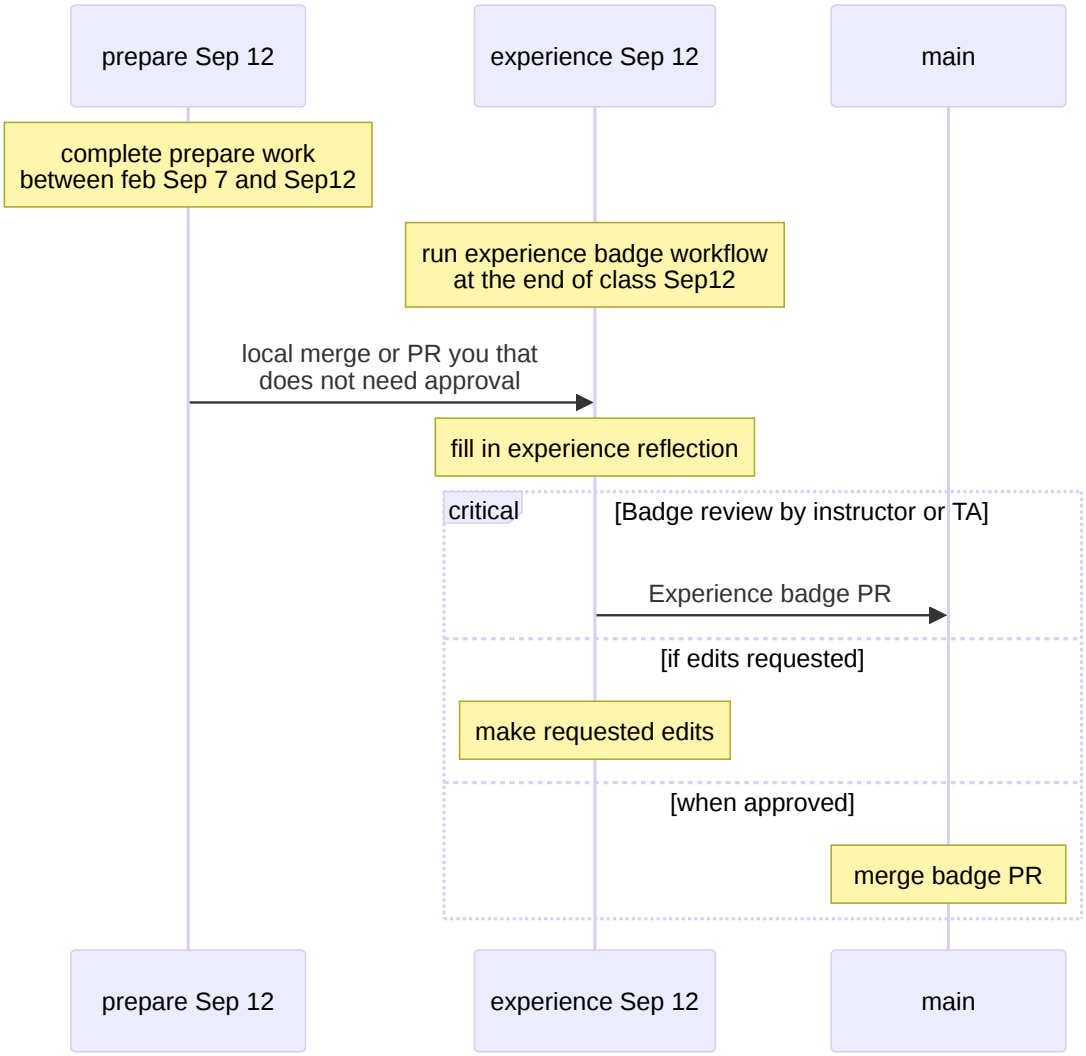
Build Badges

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

Prepare work and Experience Badges Process

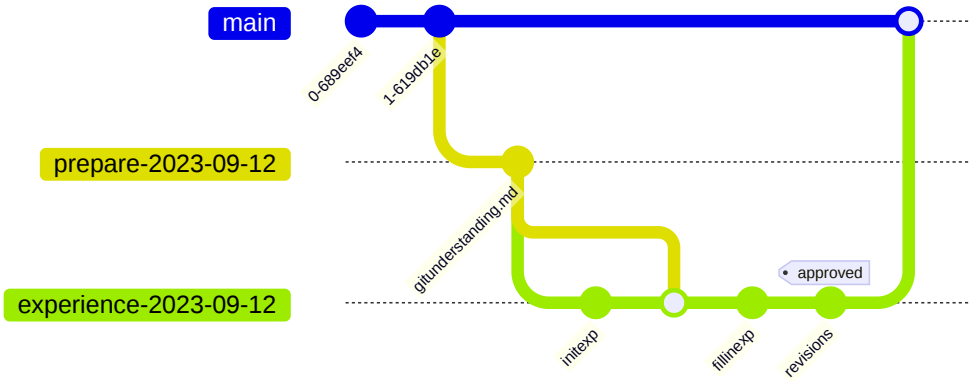
This is for a single example with specific dates, but it is similar for all future dates

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



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

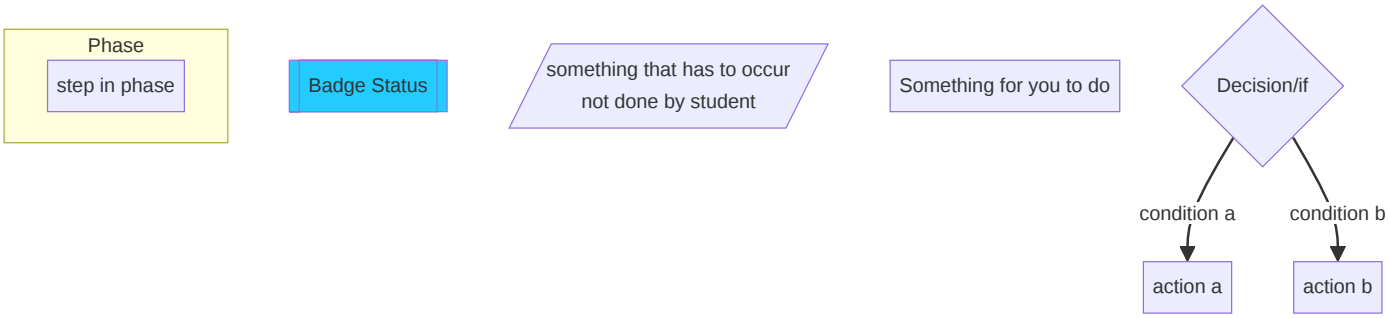
Skip to main content



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

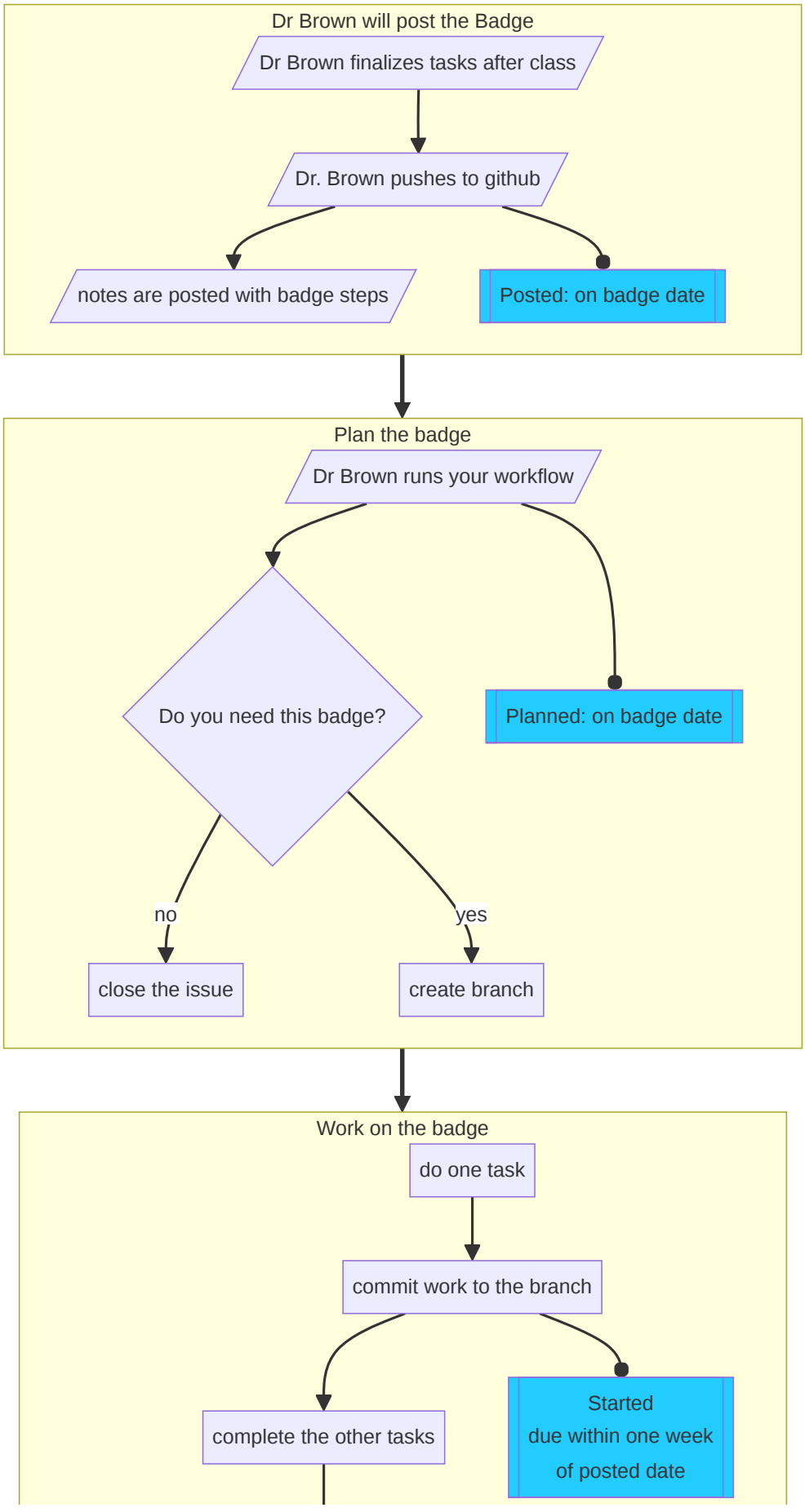
Review and Practice Badge

Legend:

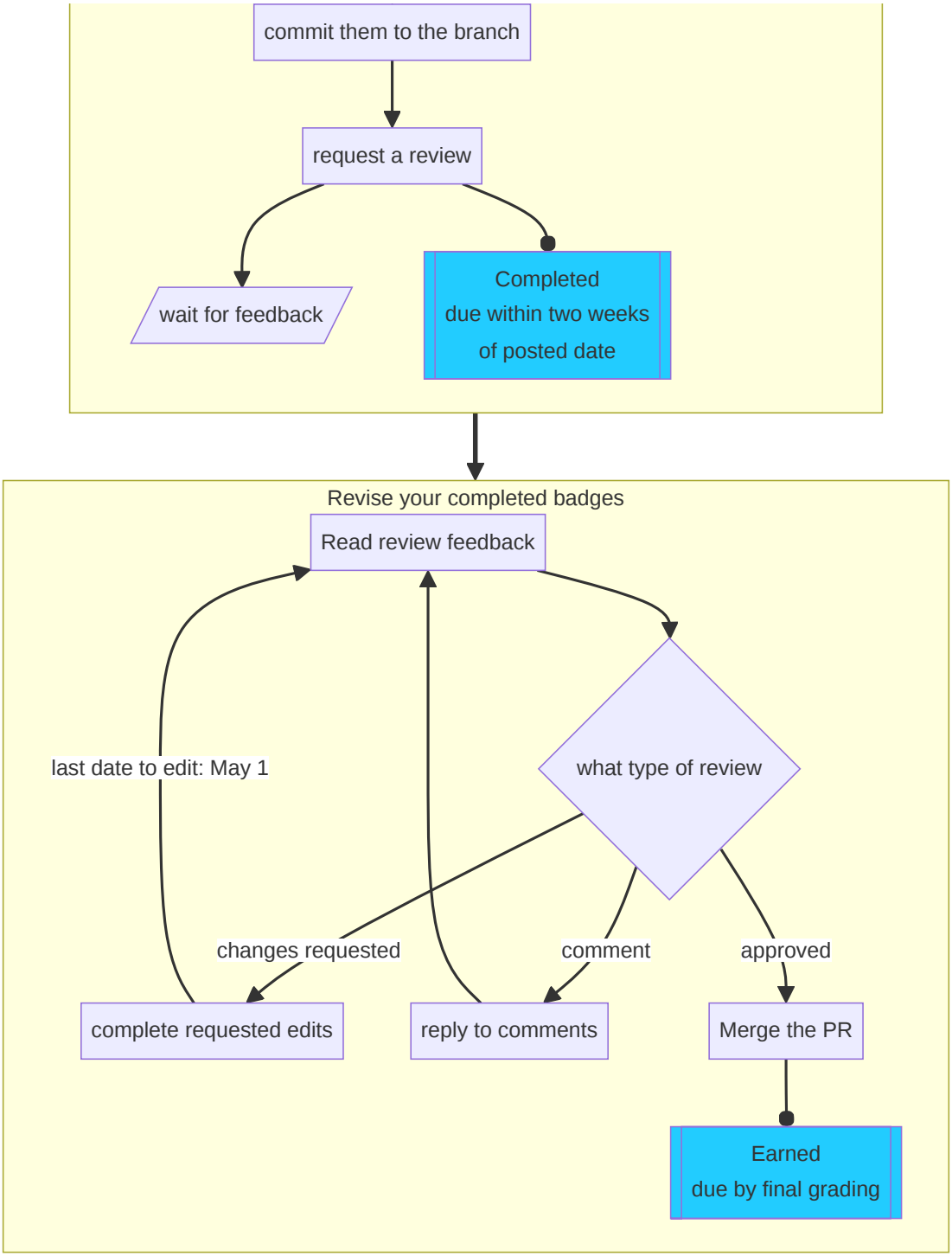


This is the general process for review and practice badges

Skip to main content



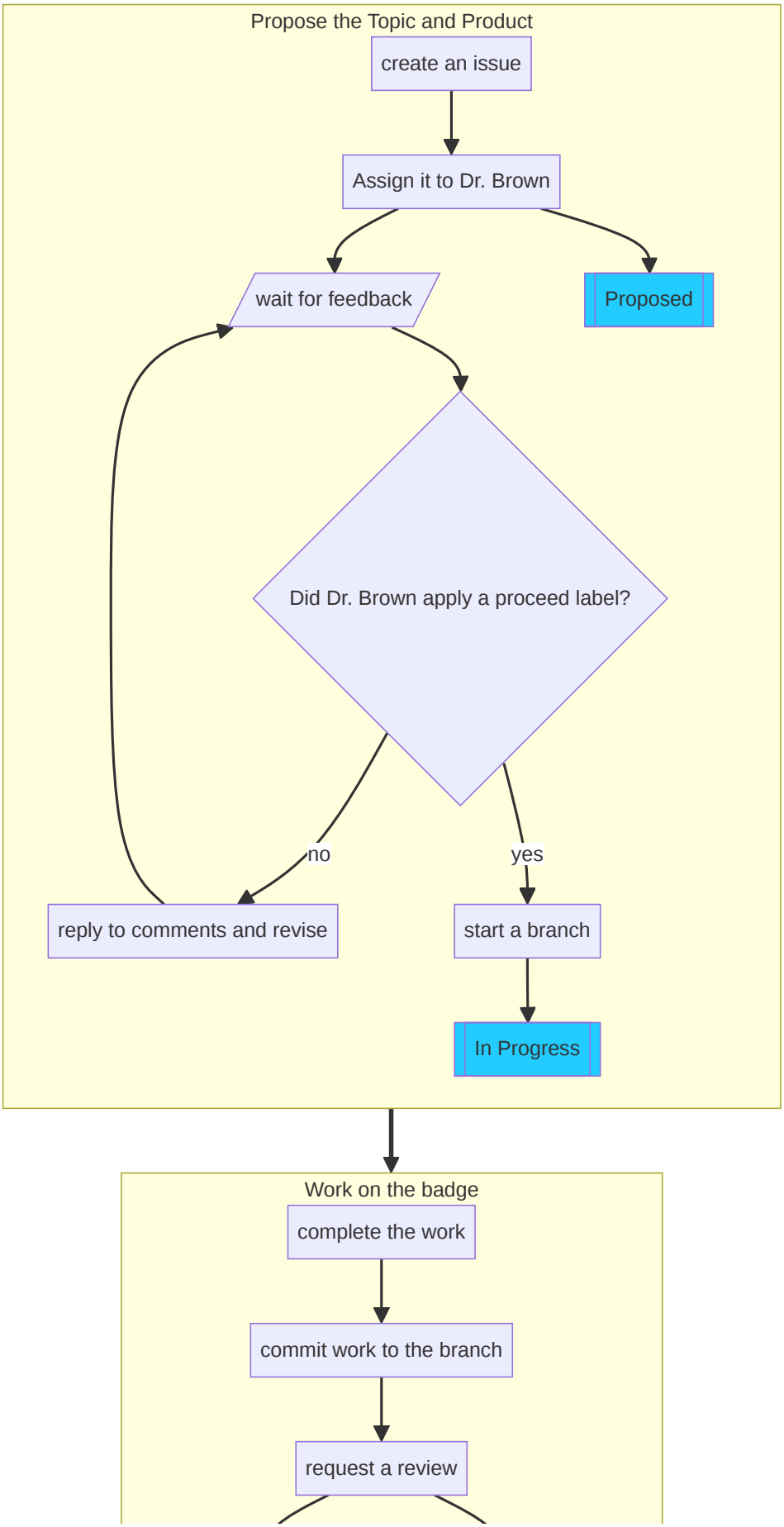
Skip to main content



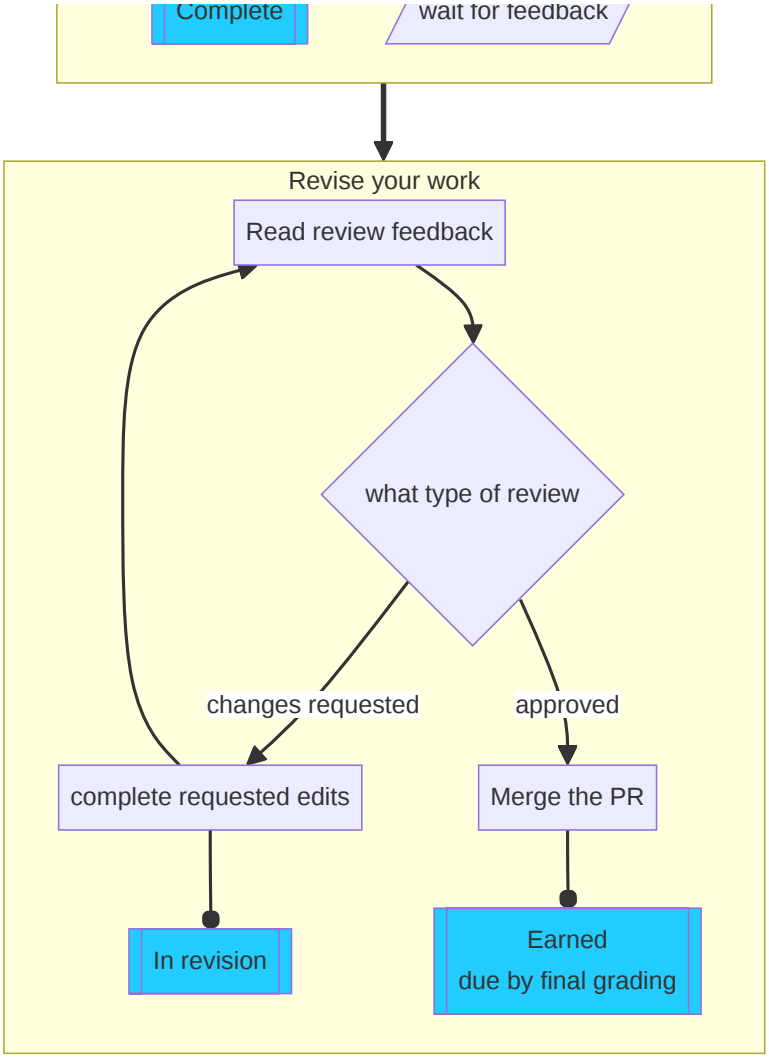
Skip to main content

Explore Badges

Skip to main content



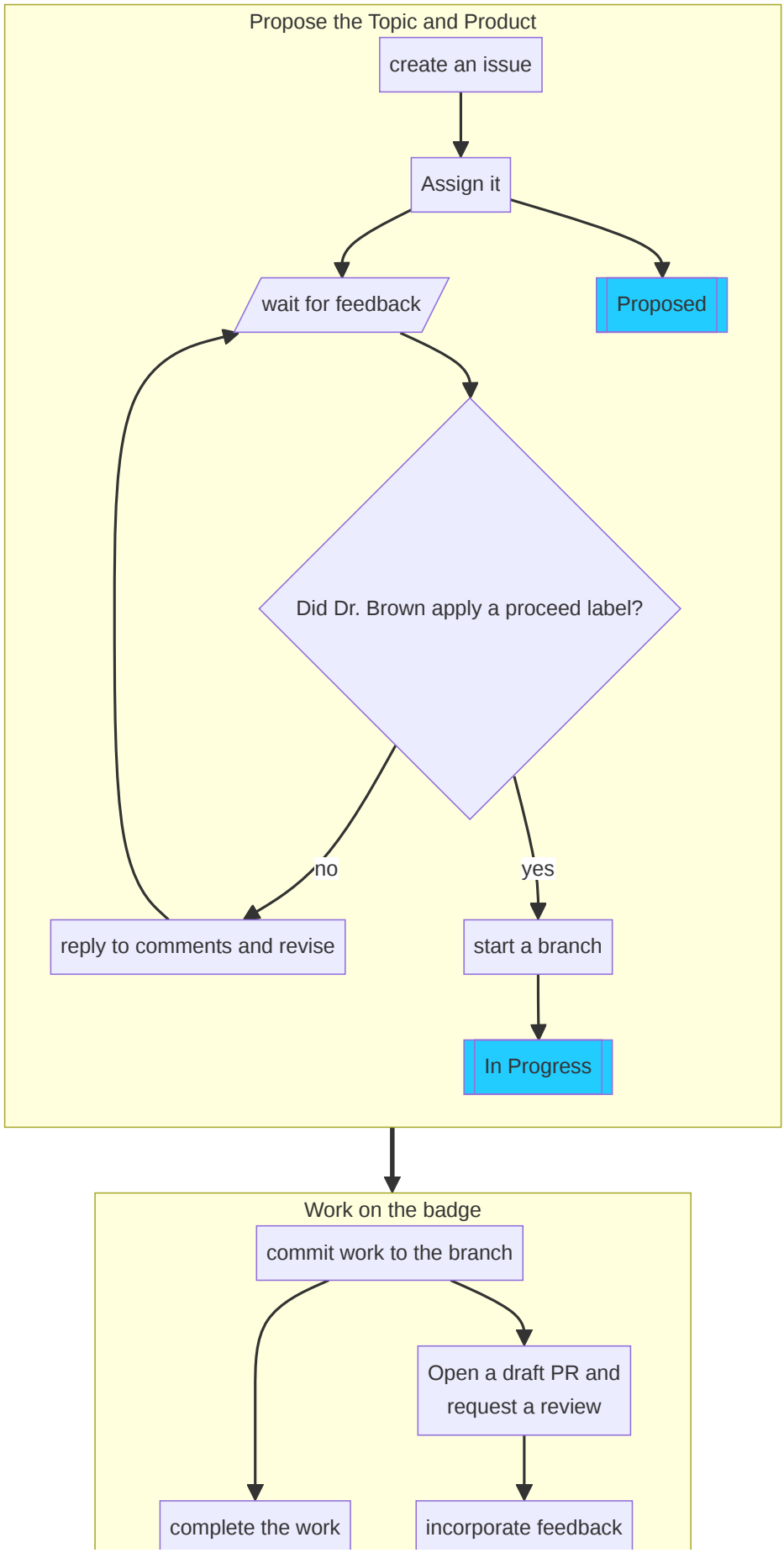
Skip to main content



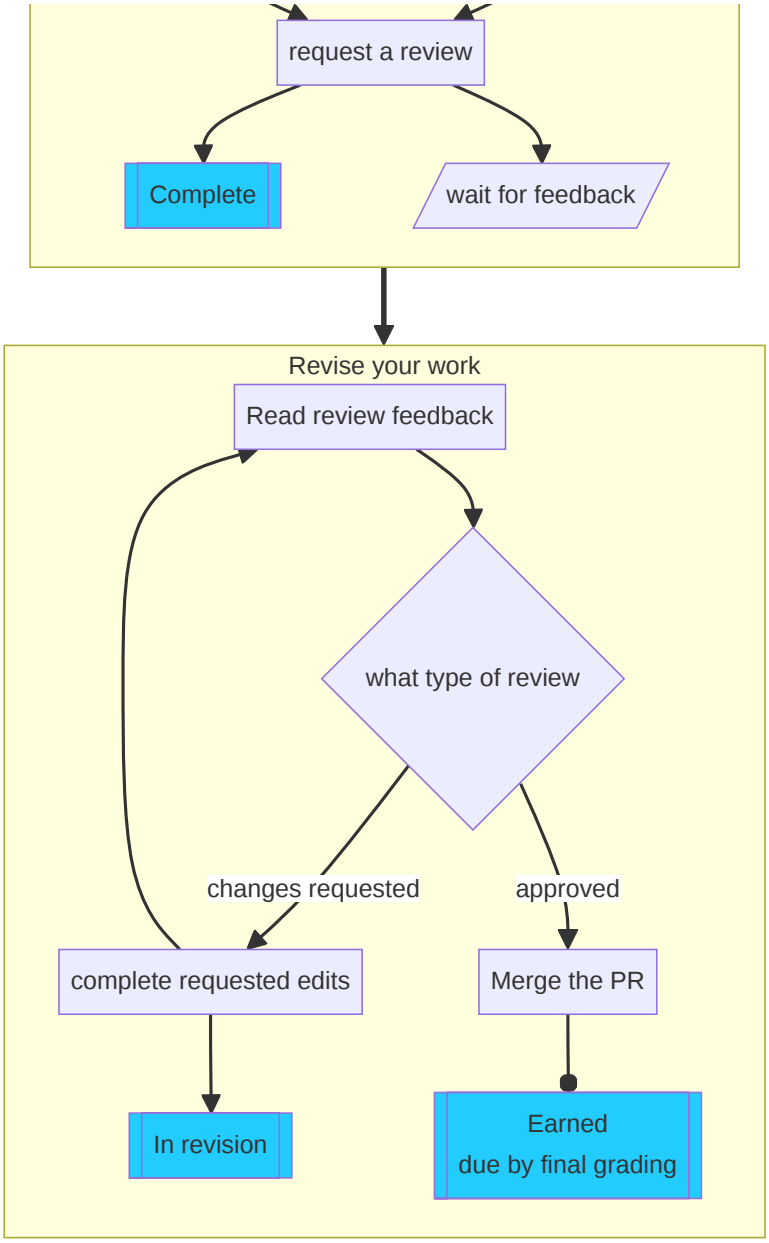
Skip to main content

Build Badges

Skip to main content



Skip to main content



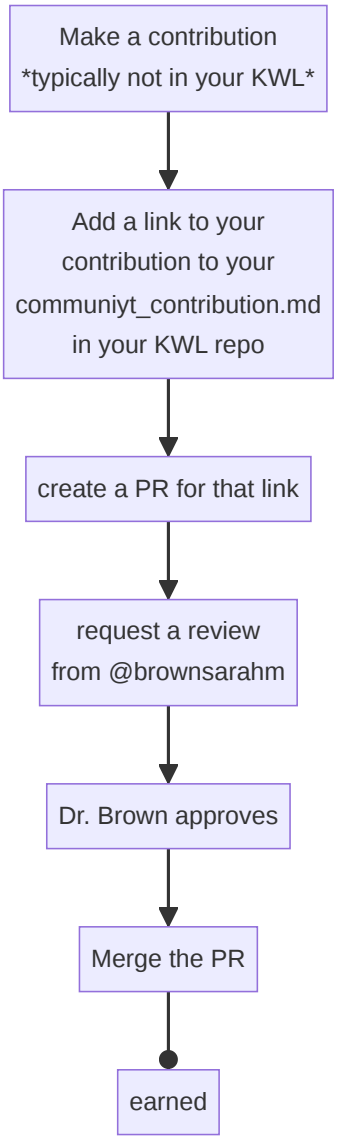
Community Badges

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

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

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

Skip to main content



Detailed Grade Calculations

Important

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

Warning

Some phrasing on this may change, but the core of what is required will not change

▶ Show code cell source

Skip to main content

```
{'D ': 106,
 'D+': 124,
 'C-': 142,
 'C ': 192,
 'C+': 210,
 'B-': 228,
 'B ': 246,
 'B+': 264,
 'A-': 282,
 'A ': 300}
```

▶ Show code cell source

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/indexes/base.py:3653,
   3652 try:
-> 3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/_libs/index.pyx:147, in pa

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/_libs/index.pyx:176, in pa

File pandas/_libs/hashtable_class_helper.pxi:7080, in pandas._libs.hashtable.PyObjectHashTable.get_item()

File pandas/_libs/hashtable_class_helper.pxi:7088, in pandas._libs.hashtable.PyObjectHashTable.get_item()

KeyError: 'weight'

The above exception was the direct cause of the following exception:

KeyError                                  Traceback (most recent call last)
Cell In[2], line 28
     26 influence_df = learning_df.fillna(1).rename(columns={'index':'badge'})
     27 # final df
---> 28 influence_df['influence'] = influence_df['complexity']*influence_df['weight']

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/frame.py:3761, in Dat
   3759 if self.columns.nlevels > 1:
   3760     return self._getitem_multilevel(key)
-> 3761 indexer = self.columns.get_loc(key)
   3762 if is_integer(indexer):
   3763     indexer = [indexer]

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/indexes/base.py:3655,
   3653     return self._engine.get_loc(casted_key)
   3654 except KeyError as err:
-> 3655     raise KeyError(key) from err
   3656 except TypeError:
   3657     # If we have a listlike key, _check_indexing_error will raise
   3658     #  InvalidIndexError. Otherwise we fall through and re-raise
   3659     #  the TypeError.
   3660     self._check_indexing_error(key)

KeyError: 'weight'
```

The total influence of each badge on the grade is as follows:

▶ Show code cell source

Skip to main content

```
---------------------------------------------------------------------------
KeyError                                  Traceback (most recent call last)
Cell In[3], line 2
      1 # display
----> 2 influence_df[['badge_type','badge','complexity','weight','influence']]

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/frame.py:3767, in Dat
   3765     if is_iterator(key):
   3766         key = list(key)
-> 3767     indexer = self.columns._get_indexer_strict(key, "columns")[1]
   3769 # take() does not accept boolean indexers
   3770 if getattr(indexer, "dtype", None) == bool:

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/indexes/base.py:5877,
   5874 else:
   5875     keyarr, indexer, new_indexer = self._reindex_non_unique(keyarr)
-> 5877 self._raise_if_missing(keyarr, indexer, axis_name)
   5879 keyarr = self.take(indexer)
   5880 if isinstance(key, Index):
   5881     # GH 42790 - Preserve name from an Index

File /opt/hostedtoolcache/Python/3.8.18/x64/lib/python3.8/site-packages/pandas/core/indexes/base.py:5941,
   5938     raise KeyError(f"None of [{key}] are in the [{axis_name}]")
   5940 not_found = list(ensure_index(key)[missing_mask.nonzero()[0]].unique())
-> 5941 raise KeyError(f"{not_found} not in index")

KeyError: "['weight', 'influence'] not in index"
```

The total influence of a badge on your grade is the product of the badge's complexity. All learning badges have a weight of 1, but have varying complexity.

Bonuses

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

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

Skip to main content

Name	Definition	Influence	type
Participation	22 experience badges	18	auto
Lab	13 lab badges	18	auto
Breadth	If review + practice badges >-18:	32	auto
Git-ing unstuck	fix large mistakes your repo using advanced git operations and submit a short reflection (allowable twice; Dr. Brown must approve)	9	event
Early bird	6 review + practice submitted by 2/19	9	event
Descriptive commits	all commits in KWL repo and build repos after penalty free zone have descriptive commit messages (not GitHub default or nonsense)	9	event
Community Star	10 community badges	18	auto

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

Note

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

Note

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

Grade thresholds

Grade cutoffs for total influence are:

▶ Show code cell source

Skip to main content

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

Bonus Implications

Attendance and participation is *very* important:

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

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

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

Example 2:

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

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

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

Skip to main content

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

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

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

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

The order of application for community badges:

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

To calculate your final grade at the end of the semester, a script will count your badges and logged event bonuses. The script will output a dictionary with keys for each type of learning badge and event bonus with a count for the value of each.

```
example_student = {'experience' :21, 'lab': 13, 'review': 6,'practice': 12,
                   'explore': 2,
                   'build' :1,
                  'community': 3,
                  'unstuck': 0,
                  'descriptive': 1,
                  'early': 1 }
```

Then these counts will go into the following function to calculate the final grade

> **Warning**
>
> This is not complete, but will be before the end of the penalty free zone.

Skip to main content

```
# set up remaining constants (some are above)
bonus_criteria = {'participation_bonus': lambda r: int(r['experience'] >=22),
                  'lab_bonus':  lambda r: int(r['lab'] >=13),
                   'breadth_bonus': lambda r: int(r['review'] + r['practice']>=18),
                 'community_bonus': lambda r: int(r['community']>=10),
                 'unstuck_bonus': lambda r: r['unstuck'],
                 'descriptive_bonus': lambda r: r['descriptive'],
                 'early_bonus': lambda r: r['early'] }
bonus_values = {'participation_bonus': bonus_participation,
                  'lab_bonus':  bonus_lab,
                   'breadth_bonus': bonus_breadth,
                 'community_bonus': 18,
                 'unstuck_bonus': 9,
                 'descriptive_bonus': 9,
                 'early_bonus': 9 }
weights = learning_weights.copy()
weights.update(bonus_values)
community_thresh = {'experience':22,
                 'review':4,
                 'practice':7,
                 'review_upgrade':3}
community_cost = {'experience':3,
                 'review':4,
                 'practice':7,
                 'review_upgrade':3}

# compute grade

# def calculate_grade(badge_dict):
#     if badge_dict['community']>0:
        # apply community badges

    # add bonuses


    # final sum
    # sum([weights[cat]*count_dict[cat] for cat in count_dict.keys()])
```

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.

Topics:

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

What Happens When I run code?

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

Topics:

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

Recommended workload distribution

> **Note**
>
> General badge deadlines are on the detailed badge procedures page.

To plan your time, I recommend expecting the following:

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

For each explore:

- 7 hours for the project

For each build:

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

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

Tentative Timeline

> **Warning**
>
> This section is not yet updated for spring 2024.
>
> This is a rough example.

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

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

Skip to main content

date	question	keyword	conceptual	practical	social	activity
2023-09-07	Welcome, Introduction, and Setup	intro	what is a system, why study tools	GitHub basics	class intros	create kwl repo in github, navigate github.com...
2023-09-12	Course Logistics and Learning	logistics	github flow with issues	syllabus	working together and building common vocab	set up to work offline together, create a folder
2023-09-14	Bash intro & git offline	terminal start	git structure, paths and file system	bash path navigation, git terminal authentication	why developers work differently than casual users	navigate files and clone a repo locally
2023-09-19	How can I work with branches offline?	gitoffline	git branches	github flow offline, resolving merge conflicts	commuication is important, git can help fix mi...	clone a repo and make a branch locally
2023-09-21	When do I get an advantage from git and bash?	why terminal	computing mental model, paths and file structure	bash navigation, tab completion	collaboration requires shared language, shared...	work with bash and recover from a mistake with...
2023-09-26	What *is* a commit?	merge conflicts	versions, git vlaues	merge conflicts in github, merge conflicts wit...	human and machine readable, commit messages ar...	examine commit objects, introduce plumbing com...
2023-09-28	How do programmers communicate about code?	documentation	build, automation, modularity, pattern matching,	generate documentation with jupyterbook, gitig...	main vs master, documentation community	make a jupyterbook
2023-10-03	What *is* git?	git structure	what is a file system, how does git keep track...	find in bash, seeing git config, plumbing/porc...	git workflows are conventions, git can be used...	examine git from multiple definitions and insp...
2023-10-05	Why are these tools like this?	unix philosophy	unix philosophy, debugging strategies	decision making for branches	social advantages of shared mental model, diff...	discussion with minor code examples
2023-10-12	How does git make a commit?	git internals	pointers, design and abstraction, intermediate...	inspecting git objects, when hashes are unique...	conventions vs requirements	create a commit using plumbing commands
2023-10-17	What is a commit number?	numbers	hashes, number systems	git commit numbers, manual hashing with git	number systems are derived in culture	discussion and use hashing algorithm
2023-10-19	How can can I release and share my code?	git references	pointers, git branches and tags	git branches, advanced fixing, semver and conv...	advantages of data that is both human and mach...	make a tag and release
2023-10-24	How can I automate things with bash?	bash scripting	bash is a programming language, official docs,...	script files, man pages, bash variables, bash ...	using automation to make collaboration easier	build a bash script that calculates a grade

Skip to main content

log into a remote erver and work with

date	question	keyword	conceptual	practical	social	activity
2023-10-31	What is an IDE?	IDE	IDE parts	compare and contrast IDEs	collaboraiton features, developer communities	discussions and sharing IDE tips
2023-11-02	How do I choose a Programming Language for a p...	programming languages	types of PLs, what is PL studying	choosing a language for a project	usability depends on prior experience	discussion or independent research
2023-11-07	How can I authenitcate more securely from a te...	server use	ssh keys, hpc system strucutre	ssh keys, interactive, slurm	social aspects of passwords and security	configure and use ssh keys on a hpc
2023-11-09	What Happens when we build code?	building	building C code	ssh keys, gcc compiler	file extensions are for people, when vocabular...	build code in C and examine intermediate outputs
2023-11-14	What happens when we run code?	hardwar	von neuman architecture	reading a basic assembly language	historical context of computer architecures	use a hardware simulator to see step by step o...
2023-11-16	How does a computer represent non integer quan...	floats	float representation	floats do not equal themselves	social processes around standard developents, ...	work with float representation through fractio...
2023-11-21	How can we use logical operations?	bitwise operation	what is a bit, what is a register, how to brea...	how an ALU works	tech interviews look for obscure details somet...	derive addition from basic logic operations
2023-11-28	What *is* a computer?	architecture	physical gates, history	interpretting specs	social context influences technology	discussion
2023-11-30	How does timing work in a computer?	timing	timing, control unit, threading	threaded program with a race condition	different times matter in different cases	write a threaded program and fix a race condition
2023-12-05	How do different types of storage work together?	memory	different type of memory, different abstractions	working with large data	privacy/respect for data	large data that has to be read in batches
2023-12-07	How does this all work together	review	all	end of semester logistics	group work final	review quiz, integration/reflection questions
2023-12-12	How did this semester go?	feedback	all	grading	how to learn better together	discussion

Tentative Lab schedule

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

Skip to main content

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

Support

Academic Enhancement Center

Academic Enhancement Center (for undergraduate courses): Located in Roosevelt Hall, the AEC offers free face-to-face and web-based services to undergraduate students seeking academic support. Peer tutoring is available for STEM-related courses by appointment online and in-person. The Writing Center offers peer tutoring focused on supporting undergraduate writers at any stage of a writing assignment. The UCS160 course and academic skills consultations offer students strategies and activities aimed at improving their studying and test-taking skills. Complete details about each of these programs, up-to-date schedules, contact information and self-service study resources are all available on the AEC website.

- **STEM Tutoring** helps students navigate 100 and 200 level math, chemistry, physics, biology, and other select STEM courses. The STEM Tutoring program offers free online and limited in-person peer-tutoring this fall. Undergraduates in introductory STEM courses have a variety of small group times to choose from and can select occasional or weekly appointments. Appointments and locations will be visible in the TutorTrac system on September 14th, FIXME. The TutorTrac application is available through URI Microsoft 365 single sign-on and by visiting aec.uri.edu. More detailed information and instructions can be found on the AEC tutoring page.
- **Academic Skills Development** resources helps students plan work, manage time, and study more effectively. In Fall FIXME, all Academic Skills and Strategies programming are offered both online and in-person. UCS160: Success in Higher Education is a one-credit course on developing a more effective approach to studying. Academic Consultations are 30-minute, 1 to 1 appointments that students can schedule on Starfish with Dr. David Hayes to address individual academic issues. Study Your Way to Success is a self-guided web portal connecting students to tips and strategies on studying and time management related topics. For more information on these programs, visit the Academic Skills Page or

- The **Undergraduate Writing Center** provides free writing support to students in any class, at any stage of the writing process: from understanding an assignment and brainstorming ideas, to developing, organizing, and revising a draft. Fall 2020 services are offered through two online options: 1) real-time synchronous appointments with a peer consultant (25- and 50-minute slots, available Sunday - Friday), and 2) written asynchronous consultations with a 24-hour turn-around response time (available Monday - Friday). Synchronous appointments are video-based, with audio, chat, document-sharing, and live captioning capabilities, to meet a range of accessibility needs. View the synchronous and asynchronous schedules and book online, visit uri.mywconline.com.

General Policies

Anti-Bias Statement:

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

Disability, Access, and Inclusion Services for Students Statement

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

Academic Honesty

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

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

- Fabricating or falsifying facts, data or references
- Facilitating or aiding another's academic dishonesty
- Submitting the same paper for more than one course without prior approval from the instructors

> **Tip**
>
> Assignments are tested against LLMs and designed so that they require you to learn and that an LLM answer will be low quality and not earn credit.

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

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

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

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

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

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

Viral Illness Precautions

The University is committed to delivering its educational mission while protecting the health and safety of our community. Students who are experiencing symptoms of viral illness should NOT go to class/work. Those who test positive for COVID-19 should follow the isolation guidelines from the Rhode Island Department of Health and CDC.

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

Excused Absences

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

Skip to main content

For short absences (1-2 classes) for any reason, you can follow the makeup procedures.

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

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

Mental Health and Wellness:

We understand that college comes with challenges and stress associated with your courses, job/family responsibilities and personal life. URI offers students a range of services to support your mental health and wellbeing, including the URI Counseling Center, MySSP (Student Support Program) App, the Wellness Resource Center, and Well-being Coaching.

Office Hours & Communication

Announcements

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

Warning

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

Sign up to watch

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

```
- [watched the repo as per announcements](https://introcompsys.github.io/spring2023/syllabus/comm
```

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

Help Hours

Day	Time	Location	Host
Monday	4:45-6:30	052 Tyler	Trevor
Tuesday	3:00-4:00pm	134 Tyler	Dr. Brown
Tuesday	11-12:30	052 Tyler	Trevor
Tuesday	2:00 - 4:00	Zoom	Marcin
Wednesday	2:00 - 4:00	049 Tyler	Gyanko
Thursday	11-12:30	052 Tyler	Trevor
Friday	2:00 - 4:00	049 Tyler	Gyanko
Friday	see GH org site	Zoom	Dr. Brown

Online office hours locations are linked on the GitHub Organization Page

Important

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

Tips

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 isses

- 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.

Skip to main content

For E-mail

- use e-mail for general inquiries or notifications
- Include `[CSC392]` in the subject line of your email along with the topic of your message. This is important, because your messages are important, but I also get a lot of e-mail. Consider these a cheat code to my inbox: I have setup a filter that will flag your e-mail if you include that in subject to ensure that I see it.

> **Should you e-mail your work?**
>
> No, request a pull request review or make an issue if you are stuck

1. Welcome, Introduction, and Setup

1.1. Introductions

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

1.2. Prismia

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

1.3. This course will be different

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

1.4. Getting started

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

1.5. We havea Glossary!!

For example, the term we used above:

repository

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

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

Key terms for the first class:

- repository
- git
- github
- PR

1.6. GitHub Docs are really helpful and have screenshots

1.7. First KWL setup steps

Today we did the following:

1. Accept the assignment to create your repo: KWL Chart
2. Edit the README to add your name by clicking the pencil icon (editing a file step 2)
3. adding a descriptive commit message and commited directly to main (editing a file step 5)
4. Edit the README to fill in one row of the table by clicking the pencil icon (editing a file step 2)
5. created a new branch (named `prior_knowledge`) (editing a file step 7-8)
6. added a message to the Pull Request (pull request step 5)
7. Creating a pull request (pull request step 6)
8. Clicking Merge Pull Request

1.8. GitHub Actions

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

We ran the `Experience Reflection` action at the end of class.

Skip to main content

1.9. Questions After Class

1.9.1. is there a limit to how many branches you can have from the main branch?

Not specifically. Technically there are limits, but they're second hand and we will see them later

1.9.2. what should I name the commit for the experience badge?

something like "reflected on class" or if you make an extra one where you add more detail you could use "add more detail"

1.9.3. When did Git and GitHub get created?

git was created in 2005. GitHub was founded in 2008.

1.9.4. Do I commit directly to the branch when I edit my experience badge?

yes

1.9.5. will these experience files be auto-generated each class?

You will run the action manually, but it will create and name the files for you.

1.9.6. Is making Pull Requests our way of submitting work?

Yes

1.9.7. What is the action doing

Copying a file, naming it with that day's date, making a branch with today's date, and opening a PR with a specific body and title.

1.9.8. How does GitHub actions work and what is it useful for?

This is a big picture question for the github docs to answer.

2. How does this course work?

Today we will:

- continue getting familiar with the structure of GitHub

Skip to main content

- practice with new vocabulary

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

2.1. Warm up

1. Navigate to your KWL repo
2. Find the issues tab
3. Open the prepare-2024-01-25 issue and follow those instructions

*hint: my KWL repo URL is: `https://github.com/compsys-progtools/kwl-sp24-brownsarahm`

2.2. What is it like to know things really well?

When you know something well:

- it becomes automatic, you can do it without thinking about the details
- you are able to anticipate things
- you use specialized vocabulary with ease

> **Important**
>
> The goal is to get you to that point with alal of the developer tools we will learn about.
>
> If the tools become automatic in this class, you an *use* them with ease in other classes to your advantage.

> **Note**
>
> We did some prismia questions on GitHub things to reinforce that material we had already seen. When we do this, I'll typically skip that section in the posted notes, but you can always view on prismia including getting a transcript from prisma:
>
> 1. In the top left corner there is a `>` icon which opens a sidebar popup.
> 2. In the top left opens a menu
> 3. "Get transcript from class"

2.3. Pull Requests and Commits

The unit of changes in a histoy is a commit. Each commmit is said to be "on" a branch.

We examined the PRs for your first experience badge.

Skip to main content

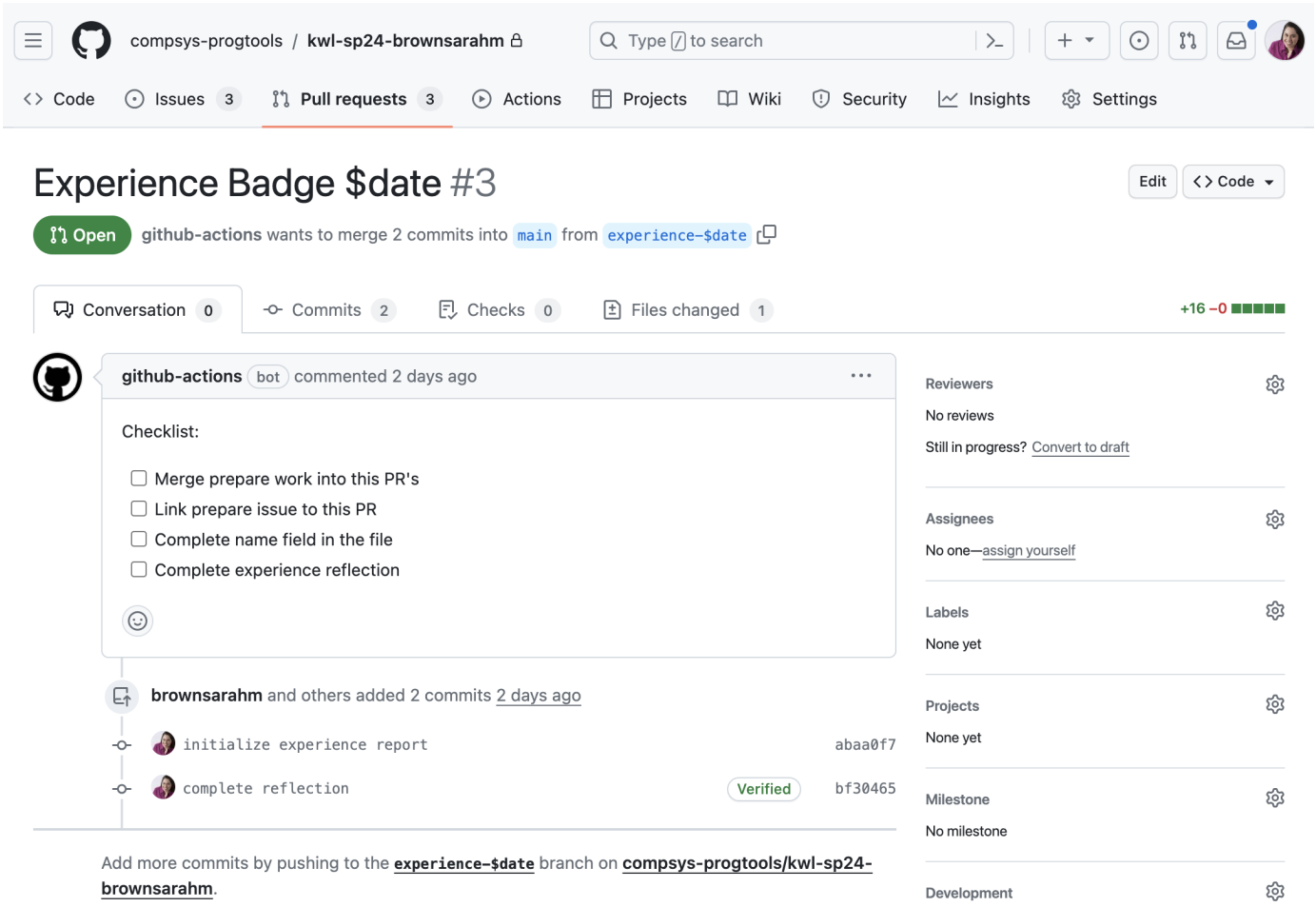


Fig. 2.1 A PR that has two commits on it with one file changed. This PR compares the `experience-$date` branch to the `main` branch.

From this page we can learn an important implication of the fact that **a pull request is a comparison between two branches**. Since the relationship is between the branches, it is not related directly to commits. We can *add* more commits to the branch after we make a PR, and they show up on the PR page.

At the bottom of this screenshot which is right above the checks for the PR is a reminder from GitHub:

> Add more commits by pushing to the experience-$date branch on compsys-progtools/kwl-sp24-brownsarahm.

Note

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

This course will proceed like this a lot; we *do* things to let you experience the important to know things *in context* first, then discuss and label and examine.

Hint

Skip to main content

2.4. Why learn by example?

> just play with it

- common attitude in CS
- not optimal for learning

It's not optimal for learning because it can leave you unguided and not knwo where to start, but it is the common culture in developer environments.

My goal is to teach you to thrive within that culture, because you are likely to encounter it at work.

My goal, however, is not to just drop you in with no preparation. If it feels that way, **please ask questions**.

We'll practice what it *looks like* to learn by tinkering and then examining.

To do this:

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

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

Experience badges are evidence of having learned, you get credit for engaging in the *process* of learning.

2.5. Admin

1. Go to you PR tab
2. Find the PR titled "Template updates" that I created
3. Merge and confirm the PR

2.6. Commit history & PRs

When we merge a PR into `main` all of the commits on the PR compare branch are added to `main` + an additional commit for the merge itself.

For easy merges, that commit does not add more changes to any files, but if the merge is not automatic, it can.

2.7. This course will be different

- no Brightspace
- 300 level = more independence

Skip to main content

- High expectations, with a lot of flexibility

2.7.1. I do not judge your reasons for missing class.

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

If you do email me about missing a single clss, I will likely not reply. Not because I do not care about your long term success; I do! I get too many emails. I try to prioritize time on things that has the biggest impact; confirming I saw an email that does not change any other policies is lower impact than, for example giving feedback on student work.

2.7.2. 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

> **Important**
>
> You **will** be asked to revise things at some point, for 2 reasons:
>
> - since I allow revisions, I enforce really high standards on the quality of your work so that you do not hold onto misconceptions
> - the process of using GitHub to make revisions is something to get familiar with.

2.7.3. Learning comes in many forms

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

2.8. Learning is the goal

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

Skip to main content

2.9. What about AI?

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

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

2.10. This is a college course

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

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

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

2.12. How does this work?

2.12.1. In class:

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

2.12.2. Outside of class:

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

Skip to main content

2.13. How to be successful

> **Important**
>
> There are links on the side to advice from previous semesters, details in the grading have changed, but the core is still the same.

I give a time breakdown in the syllabus.

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

This time breakdown is planned based on if the time-blocking strategy is helpful to you, but the most important thing is for you to make a plan that will work for you. I want this to be flexible enough that you can adapt it to your liking, but that mean you have to make choices.

2.14. 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?

We will:

- 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).

2.14.1. How it fits into your CS degree

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

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

(for BS) in CSC 340 and 440 you study the algorithms more mathematically, their complexity, etc.

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

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

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

Skip to main content

2.14.2. Programming is Collaborative

There are two very common types of collaboration

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

We are going to build your skill in the *code review* model, especially.

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

2.15. GitHub Actions Tab Review

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

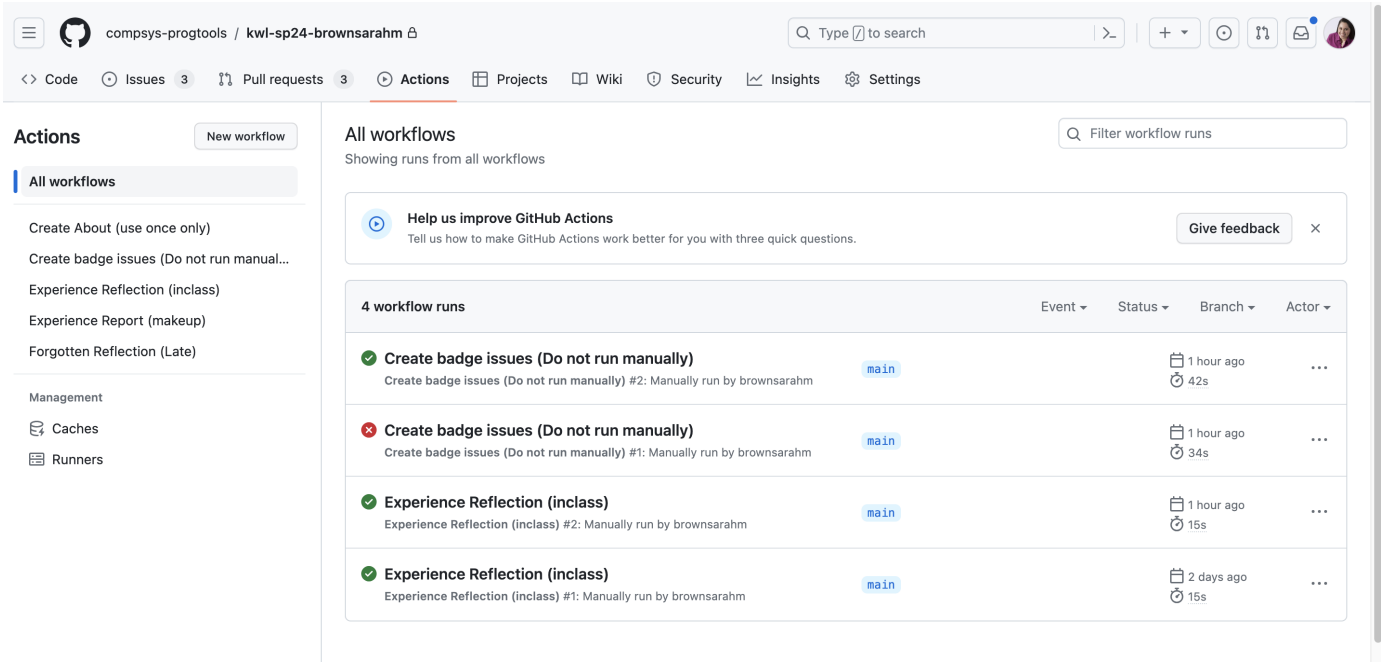


Fig. 2.2 Screen shot of the actions tab of my repo showing 4 total runs in the center panel. Three of the runs succeeded and have a green check mark. One failed and has a red x. On the left hand side there is a list of 5 possible actions to run. Notice the list of workflows on the left each has a unique name (because we generally choose to name things so that names can be used as identifies) the list of workflow runs includes the same name multiple times because we can *run* each workflow multiple times.

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

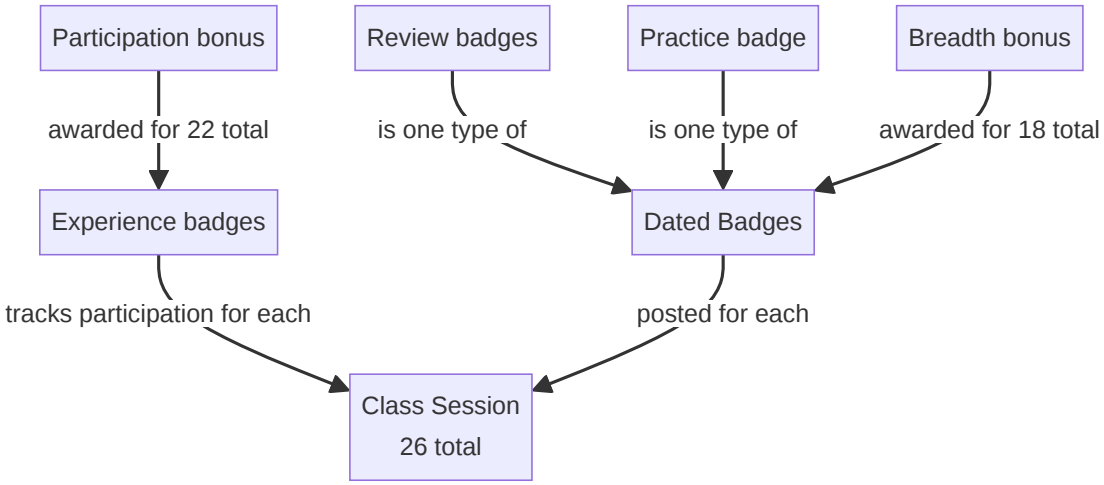
2.15.1. Get Credit for Today's class

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

Skip to main content

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

2.16. Reminder about class structure



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

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

> **Hint**
>
> Remember, this website is generated from a GitHub repo, you can find it from the course organization page on GitHub. The organization is named `compsys-progtools` so the org page is at `https://github.com/compsys-progtools` and it is the owner of your KWL repo so there is a link to it in the top left corner.

2.17. Prepare for lab

this is really a tip/hint about what we will do in the next lab

Next lab we will give you help with sorting out the procedures of PRs and issues and submitting work.

If you work on the badges, but you are not sure about things, log your work as comments on the issue or even in a separate file somewhere, and we will help you get it submitted in lab.

You can also bring questions about anything in the syllabus to get help (or post them for written answers).

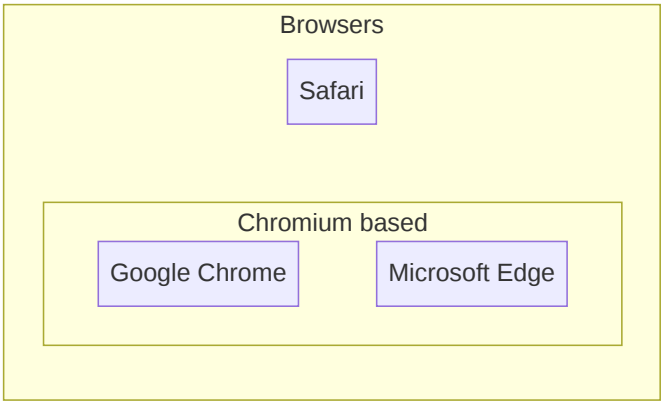
2.18. Prepare for next class

1. Find the glossary page for the course website, link it below. Review the terms for the next class: shell, terminal, bash, git, other

Skip to main content

2. Check your kwl repo before class and see if you have recieved feedback, reply or merge accordingly.

Example “venn diagram ” with mermaid subgraphs



2.19. Badges

Review | **Practice**

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 or what you know about git/GitHub so far and add it to your kwl chart repo in a file called prior-knowledge-map.md. Use mermaid

2.20. Questions After Class

2.20.1. In experience badge, after I commit, do I do anything after?

Yes, request a review from `@instructors`

2.20.2. Are we adding a reviewer to our experience badge?

Yes

2.20.3. do we check off the things in the experience badge checklist or does a TA do that?

In the future, you will, but we’ll work through what that looks like in more detail next week.

2.20.4. do we do both the review and prepare?

Only one per date. If you look, they’re very similar. For the first class, they’re just about the same. As we go on for the first few weeks the difference between them will increase (which is mostly complexity or depth of understanding that I am looking to

2.20.5. How do I submit work?

Create a PR and request a review.

Two things to note:

- *in the penalty free zone, there is no penalty for doing it wrong*
- checking this is the main topic of the next lab, so we will one on one check in with each of you that you know how to do this

2.20.6. If we wanted to ping an instructor/the instructor that got assigned to us, would we @ them in a comment?

Yes! you can @ mention any of us or the team whenever you need help.

2.21. A final note

> **Tip**
>
> Reading to the end is always valued, you can claim a community badge for finding this by linking to the heading above and requesting a review from @brownsarahm, title your PR `community-reading-notes`.
>
> Extra bonus, there are tips like this throughout the website that let you get community badges for setting things up, or reading carefully.

3. Working offline

3.1. Why do we need this for computer systems?

3.1.1. Computer Systems are designed by people

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

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

Of course, not *all* programmers think the same way, but when people spend time together and communicate, they start to

will help you reading code, and understanding things.

3.1.2. 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.

3.1.3. 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.

3.2. Let’s get organized

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

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

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

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

On mac you can optionally use the `bash` command to switch to bash.

```
bash
```

```
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.
```

Then it tells you Apple’s default is somethign else and how to switch back.

We can use `pwd` you can see your current “location”. It stands for **p**rint **w**orking **d**irectory.

```
pwd
```

this is called the path and specifically this is an absolute path, we can tell because it starts with `/` that is the “root” or “top” of the directory tree structure.

```
/Users/brownsarahm
```

Skip to main content

> **Tip**
>
> You can think of absolute paths like a street address. It describes the location in a general (absolute) way; not relative to another location.
>
> I can say Tyler hall is at 9 Greenhouse Road and you can get there from anywhere.
>
> In contrast if I say it is that building in the distance, across the quad past the large glass building, that works from our classroom in Ranger, but not from the library, or anywhere off campus. This would be a relative set of directions to get to Tyler Hall.

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

```
cd Documents/
```

We can use a the relative path to describe where we want to change to. The `Documents/` is a relative path because it does not start with `/`

To see what changed, we use `pwd` again

```
pwd
```

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

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

I moved one step further into my inclass folder

```
cd inclass/
```

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

```
mkdir systems
```

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

3.3. Your Home directory is easy to get to

Next we will use `cd` without path at all.

```
cd
```

Here we noted that the prompt changed back to what we started with, the `~`. The home directory.

Skip to main content

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

Remember, you can use `tab` to completed. So I typed: `cd Doc` `tab` `in` `tab` `sys` `tab`

This saves time **and** reduces typos, the terminal is filling in based on what the available options are, what exists in your filesystem, it will not fill in with something that is not there.

3.4. Relative paths

Let's look at our current location again:

```
pwd
```

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

`..` is a special file that points to a specific relative path, of one level up.
We can use `..` to go up one level from wherever we are.

```
cd ..
```

we can see the impact using `pwd`

```
pwd
```

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

Notice that before the last part was systems and the new path is missing that last part.

We will go back to the folder we made, since that is where we want to work for class.

```
cd systems/
```

3.5. A toy repo for in class

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

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

Skip to main content

only work in this repo during class time or making up class, unless specifically instructed to (will happen once in a few weeks)

After you have the repo, run the one action that is there.

Then find your readme issue and note its number, and copy the template for the README.

From the code tab, create a new file called README.md and paste in the template.

Commit the changes directly to main with `start readme closes #` and the number of your issue as the message, mine was `start readme closes #3`.

This will close the issue.

3.6. Connecting with GitHub

We have two choices to Download a repository:

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

we want option 1 beacuse we are learning git

3.6.1. Authenticating with GitHub

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

1. ssh keys (we will do this later)
2. `gh` CLI / gitscm in GitBash through browser

3.6.2. Windows (gitbash)

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

3.6.3. MacOS X

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

3.6.4. If nothing else works

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

Skip to main content

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

3.6.5. Cloning a repo

```
git clone https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm.git
```

then we get several messages back from git and GitHub (the remote, it *could* be a different host and a repo can have multiple remotes)

```
Cloning into 'gh-inclass-sp24-brownsarahm'...
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.
```

We can see that it made a new folder by looking at what is in our folder.

We can view what is in a folder with `ls` for **list**

```
ls
```

```
gh-inclass-sp24-brownsarahm
```

It was empty and now it has your repo!

3.7. What is in a repo?

We can enter that folder

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

When we compare the local directory to GitHub

```
ls
```

```
README.md
```

Notice that the `.github/workflows` that we see on GitHub is missing, that is because it is *hidden*. All file names that start with `.` are hidden. Hidden files are not protected, they are just lightly hidden to protect a casual user from accidentally editing them. As a developer, you will likely edit hidden files a lot.

In this case, the hidden files are **for** GitHub's server, not for local use.

Skip to main content

We can actually see the rest with the `-a` for **all** option or *flag*. Options are how we can modify how a command line program works, mostly they are optional or nonrequired.

```
ls -a
```

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

Now we see the `.github` folder like we saw on GitHub.

There is also the `.git` directory. This folder is for use by the git program. Most simple programs we write in school, run, store all of their values in variables in memory and thn when you restart you make new values. git, as a program uses that `.git` directory to store all of the information it needs in files.

We also see some special “files” that we will always see in every location:

- `.` the current location
- `..` up one directory

3.8. How do I know what git knows?

`git status` is your friend.

Let’s see how it works:

```
git status
```

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

nothing to commit, working tree clean
```

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

this tells us:

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

3.9. Making a branch with GitHub.

On your about issue on Github.com, create a branch using the link in the development section of the right side panel. See the github docs for how to do that.

Skip to main content

ach do.

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

```
git fetch origin
```

```
From https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm
 * [new branch]      2-create-an-about-file -> origin/2-create-an-about-file
```

then we check status again

```
git status
```

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

nothing to commit, working tree clean
```

Looks like nothing so far.

Next, we switch to that branch.

```
git checkout 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
```

Now it shows us the new branch!

3.10. Creating a file on the terminal

The `touch` command creates an empty file

```
touch about.md
```

```
. . .
```

Skip to main content

```
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.

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

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

```
nano about.md
```

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

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

and we can look at the contents of it.

Now we will check again with git.

```
git status
```

```
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)
```

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

Skip to main content

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

```
git add .
```

And again, we will check in with git

```
git status
```

```
On branch 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 about file close s #2'
```

```
[2-create-an-about-file 81c6f18] create about file close s #2
 1 file changed, 2 insertions(+)
 create mode 100644 about.md
```

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

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
```

Skip to main content

```
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

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

We can send them to github with `git push`

```
git push
```

```
Enumerating objects: 4, done.
Counting objects: 100% (4/4), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 337 bytes | 337.00 KiB/s, done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm.git
   faef6af..81c6f18  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

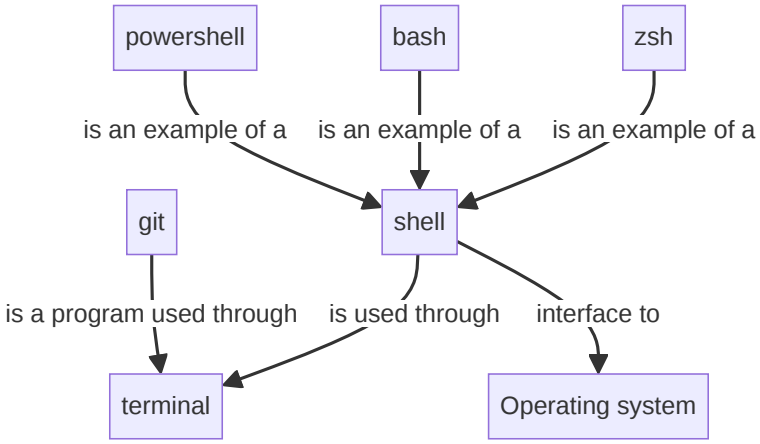
3.11. Concept Overview

Note

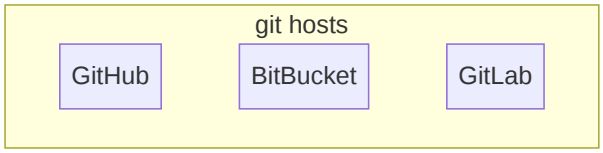
In the notes from last class I described what this type of diagram is called and how to read it.

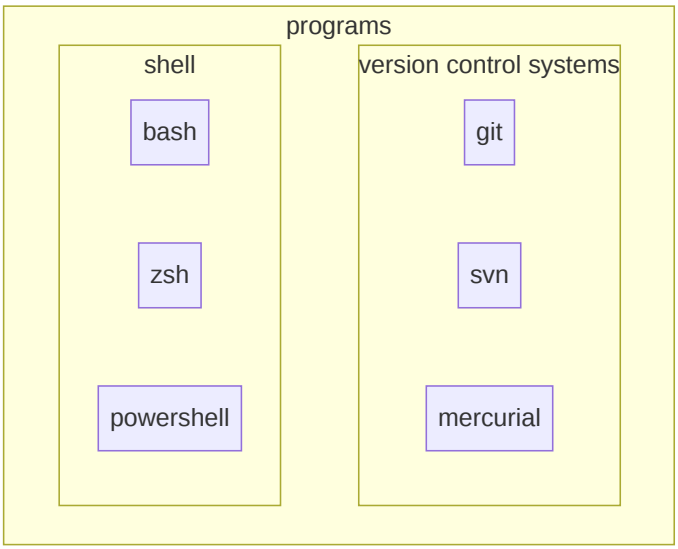
Important

This is not the only thing that would be correct for the prep work, in the next prepare, you can see if yours *conceptually* matches this even if visualized differntly.



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





Today's bash commands:

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

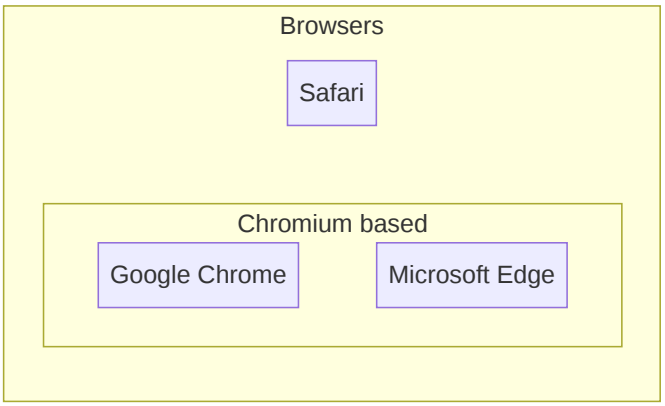
Skip to main content

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

3.12. Prepare for Next Class

1. Find the glossary page for the course website, link it below. Review the terms for the next class: shell, terminal, bash, git, zsh, powershell, GitHub. Make a diagram using mermaid to highlight how these terms relate to one another
2. Check your kwl repo before class and see if you have recieved feedback, reply or merge accordingly.

Example “venn diagram ” with mermaid subgraphs



3.13. Badges

Review | **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 comment. Be sure it is not inside another repository.
3. Try using setting up git using your favorite IDE’s git integration (not its terminal) or GitHub Desktop. Make a file gitgui.md and include some notes of how it went. Give the file a heading like `# Setting up <tool name>`, with the actual tool name you setup in the title (eg Github Desktop or VSCode source control panel or …) Was it hard? easy? what did you figure out or get stuck on? Is the terminology consistent to the terminal or does it use different terms?

Skip to main content

3.14. Experience Report Evidence

If you missed class today, link to your gh-inclass repo in the Experience report PR.

3.15. Questions After Today's Class

3.15.1. can you link an issue to both commits and PRs, or only PRs

both! we did both in today's class. We closed an issue via commit for the README file and then make a PR with a linked issue for the about.md file.

3.15.2. if we want to stage multiple files, can we do that in one command, like git add about.md README.md

Yes! or you can use `.`

3.15.3. do we do anything with the pull request we just made in the in-class repo?

Leave the about PR open for class on Thursday

3.15.4. Is mermaid something we will be using just for now, or should I become more familiar with it?

It is useful and we will use it on an off throughout class. Since GitHub supports it in all markdown, including comments, it is handy to be able to put in a quick diagram to explain something.

If you like it, it is something you can use even more, for a possible explore badge.

3.15.5. What are the benefits of using GitHub locally

Using git and GitHub locally means you can use your regular devtools and use GitHub as a could copy of your work.

4. How do branches work?

4.1. Review

> **Note**
>
> This part of the notes is the key facts of the questions we did, not the questions, you can reivew the actual questions

Skip to main content

- `cd` with no path changes to home (`~`)
- absolute paths start with `/` or `C:\` or similar on Windows outside of bash
- absolute paths always work, like an address; relative paths are starting point specific, like directions
- `.` is a special file that references the current location, for example, we can stage all of the files in the working directory with `git add .`

4.2. Back to the gh-inclass repo

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

First we will navigate to our working directory for class that we made in the last class

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

```
pwd
```

```
/Users/brownsarahm
```

Now that we know where we are, we can move to the target location

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

and look at the location again to confirm it is what we expected

```
pwd
```

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

Tip

In class, I demod the fact about `cd` without a path going to home and then used my built in history to repeat a command by using the `&#8593;` key

Then we move into the gh inclass repo

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

and confirm the path

```
pwd
```

Skip to main content

4.3. Branches do not sync automatically

First, we will look at where we left off to refresh, then we will make a change in GitHub and see how that impacts our local copy.

We start examining our local copy by looking at the working directory

```
ls
```

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

we see both files like we expected.

```
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
```

and everything is logged in git.

4.4. Getting to GitHub from your local system

> **Warning**
>
> This requires that you have the `gh` CLI working
>
> It is finicky on GitBash, but if it does not work you can navigate in your browser.
>
> If yours did not work, but then you figure out a fix to make it work, submit that as an FAQ on the git and github faq page for a community badge.

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

```
Opening github.com/compsys-progtools/gh-inclass-sp24-brownsarahm in your browser.
```

In the browser, we merged the PR we opened on Tuesday, that updates the main branch on GitHub with 2 more commits:

- the commit that was on the `2-create-an-about-file`
- the merge commit

Skip to main content

```
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
```

merging on GitHub does not change anythign locally

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

```
git cehkout main
```

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

The most similar command is
        checkout
```

Now, I spell it correctly and actually switch

```
git checkout main
```

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

It says we are up to date with `origin/main` and remember `origin` is the *name* of the remote that is where we cloned from on GitHub.com.

Now, we look at the working directory.

```
ls
```

```
README.md
```

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

We can also confirm that it is not hidden:

```
ls -a
```

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

Skip to main content

4.5. Updating Locally

Updating locally involves 2 steps:

- update the local repo (the .git directory with all of the commits)
- update the working directory

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

```
git fetch
```

```
remote: Enumerating objects: 1, done.
remote: Counting objects: 100% (1/1), done.
remote: Total 1 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (1/1), 919 bytes | 919.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm
   faef6af..1e2a45f  main       -> origin/main
```

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

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

Now, again, we check the working directory

```
ls
```

```
README.md
```

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

However, we can see the impact of the `fetch` using `git status`

```
git status
```

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

nothing to commit, working tree clean
```

Now, git knows that our local main is behind `origin/main` and by how much and tells us how to apply the changes to the local branch.

Skip to main content

```
git pull
```

```
Updating faef6af..1e2a45f
Fast-forward
 about.md | 2 ++
 1 file changed, 2 insertions(+)
 create mode 100644 about.md
```

It applies the changes to the local directory too:

```
ls
```

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

4.6. What other tools do I have to examine things on the terminal

Lets try two more ways of looking at our repo and files. Then we will use those to learn more about working with branches.

4.6.1. Git log

We can see commits with `git log`

```
git log
```

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

```
commit 1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4 (HEAD -> main, origin/main, origin/HEAD)
Merge: faef6af 81c6f18
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 1 12:51:17 2024 -0500

    Merge pull request #4 from compsys-progtools/2-create-an-about-file

    create about file close s #2

commit 81c6f187f146caaaf43d97bc1bb8ed237142f4c3 (origin/2-create-an-about-file, 2-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Jan 30 13:33:54 2024 -0500

    create about file close s #2

commit faef6af98e6bd0951ebcefb809ff4e353a0c7fbc
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Jan 30 13:03:36 2024 -0500

    start readme, closes #3

commit 98cff657d25adf9a0820a04d60f6409445f70e76
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Jan 30 17:53:54 2024 +0000
```

Skip to main content

4.6.2. Concatenate a file

the `cat` command *concatenates a files' contents to the terminal output (which is actually a special file called standard out or stdout)

```
cat about.md
```

```
I like to ski
```

4.7. Making a branch locally

> **Important**
>
> It is totally okay to test ideas out and see what happens as a way to investigate how things work. Here we tested a few suggestions to see what works. This lets us learn 2 things:
>
> - what works
> - how to read error messages

4.7.1. Checkout requires the branch to exist

```
git checkout my_branchcehcekd
```

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

This error message says that it cannot switch because the name we gave does not already exist. This is a good to recognize error message because this is also what would happen if you tried to switch, but spelled the branch name wrong.

4.7.2. the -b option lets checkout create a branch

```
git checkout -b my_branch_cehckoutb
```

```
Switched to a new branch 'my_branch_cehckoutb'
```

Success!

the `-b` lets `git checkout` both create and switch to a branch

Skip to main content

4.7.3. create is not an option

```
git branch create example
```

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

This error message is tricky; it only complains about `example`. We could try

```
git branch create
```

this gives no response, but if we use `git branch` we would see it created a branch called `create`.

So, back to our original attempt, is that it thinks we are asking to make a branch named `create` but then sees this extra thing `example` that it doesn't know what to do with, so blames that for the error.

4.7.4. Chaining commands together

Finally, we tried:

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

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

This worked. From this example, we learn a 2 things:

- `git branch <new name>` creates a new branch
- we can put multiple commands on a "single line" using `;` between them

4.7.5. We can list branches

without any argument, we get a list of the branches that exist`

```
git branch
```

```
  2-create-an-about-file
  main
* my_branch
  my_branch_cehckoutb
```

note it also indicates which branch you are on

Skip to main content

4.8. Branches are pointers

We'll go back to main

```
git checkout main
```

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

and then make a new branch for the next changes we will make.

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

```
git checkout -b fun_fact
```

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

If we use `git log` to see the commits, we can see more about the branches.

```
git log
```

```
commit 1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4 (HEAD -> fun_fact, origin/main, origin/HEAD, my_branch_ce
Merge: faef6af 81c6f18
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 1 12:51:17 2024 -0500

    Merge pull request #4 from compsys-progtools/2-create-an-about-file

    create about file close s #2

commit 81c6f187f146caaaf43d97bc1bb8ed237142f4c3 (origin/2-create-an-about-file, 2-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Jan 30 13:33:54 2024 -0500

    create about file close s #2

commit faef6af98e6bd0951ebcefb809ff4e353a0c7fbc
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Jan 30 13:03:36 2024 -0500

    start readme, closes #3

commit 98cff657d25adf9a0820a04d60f6409445f70e76
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Jan 30 17:53:54 2024 +0000

    Initial commit
```

branches are pointers a branch points to a commit.

4.9. Linking a locally created branch to a remote

Next we will edit the file so we can create a commit.

```
nano about.md
```

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

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.

```
cat about.md
```

```
I like to ski
I came to URI in 2020
```

My file now has 2 lines in it.

Now we check the status

```
git status
```

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

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

Now the file is *modified* instead of what we saw before that was *untracked*

Again we stage it first:

```
git add about.md
```

then look again

```
git status
```

```
On branch fun_fact
Changes to be committed:
```

Skip to main content

```
  (use "git restore --staged <file>..." to unstage)
        modified:   about.md
```

and commit.

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

```
[fun_fact 07897fd] add a fun fact
 1 file changed, 1 insertion(+), 1 deletion(-)
```

Then we push

```
git push
```

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

    git push --set-upstream origin fun_fact

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

but it fails.

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.

We follow git’s advice

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

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (2/2), done.
Writing objects: 100% (3/3), 287 bytes | 287.00 KiB/s, done.
Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), completed with 1 local object.
remote:
remote: Create a pull request for 'fun_fact' on GitHub by visiting:
remote:      https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm/pull/new/fun_fact
remote:
To https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm.git
 * [new branch]      fun_fact -> fun_fact
branch 'fun_fact' set up to track 'origin/fun_fact'.
```

and success!

4.10. Merge conflicts

Skip to main content

This means we are learning two things:

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

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

To create the merge conflict we are going to edit the same branch in two different ways locally and in browser. This couls also happen by two different people making edits to the same branch *or* more commonly, by having edits made on both the compare branch and the base branch of a PR to the same part of the same file.

4.10.1. Edit in browser

First, in your browser edit the about.md file to have a second fun fact. Commit directly to the `fun_fact` branch.

4.10.2. Edit locally

Then edit it locally to also have 2 fun facts.

```
nano about.md
```

My local version looks like this:

```
cat about.md
```

```
I like to ski
I came to URI in 2020
I moved to RI in 2018
```

Now we check with git

```
git status
```

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

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

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

and stage and commit

```
git add .
git commit -m 'another funfact'
```

```
[fun_fact 7985615] another funfact
 1 file changed, 1 insertion(+)
```

4.10.3. Getting a conflict

Now if we try to pull the changes from GitHub.

```
git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 938 bytes | 312.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm
   07897fd..7e78493  fun_fact   -> origin/fun_fact
hint: You have divergent branches and need to specify how to reconcile them.
hint: You can do so by running one of the following commands sometime before
hint: your next pull:
hint:
hint:   git config pull.rebase false  # merge
hint:   git config pull.rebase true   # rebase
hint:   git config pull.ff only       # fast-forward only
hint:
hint: You can replace "git config" with "git config --global" to set a default
hint: preference for all repositories. You can also pass --rebase, --no-rebase,
hint: or --ff-only on the command line to override the configured default per
hint: invocation.
fatal: Need to specify how to reconcile divergent branches.
```

Now it cannot work because the branches are divergent. This illustrates the fact that our two versions of the branch `fun_fact` and `origin/fun_fact` are two separate things.

diverged means like this diagram:

```
gitGraph
    commit id:"A"
    commit id:"B"
    branch fun_fact
    checkout fun_fact
    commit id:"C"
    branch origin/fun_fact
    checkout origin/fun_fact
    commit id:"D"
    checkout fun_fact
    commit id:"E"
```

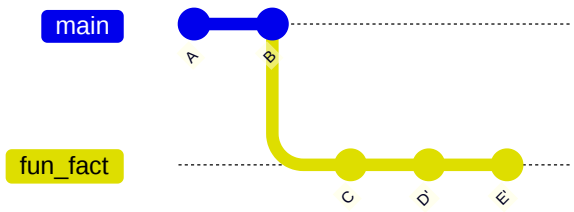
Note that the `origin/fun_fact` and `fun_fact` have different histories.

the branches have diverged means that they do not agree and that they each have at least one commit that is different from

diverged does not necessarily mean a conflict, sometimes we can simply change what commit has what parent and it will work.

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

Rebase will try to make it like this:



the commits `D` and `E` get modified some, the time and parent info is updated, but the changes to the content are the same.

```
git pull --rebase
```

```
Auto-merging about.md
CONFLICT (content): Merge conflict in about.md
error: could not apply 7985615... another funfact
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 7985615... another funfact
```

It was not able to do it; beacuse both had edited the same line of the same file. This requires a person to figure out what the content should be.

it tells us what file the issue is in the 2nd line.

4.10.4. Resolving a merge conflict

To resolve manually, we have to edit the file. We are going to use nano again here.

```
nano about.md
```

The file looks like this when we open it up:

```
I like to ski
I came to URI in 2020
<<<<<<< HEAD
I am from NH
=======
I moved to RI in 2018
>>>>>>> 7985615
```

git added markup to the file to show us what was in each of the two versions, labeled by the HEAD pointer and the specific commit for the other. It puts `=` between the two. To resolve, we edit the file to be what we want. We can choose one version,

Skip to main content

```
cat about.md
```

```
I like to ski
I came to URI in 2020
I am from NH
I moved to RI in 2018
```

Next, we go back to check in with git.

```
git status
```

```
interactive rebase in progress; onto 7e78493
Last command done (1 command done):
   pick 7985615 another funfact
No commands remaining.
You are currently rebasing branch 'fun_fact' on '7e78493'.
  (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")
```

it tells us how to finish up what we were doing.

First we commit:

```
git add .
git commit -m 'resolve conflit'
```

```
[detached HEAD 07959c0] resolve conflit
 1 file changed, 3 insertions(+), 1 deletion(-)
```

then use the `continue` option

```
git rebase --continue
```

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

Finally we check in:

```
git status
```

```
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

and push

```
git push
```

```
Enumerating objects: 5, done.
Counting objects: 100% (5/5), done.
Delta compression using up to 8 threads
Compressing objects: 100% (3/3), done.
Writing objects: 100% (3/3), 314 bytes | 314.00 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.
To https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm.git
   7e78493..07959c0  fun_fact -> fun_fact
```

4.11. Prepare for Next Class

1. Add file called terminal-vocab.md on a branch linked to this issue. Fill in the template below:

```
# Terminal vocab

## Before

<insert your diagram from 2024-01-30 prepare>

## Reflection

<!-- write a few bullets assessing your diagram above based on the notes from 2024-01-30. was anything ir


## Final thoughts

<!-- write a few bullets/sentences on how trying to make the diagram did/not help you think about the ter
```

4.12. Badges

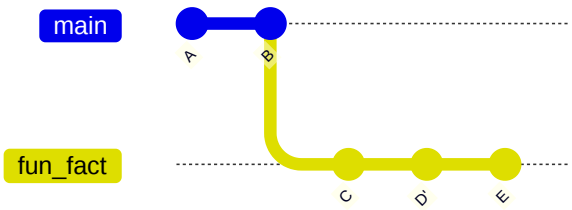
Review **Practice**

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

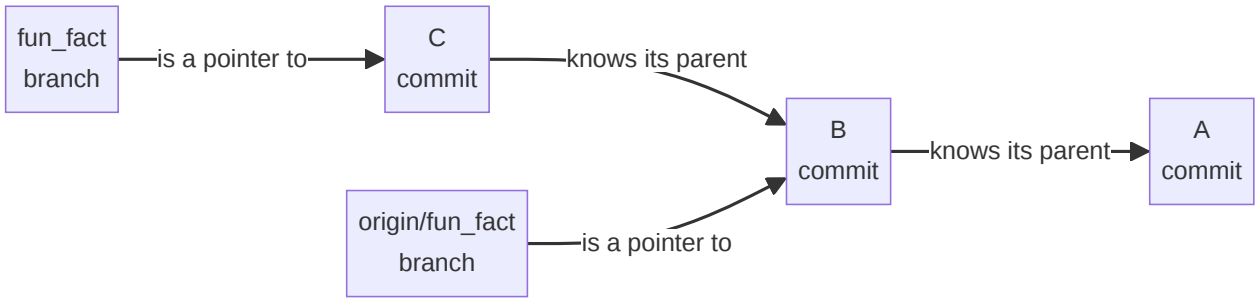
4.13. Summary

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

We often visualize git using graphs like subway maps:



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



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

4.13.1. New bash commands

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

Skip to main content

4.13.2. New git commands

command	explanation
`git log`	show list of commit history
`git branch`	list branches in the repo
`git branch new_name`	create a `new_name` branch

command	explanation
`git checkout -b new_Name`	create a `new_name` branch and switch to it
`git pull`	apply or fetch and apply changes from a remote branch to a local branch

4.14. Experience Report Evidence

run the command:

```
git log >> makeup-2024-02-02.md
```

then `mv makeup-2024-02-02.md <rel>` where `<rel>` is the relative path from your gh incalss repo to your KWL repo.

commit this file to the experience report branch

4.15. Questions After Today's Class

5. When do git and bash give me an advantage?

so far we have used git and bash to accomplish familiar goals, and git and bash feel like just extra work for familiar goals.

Today, we will start to see why git and bash are essential skills: they give you efficiency gains and time traveling super powers (within your work, only, sorry)

5.1. Setting the stage

Discuss your prepare work with your neighbors. Share what project you looked at and discuss the following. If you all looked at the same repo, split up and look at different ones now.

- What things were common across different repos?
- What was different?

5.2. Important references

Use these for checking facts and resources.

- bash
- git

Skip to main content

5.3. Setup

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

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

Find your PR that I opened for you today that has the title, "2/6 in class activity"

I am going to use the `gh` CLI for this. It might not work on Windows unless you got gh and GitBash configured correctly.

```
gh pr list
```

```
Showing 1 of 1 open pull request in compsys-progtools/gh-inclass-sp24-brownsarahm

#5  2/6 in class activity  organizing_ac  about 1 hour ago
```

It shows me the number of total PRs and then for each one:

- the PR number (remember there is one sequence of numbers for issues and PRs)
- PR title
- branch name
- when it was created

Mine was 5, so I can view it with this command:

```
gh pr view 5
```

```
2/6 in class activity #5
Draft • brownsarahm wants to merge 1 commit into main from organizing_ac • about 1 hour ago
+17 -0 • No checks


  this draft PR https://github.blog/2019-02-14-introducing-draft-pull-
  requests/ adds some example files that we will use during class
  on 2/6.  \n\n wait until class time (or when you make up class
  by following the notes) to do anything with this PR


View this pull request on GitHub: https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm/pull/5
```

by default it shows the contents of the PR message and then the link to view in browser

Alternatively, we can use the `--web` option to open it in browser.

```
gh pr view 5 --web
```

Skip to main content

Then we merged in the PR after converting it from a draft PR to a regular PR using the “Ready for Review” button.

Then back locally, we check where we are

```
git status
```

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

nothing to commit, working tree clean
```

5.4. Organizing a project (workign with files)

A common question is about how to organize projects. While our main focus in this class session is the `bash` commands to do it, the *task* that we are going to do is to organize a hypothetical python project

Put another way, we are using organizing a project as the *context* to motivate practicing with bash commands for moving files.

A different the instructor might go through a slide deck that lists commands and describes what each one does and then have examples at the end. Instead, we are going to focus on organizing files, and I will introduce the commands we need along the ways.

We’ll go back to main first

```
git checkout main
```

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

Then we will get the update from main.

```
git pull
```

```
remote: Enumerating objects: 20, done.
remote: Counting objects: 100% (20/20), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 19 (delta 1), reused 17 (delta 0), pack-reused 0
Unpacking objects: 100% (19/19), 2.46 KiB | 114.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm
   1e2a45f..9f39946  main          -> origin/main
 * [new branch]      organizing_ac -> origin/organizing_ac
Updating 1e2a45f..9f39946
Fast-forward
 API.md                | 1 +
 CONTRIBUTING.md       | 1 +
 LICENSE.md            | 1 +
 _config.yml           | 1 +
 _toc.yml              | 1 +
 abstract_base_class.py | 1 +
 alternative_classes.py | 1 +
```

Skip to main content

```
 important_classes.py   | 1 +
 philosophy.md          | 1 +
 scratch.ipynb          | 1 +
 setup.py               | 1 +
 test_abc.py            | 1 +
 tests_alt.py           | 1 +
 tests_helpers.py       | 1 +
 tests_imp.py           | 1 +
 17 files changed, 17 insertions(+)
 create mode 100644 API.md
 create mode 100644 CONTRIBUTING.md
 create mode 100644 LICENSE.md
 create mode 100644 _config.yml
 create mode 100644 _toc.yml
 create mode 100644 abstract_base_class.py
 create mode 100644 alternative_classes.py
 create mode 100644 example.md
 create mode 100644 helper_functions.py
 create mode 100644 important_classes.py
 create mode 100644 philosophy.md
 create mode 100644 scratch.ipynb
 create mode 100644 setup.py
 create mode 100644 test_abc.py
 create mode 100644 tests_alt.py
 create mode 100644 tests_helpers.py
 create mode 100644 tests_imp.py
```

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

I gave a bunch of files, each with a short phrase in them.

- none of these are functional files
- the phrases mean you can inspect them on the terminal

We can view the contents of a file with `cat`.

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

```
cat setup.py
```

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

Important

file extensions are for people; they do not specify how the file is actually written . these are all *actually* plain text files, despite different file exensions.

Before we start organizing, we will make a new branch.

```
git checkout -b organization
```

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

Skip to main content

```
git status
```

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

5.5. Files, Redirects, git restore

Let's first look at the README file

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
```

Echo allows us to send a message to stdout.

```
echo "hello"
```

```
hello
```

Now we are going to put these two things together, but first let's connect what we are about to learn to something you have probably seen before.

Think about a time you opened a file within a program that you wrote. For example

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

in both cases one parameter is the file to open, what other parameters have you used?

Typically we also have to tell it what *mode* to open the file with, and some options are:

- read
- write
- append

We can also **redirect** the contents of a command from stdout to a file in `bash`. Like file operations while programming there is a similar concept to this mode.

There are two types of redirects, like there are two ways to write to a file, more generally:

- write (`>`)
- append (`>>`)

Skip to main content

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

This command does not return anything, because instead of writing to stdout, it wrote to the file.

We can see the impact by looking at the README file.

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
hello
```

We can redirect other commands too:

```
git status >> curgit
```

we see this created a new file

```
ls
```

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

and we can look at its contents too

```
cat curgit
```

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

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

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

This also highlights a bit more about how the redirect works.

Skip to main content

of to stdout.

Fu

Ref

-
-

C is
the
"offi

Important

Redirect docs

Note this also explains that `<` is for redirecting inputs (stdin is *also* a file like stdout and has input, like your keyboard)

this is not a file we actually want, which gives us a chance to learn another new bash command: `rm` for remove

```
rm curgit
```

Important

Note that this is a true, full, and complete DELETE, this does not put the file in your recycling bin or the apple trash can that you can recover the file from, it is **gone** for real.

We will see soon a way around this, because git can help.

use `rm` with great care

```
git status
```

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

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

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

When we want to commit all files that are changed or deleted we can use the `-a` option. This does not do anything to untracked files. Thi option is short for all, and can also be used like `--all`, so you cannot pick a subset of files out with this option.

```
git commit -a -m 'add to readme'
```

```
[organization 29ffc88] add to readme
 1 file changed, 1 insertion(+)
```

and we confirm that it is what we want

```
git status
```

Skip to main content

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

clean working area, ready to keep going

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 "hello" > README.md
```

Now we look at the file:

```
cat README.md
```

```
hello
```

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

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

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

We can also use `git status` for a hint.

```
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 working directory.

Let's follow the hint

```
git restore README.md
```

First we will check with git

```
git status
```

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

back to a clean working tree is a good sign!

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

```
cat README.md
```

```
# GitHub Practice

Name: Sarah Brown
hello
```

back how we wanted it!

5.6. What if I forget the `-m`

Now we will add some text to the README so we have a single point of reference about what these files are “about”

```
echo "|file | contents |
> > | ------| ------- |
> > | abstract_base_class.py | core abstract classes for the project |
> > | helper_functions.py | utitly funtions that are called by many classes |
> > | important_classes.py | classes that inherit from the abc |
> > | alternative_classes.py | classes that inherit from the abc |
> > | LICENSE.md | the info on how the code can be reused|
> > | CONTRIBUTING.md | instructions for how people can contribute to the project|
> > | setup.py | file with function with instructions for pip |
> > | test_abc.py | tests for constructors and methods in abstract_base_class.py|
> > | tests_helpers.py | tests for constructors and methods in helper_functions.py|
> > | tests_imp.py | tests for constructors and methods in important_classes.py|
> > | tests_alt.py | tests for constructors and methods in alternative_classes.py|
> > | API.md | jupyterbook file to generate api documentation |
> > | _config.yml | jupyterbook config for documentation |
> > | _toc.yml | jupyter book toc file for documentation |
> > | philosophy.md | overview of how the code is organized for docs |
> > | example.md | myst notebook example of using the code |
> > | scratch.ipynb | jupyter notebook from dev |" >> README.md
```

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

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

We also learn something about bash: using the open quote `"` then you stay inside that until you close it. when you press enter the command does not run until after you close the quotes

Skip to main content

```
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")
```

This time we will add separately

```
git add .
```

and commit without using `-m`

```
git commit
```

This brings us into vim. vim is a ubiquitous, but not novice friendly terminal text editor.

What is is doing is making a temporary file for you, opening it in the vim editor and waiting for you to write a commit message.

Important

What you need to know about vim:

- it has 2 modes, command and edit
- in command mode pressing keys does not enter text, but does commands
- in edit mode, it says `~insert~` at the bottom
- to get into edit mode press `a`
- to get into command mode press `escape`
- to exit type `:wq` and press `enter` or `return`

Once you edit your temp file so that there is a line without a `#` that has your commit message and exit, your commit gets created.

```
[organization 260c9c3] describe the files
 1 file changed, 19 insertions(+)
```

5.7. Moving files and patterns

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

```
mkdir docs
```

Skip to main content

```
ls
```

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

next we will move a file there with `mv`

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

what this does is change the path of the file from `.../gh-inclass-sp24-brownsarahm/philosophy.md` to `.../gh-inclass-sp24-brownsarahm/docs/philosophy.md`

Important

moving a file means changing its path; it does not rewrite the file.

While it is useful to think of a path like an address, a path is not tied to a specific memory location on the disk. The operating system keeps an index of the human readable paths that relates them to atual memory locations on the storage device.

`mv` updates that index, not the actual storage device.

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

```
ls
```

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

the `philosophy.md` file is no longer visible in the current working directory.

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

```
```

```
philosophy.md
```

5.7.1. Move can also rename files

We have one file with `test_` and a bunch of others with `tests_`. We want them all to follow the same pattern, so let's rename.

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 test_abc.py tests_abc.py
```

This changes the path from `.../test_abc.py` to `.../tests_abc.py` to. It is doing the same thing as when we use it to move a file from one folder to another folder, but changing a different part of the path.

```
ls
```

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

5.7.2. Moving multiple files with patterns

let's look at the list of files again.

```
ls
```

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

We have a lot of `tests_` files, we will put them all in one folder. First we create the folder:

```
```

We can use the `*` wildcard operator to move all files that match the pattern.

```
mv tests_* tests/
```

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

```
ls
```

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

and that they *are* in `tests`

```
ls tests/
```

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

success!

We will do the same thing for the `.yml` files to move them to the docs folder

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

Next we are going to move the Python files that are for the “package” that we have made. We will make a folder for this too.

```
mkdir pkg
```

Here we want to move all of the `.py` files that have a `_` in the name, because the only one we do not want to move is `setup.py`

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

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

I made an error here so it gave me usage for the command.

`mv` requires both a source and target, and I did not provide one.

Skip to main content

```
mv *_*.py pkg/
```

we see that it works.

> **Warning**
>
> Some students got a different error message, instead of being told they had used the command wrong, their error message said that `important_classes.py` is not a directory. That tells me that what happened was the `*_*.py` was expanded to a list of files and then `mv` tried to work on the list of files using all but the last one as sources and tried to use the last one as the destination.
>
> To debug, I tried comparing bash versions, but it appears that we had the same version, so why we saw different behavior is TBD.
>
> **this is worth a community badge if you can explain why**

```
bash --version
```

```
GNU bash, version 3.2.57(1)-release (x86_64-apple-darwin20)
Copyright (C) 2007 Free Software Foundation, Inc.
```

5.8. Hidden files

We are going to make a special hidden file and an extra one. We will use the following command:

```
touc .secret .gitignore
```

```
bash: touc: command not found
```

I spelled `touch` wrong so `bash` told me the command was not found.

```
touch .secret .gitignore
```

We also learn 2 things about `touch` and `bash`:

- `touch` can make multiple files at a time
- lists in `bash` are separated by spaces and do not require brackets

We can see if it worked:

```
ls
```

Skip to main content

```
API.md          README.md       example.md      setup.py
CONTRIBUTING.md about.md        pkg             tests
LICENSE.md      docs            scratch.ipynb
```

but then notice that we cannot see hidden files as is

using the `-a` option we can:

```
ls -a
```

```
.               .secret         about.md        setup.py
..              API.md          docs            tests
.git            CONTRIBUTING.md example.md
.github         LICENSE.md      pkg
.gitignore      README.md       scratch.ipynb
```

lets put some content in the secret (it will actually *not* be going to GitHub)

```
echo "my dev secret" >> .secret
```

Now we can check with git

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   tests_abc.py

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    abstract_base_class.py
        deleted:    alternative_classes.py
        deleted:    helper_functions.py
        deleted:    important_classes.py
        deleted:    philosophy.md
        deleted:    test_abc.py
        deleted:    tests_abc.py
        deleted:    tests_alt.py
        deleted:    tests_helpers.py
        deleted:    tests_imp.py

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

we have lots of changes, including he secret file. it is not tracked yet, but git is still telling us that it sees it. We want git to

gitignore lets us *not* track certain files

let's ignore that `.secret` file

```
echo ".secret" >> .gitignore
```

Next we check in with git again

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   tests_abc.py

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    abstract_base_class.py
        deleted:    alternative_classes.py
        deleted:    helper_functions.py
        deleted:    important_classes.py
        deleted:    philosophy.md
        deleted:    test_abc.py
        deleted:    tests_abc.py
        deleted:    tests_alt.py
        deleted:    tests_helpers.py
        deleted:    tests_imp.py

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

and the `.secret` file is no longer there! git is not checking on that file at all.

5.8.1. Gitignore works with patterns too

First, I'll make a folder

```
mkdir my_secrets
```

then go into it

```
cd my_secrets/
```

and make a bunch of files

Skip to main content

```
touch a b c d e f
```

```

```

```
git status
```

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

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    ../_config.yml
        deleted:    ../_toc.yml
        deleted:    ../abstract_base_class.py
        deleted:    ../alternative_classes.py
        deleted:    ../helper_functions.py
        deleted:    ../important_classes.py
        deleted:    ../philosophy.md
        deleted:    ../test_abc.py
        deleted:    ../tests_abc.py
        deleted:    ../tests_alt.py
        deleted:    ../tests_helpers.py
        deleted:    ../tests_imp.py

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

Important

notice that the paths all start with `..` because we are in a *subdirectory* of the git repo, but not the top of it.

we will go back up

```
cd ..
```

and now ignore that folder we just made:

```
echo my_secrets/* >> .gitignore
```

next we check in with git

```
git status
```

Skip to main content

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   tests_abc.py

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    abstract_base_class.py
        deleted:    alternative_classes.py
        deleted:    helper_functions.py
        deleted:    important_classes.py
        deleted:    philosophy.md
        deleted:    test_abc.py
        deleted:    tests_abc.py
        deleted:    tests_alt.py
        deleted:    tests_helpers.py
        deleted:    tests_imp.py

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

It did not work because we did not put the text we were adding to `.gitignore` with echo in quotes. Let's fix it and then inspect futher what happened

```
echo "my_secrets/*" >> .gitignore
```

Now we check again

```
git status
```

```
On branch organization
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        new file:   tests_abc.py

Changes not staged for commit:
  (use "git add/rm <file>..." to update what will be committed)
  (use "git restore <file>..." to discard changes in working directory)
        deleted:    _config.yml
        deleted:    _toc.yml
        deleted:    abstract_base_class.py
        deleted:    alternative_classes.py
        deleted:    helper_functions.py
        deleted:    important_classes.py
        deleted:    philosophy.md
        deleted:    test_abc.py
        deleted:    tests_abc.py
        deleted:    tests_alt.py
        deleted:    tests_helpers.py
        deleted:    tests_imp.py

Untracked files:
```

Skip to main content

```
        .gitignore
        docs/
        pkg/
        tests/
```

It works!

To inspect, let's look at the `.gitignore` file

```
cat .gitignore
```

```
.secret
my_secrets/a my_secrets/b my_secrets/c my_secrets/d my_secrets/e my_secrets/f
my_secrets/*
```

We see that it has a list of all of the files that match the pattern `my_secrets/*` on the second line. That would match to *each* of those files, but not to the directory itself. In contrast, the last line matches both the files and the directory itself.

Finally, we will add and commit. We need to use `git add` directly because we have files that are untracked that we want to track.

```
git add .
```

```
git commit -m 'oranizizng'
```

```
[organization e3b192a] oranizizng
 12 files changed, 3 insertions(+)
 create mode 100644 .gitignore
 rename _config.yml => docs/_config.yml (100%)
 rename _toc.yml => docs/_toc.yml (100%)
 rename philosophy.md => docs/philosophy.md (100%)
 rename abstract_base_class.py => pkg/abstract_base_class.py (100%)
 rename alternative_classes.py => pkg/alternative_classes.py (100%)
 rename helper_functions.py => pkg/helper_functions.py (100%)
 rename important_classes.py => pkg/important_classes.py (100%)
 rename test_abc.py => tests/tests_abc.py (100%)
 rename tests_alt.py => tests/tests_alt.py (100%)
 rename tests_helpers.py => tests/tests_helpers.py (100%)
 rename tests_imp.py => tests/tests_imp.py (100%)
```

5.9. Summary

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

Skip to main content

5.9.1. New commands

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

5.9.2. New git commands

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

5.10. Prepare for next class

1. Bring git questions or scenarios you want to be able to solve to class on Thursday (in your mind or comment here if that helps you remember)
2. Practice using the new bash commands we have learned to read and understand the workflow files in your KWL repo. Look for bash commands that we have seen before. Try to modify files on a prepare branch so that your name is already filled in when your experience badge (inclass) action runs.

5.11. Badges

Review Practice

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on. In this case, in lab, we will check that you know what to do, but if we want you to do revisions those will be done through the badge.

1. Update your KWL chart with the new items and any learned items.
2. Clone the course website. Append the commands used and the contents of your `fall2023/.git/config` to a terminal_review.md (hint: history outputs recent commands and redirects can work with any command, not only echo). Edit the README.md, commit, and try to push the changes. What happens and what GitHub concept that we have not used yet might fix it? see your `vocab-` repo for a list of key github concepts. (answer in the `terminal_review.md`)
3. **lab** Organize the provided messy folder in a Codepsace (details will be provided in lab time). Commit and push the changes. Answer the questions below in your kwl repo in a file called `terminal_organization.md`
4. clone your `messy_repo` locally and append the `history.md` file to your `terminal_organization.md`

Skip to main content

5.12. Experience Report Evidence

redirect the history to a file

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

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

5.13. Questions After Today's Class

5.13.1. Why exactly does the mv command also work as the rename function?

There is also a `rename` utility, but `mv` can do both because "moving" a file is just changing its path.

5.13.2. Can you access the . folders not using terminal?

Yes, but I do not know the specifics for each operating system. Usually I think you can right click or control a setting somewhere.

5.13.3. Why do some open source projects have a lot of open issues, maybe even more issues than files?

Some open source projects get innundated with bug reports or questions. Some projects even use issues for other things. For example, the Carpentries trainers repo uses issues to keep track of proposals for running a leadership committee of a group of people. The journal of open source software uses the joss-reviews repo to do peer reviews, it has 12 files and 291 open issues + 6017 closed issues.

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

It fills in for any number of characters.

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

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

Skip to main content

Re-request a review

5.13.6. can you close pull requests on the terminal?

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

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

In the current working directory.

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

No those are basically the two purposes.

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

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

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

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

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

6. What *is* a commit?

6.1. Admin

Penalty free zone ends with today's badges

- all past badges can be submitted at any time
- all badges posted today going forward have deadlines per the syllabus
- There is a bonus for getting 6 badges completed (do not have to be approved) by 2/15.

6.2. Defining terms

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

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

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

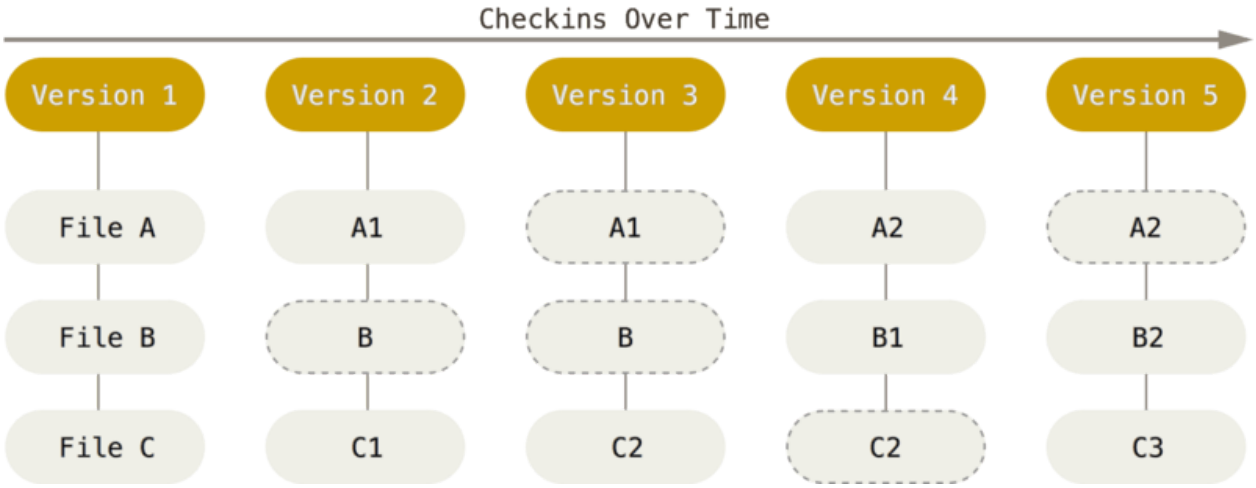
Now we need to contend with multiple definitions

These definitons could be based on

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

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

6.3. Conceptually, a commit is a snapshot



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

All other git things are defined relative to commits

- branches are pointers to commits
- tags are pointers to commits
- trees are how file path/organization information is stored for a commit
- blobs are how files contents are stored when a commit is made

6.5. Parts of a commit

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

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

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

Skip to main content

We can use `git log` to view past commits

```
git log
```

```
commit e3b192aa0cd490226e8adcd81d3d0b95adb5676b (HEAD -> organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:41:53 2024 -0500

    oranizizng

commit 260c9c309922970f80bfa2c93cc23bcfbb962740
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:06:20 2024 -0500

    describe the files

commit 29ffc88519103085ed3a2ab01cffb3c99d70fc6a
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:59:20 2024 -0500

    add to readme

commit 9f399466ad6a1ad572e104209f4469eb8cd48516 (origin/main, origin/HEAD, main)
Merge: 1e2a45f fca59e8
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:43:18 2024 -0500

    Merge pull request #5 from compsys-progtools/organizing_ac

    2/6 in class activity

commit fca59e8cca05bb0861f9348a40fe8300b3d55637 (origin/organizing_ac)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 11:21:21 2024 -0500

    add files for organizing

commit 1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4 (my_branch_cehckoutb, my_branch)
```

here we see some parts:

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

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

the hash is the *unique identifier* of each commit

we can view individual commits with `git cat-file` and at least 4 characters of the hash or enough to be unique. We will try 4 characters and I will use the last visible commit above (`1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4`)

`git cat-file` has different modes:

- `-p` for pretty print

Skip to main content

- `-t` to return the type

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

```
tree d5c7c4543f48ca7fb5cf75881fd0dbe21d75f51c
parent faef6af98e6bd0951ebcefb809ff4e353a0c7fbc
parent 81c6f187f146caaaf43d97bc1bb8ed237142f4c3
author Sarah Brown <brownsarahm@uri.edu> 1706809877 -0500
committer GitHub <noreply@github.com> 1706809877 -0500
gpgsig -----BEGIN PGP SIGNATURE-----

 wsFcBAABCAAQBQJlu9oVCRC1aQ7uu5UhlAAAxpQQAFxq5TBSKvb2Qoyilvayo80x
 9mY1stuFWwm2ymZhIGLJbJMbJ6ZAnlpfyAZHfQ/QQ7MnPdlt/GIAKKtIo+NLUkZj
 9aDQMk1vjkP1IEx1LjAjxoVHTvEKsDPWxKNlPLjyg9KqYH83LVtQYaMT788YNHaL
 Bcg7vgAbSmpDtn84HPmXGvgq7/V7kJyl/zADxDJaA+3ETHGehm7Fkuo6UdsbaPZH
 5is6eYsNwTrC/19vdnZcr9Gcfr6mNzttyiQDkiUZ1A77Eucrk4Di/W++JLjq/ufk
 Kl6M1T/ItpWQx3DmgoKTYcX5TsHQaS8ONCaGDDHimL6P+f4os7dHboYU6aDFipgt
 WyrLKnCRCIrVzEwHJvWCGHiUW50Ho3yOpwi/1N24U8mXBckKGP59Tgtv+Ef6Zik1
 Aj+qFzRM1d0/A/c3MYBbmBtdcdlS/dVwLu9uqGq+DjSrPoi8T292YVC6YduQAe1i
 Effj9V6sGtyZv83ZkCCig1SKfbt4zoW1P1zK14XVNdcIRrDwwyKXfqKUUl4k9gZV
 1yjMHVEaljA4Ey3qrn4rwh5teaZpAsPrj/5FF6roUeNUoD7iXXq3MHDz619otALB
 o9Xe4x/Zn167j9M0R78Gid6qOQxQx+LgtOjqidwt1HjlOgLUEgSDloFpxhgH56Ec
 TVSXjxLMrPrRlf+4ZaAZ
 =0h+s
 -----END PGP SIGNATURE-----


Merge pull request #4 from compsys-progtools/2-create-an-about-file

create about file close s #2(base) brownsarahm@gh-inclass-sp24-brown7
040000 tree 95b60ce8cdec1bc4e1df1416e0c0e6ecbd3e7a8c    .github
100644 blob 2582d225ab3de985146828e170a4320b16df5206    README.md
100644 blob 9ea743825de328435ce38670898e4109619a4d15    about.md
```

Now we see more detail:

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

6.5.1. What is the PGP signature?

Signed commits are extra authentication that you are who you say you are.

The commits that are labeled with the `verified` tag on GitHub.com

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

```
git cat-file -p fca59e8cca05bb0861f9348a40fe8300b3d55637
```

Skip to main content

```
tree aa02c633322970a4095e564aeb8fd96a3d4f31ab
parent 1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4
author Sarah M Brown <brownsarahm@uri.edu> 1707236481 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1707236481 -0500

add files for organizing
```

6.6. How are commits implemented?

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

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

```
ls .git/
```

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

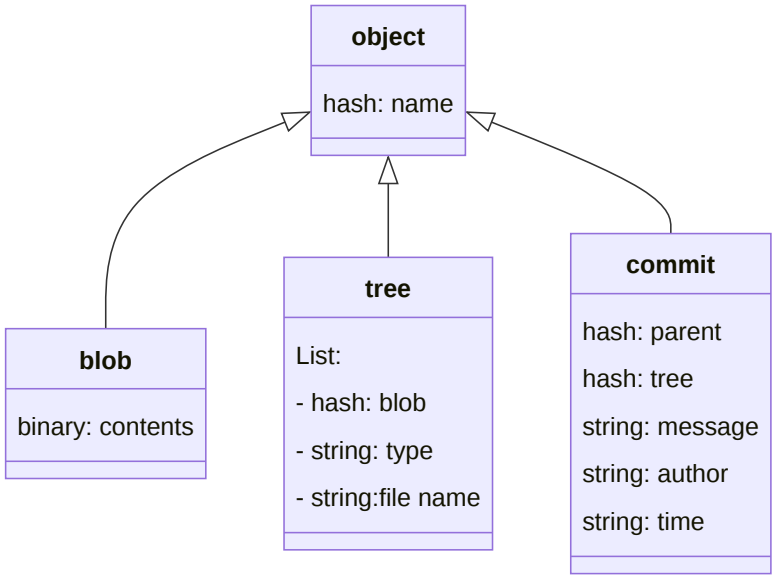
the ones in all caps are files and the others are folders.

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

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

```
00      26      3c      74      86      9e      d3      f5
05      29      42      76      8c      9f      d5      f9
07      2b      5b      79      8e      aa      d8      fc
0c      2d      63      7a      93      ab      e0      info
1e      39      67      7e      9b      b0      e3      pack
1f      3a      6b      81      9d      c8      ed
```

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



Each of those folders is the first 2 digits of at least one hash, or unique identifier for an object. We can list what is in one of those folders

```
ls .git/objects/26
```

```
0c9c309922970f80bfa2c93cc23bcfbb962740
```

Mine has just one, most will in a small repo like this, but it could be mroe than one.

We can look at the plain file using `cat`

```
cat .git/objects/26/0c9c309922970f80bfa2c93cc23bcfbb962740
```

```
x??Aj?0{?+?     +)?JJ?=??`??ņ?.?L??:O?i```d??C??ħ*`?*?h4L?"?kP???b%?LQJ??&Wv??qm?ōGx£??nÁ?p?}o?Y??      ?
```

The content of the file is stored in binary.

To be able to read it we have to use the plumbing command.

```
git cat-file -p 260c
```

```
parent 29ffc88519103085ed3a2ab01cffb3c99d70fc6a
author Sarah M Brown <brownsarahm@uri.edu> 1707242780 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1707242780 -0500

describe the files
```

then we can see the object, mine is a commit, so I see the info like before.

Some of us picked an object that wa a tree or a blob instead.

Skip to main content

```
git cat-file -t 260c
```

```
commit
```

6.7. Commit messages are essential

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

the conventional commits standard is a format of commits

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

Tooling and examples of conventional commits

6.8. Using a codespace

Codespaces are a virutal machine that you can use VSCode on in browser. You only have VSCode access to this system, but VSCode with the terminal is a lot of power.

If VSCode is new to you, use their documentation of the VSCode interface to get oriented to the different parts of the screen.

Before we open a code space, we will first push our content.

```
git push
```

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

    git push --set-upstream origin organization

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

We follow what it says to do next.

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

```
Enumerating objects: 14, done.
Counting objects: 100% (14/14), done.
Delta compression using up to 8 threads
Compressing objects: 100% (11/11), done.
Writing objects: 100% (12/12), 1.75 KiB | 1.75 MiB/s, done.
Total 12 (delta 3), reused 0 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (3/3), completed with 1 local object.
remote:
remote: Create a pull request for 'organization' on GitHub by visiting:
remote:      https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm/pull/new/organization
remote:
```

```
 * [new branch]      organization -> organization
branch 'organization' set up to track 'origin/organization'.
```

Now we will go to our browser.

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

a codespace is a virutal machine on a cloud platform, not cloud access to github.com this means:

- you need to commit changes
- codespace is linux

this is why i teach bash

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

Here we can use the `gh` CLI, this allows us to use a terminal to do github operations instead of only `git` commands.

When we open a new code space, we get a fresh clone of the repo on the virtual machine.

One consequence of this is that if we list the branches, we only have one:

```
git branch
```

We have to list the remote branches, with the `-r` option

```
git branch -r
```

```
  origin/2-create-an-about-file
  origin/HEAD -> origin/main
  origin/fun_fact
  origin/main
  origin/organization
  origin/organizing_ac
```

Then we can check out one of the listed branches and make a local copy of the branch with `gitcheckout`

```
git checkout organization
```

Once we are on a branch, we can open a PR

```
gh pr create
```

It then prompts you how to make the PR

And we can merge from there too

Skip to main content

```
gh pr merge
```

6.9. Prepare for this class

1. Bring git questions or scenarios you want to be able to solve to class on Thursday (in your mind or comment here if that helps you remember)
2. Practice using the new bash commands we have learned to read and understand the workflow files in your KWL repo. Look for bash commands that we have seen before. Try to modify files on a prepare branch so that your name is already filled in when your experience badge (inclass) action runs.

6.10. Badges

Review | Practice

1. Export your git log for your KWL main branch to a file called gitlog.txt and commit that as exported to the branch for this issue. **note that you will need to work between two branchse to make this happen**. Append a blank line, `## Commands`, and another blank line to the file, then the command history used for this exercise to the end of the file.
2. In commit-def.md compare two of the four ways we described a commit today in class. How do the two descriptions differ? How does defining it in different ways help add up to improve your understanding?

6.11. Experience Report Evidence

redirect your `history` to a file `log-2024-02-08.txt` and include it with your experience report.

6.12. Questions After Today's Class

Important

Submit a question as an issue or a PR to this page

7. How do programmers communicate about code?

Tip

check if your codespace has uncommitted changes on github.com/codespaces

note:

- you can only have 2 active at a time(green dots)
- you can see if any have uncommitted changes

Skip to main content

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

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

7.1. Why Documentation

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

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

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

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

In particular documentation tools are really good examples of:

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

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

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

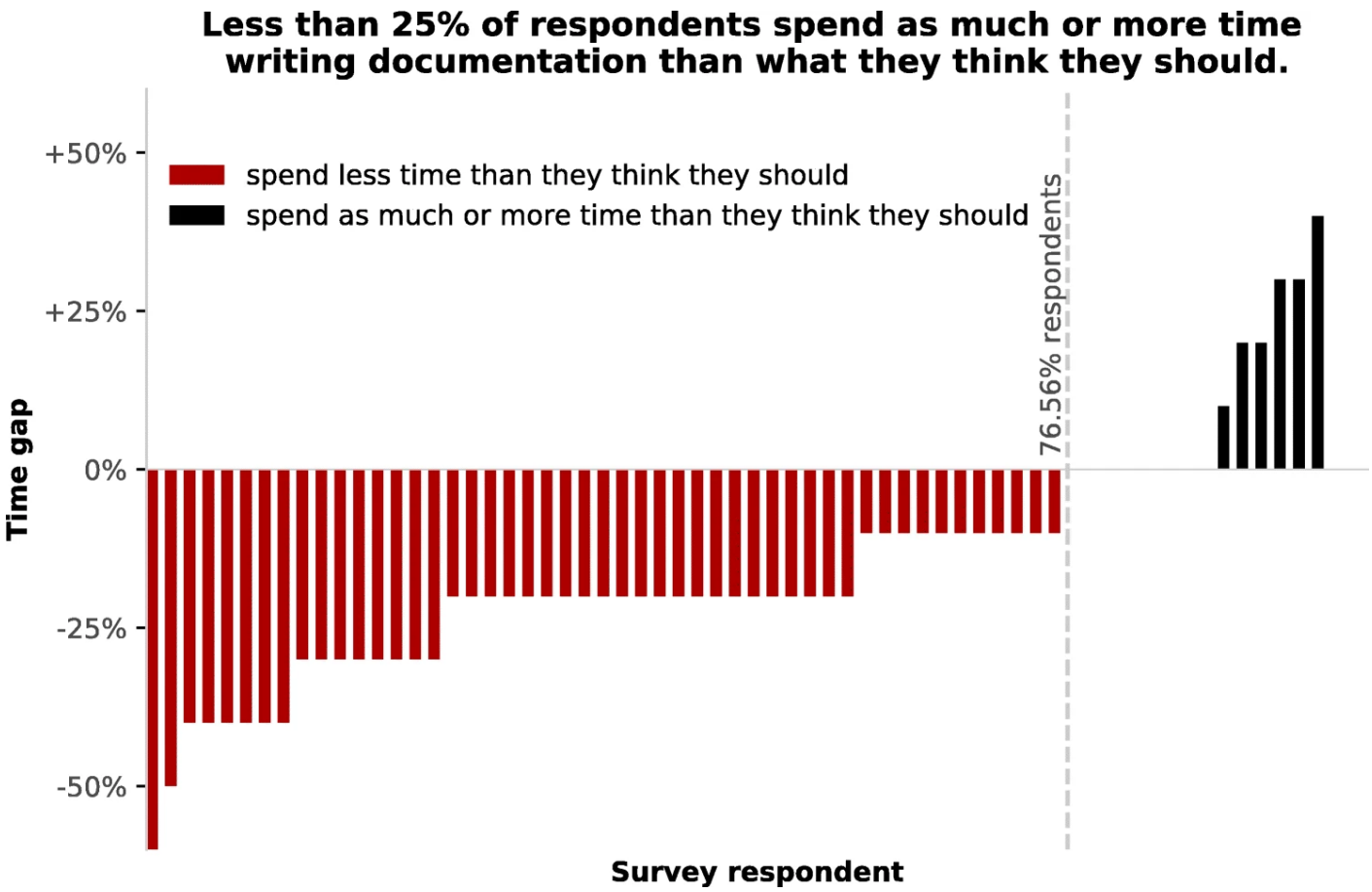
7.2. What is documentation

documentation types table

from ethnography of docuemtnation data science

7.2.1. Why is documentation so important?

we should probably spend more time on it



via source

7.3. So, how do we do it?

Documenation Tools

write the docs

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

7.4. Jupyterbook

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

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

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

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

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

Skip to main content

```
Jupyter Book      : 0.15.1
External ToC      : 0.3.1
MyST-Parser       : 0.18.1
MyST-NB           : 0.17.2
Sphinx Book Theme : 1.0.1
Jupyter-Cache     : 0.6.1
NbClient          : 0.5.13
```

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

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

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

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

Your book template can be found at

    tiny-book/

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

We see that it succeeds

You can make it with any name, beacuse the name is an argument or input

```
jupyter-book create example
```

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

Your book template can be found at

    example/

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

Each one makes a directory, we can see by listing

```
ls
```

```
example                       tiny-book
gh-inclass-sp24-brownsarahm
```

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

```
rm example/
```

Skip to main content

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

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

Instead, we have to tell it two additional things:

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

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

```
rm -rf example/
```

Next we will go into the folder we mad and explore it some

```
cd tiny-book/
```

```

```

```
ls -a
```

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

7.5. Starting a git repo locally

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

We can try `git status`

```
git status
```

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

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

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

```
git init .
```

Skip to main content

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

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

7.5.1. 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
```

and check in with git now

```
git status
```

```
On branch main

No commits yet

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

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

Skip to main content

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

and we will commit the template so that we have it saved as a point we could go back to.

```
git add .
```

We have to add separately because the files are **untracked** we cannot use the `-a` option on commit

and then we will commit with a simple message

```
git commit -m 'jupyter book template'
```

```
[main (root-commit) e34f91d] jupyter book template
 9 files changed, 341 insertions(+)
 create mode 100644 _config.yml
 create mode 100644 _toc.yml
 create mode 100644 intro.md
 create mode 100644 logo.png
 create mode 100644 markdown-notebooks.md
 create mode 100644 markdown.md
 create mode 100644 notebooks.ipynb
 create mode 100644 references.bib
 create mode 100644 requirements.txt
```

7.6. Structure of a Jupyter book

We will explore the output by looking at the files

```
ls
```

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

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

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

Note

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

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
```

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

```
cat _config.yml
```

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

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

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

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

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

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

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

Skip to main content

```
ls
```

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

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

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

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

The one last file tells us what dependencies we have

```
cat requirements.txt
```

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

```
jupyter-book
matplotlib
numpy
```

7.7. Building Documentation

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

```
jupyter-book build .
```

```
Running Jupyter-Book v0.15.1
Source Folder: /Users/brownsarahm/Documents/inclass/systems/tiny-book
Config Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_config.yml
Output Path: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/html
Running Sphinx v4.5.0
making output directory... done
[etoc] Changing master_doc to 'intro'
checking bibtex cache... out of date
parsing bibtex file /Users/brownsarahm/Documents/inclass/systems/tiny-book/references.bib... parsed 5 ent
myst v0.18.1: MdParserConfig(commonmark_only=False, gfm_only=False, enable_extensions=['colon_fence', 'do
myst-nb v0.17.2: NbParserConfig(custom_formats={}, metadata_key='mystnb', cell_metadata_key='mystnb', ker
Using jupyter-cache at: /Users/brownsarahm/Documents/inclass/systems/tiny-book/_build/.jupyter_cache
building [mo]: targets for 0 po files that are out of date
building [html]: targets for 4 source files that are out of date
updating environment: [new config] 4 added, 0 changed, 0 removed
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executing notebook using lo
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
                                                                                ıis validation.
```

Skip to main content

```
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
/Users/brownsarahm/Documents/inclass/systems/tiny-book/markdown-notebooks.md: Executed notebook in 2.18 s
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executing notebook using local CW
0.00s - Debugger warning: It seems that frozen modules are being used, which may
0.00s - make the debugger miss breakpoints. Please pass -Xfrozen_modules=off
0.00s - to python to disable frozen modules.
0.00s - Note: Debugging will proceed. Set PYDEVD_DISABLE_FILE_VALIDATION=1 to disable this validation.
/Users/brownsarahm/Documents/inclass/systems/tiny-book/notebooks.ipynb: Executed notebook in 2.44 seconds

looking for now-outdated files... none found
pickling environment... done
checking consistency... done
preparing documents... done
writing output... [100%] notebooks
generating indices... genindex done
writing additional pages... search done
copying images... [100%] _build/jupyter_execute/137405a2a8521f521f06724f6d604e5a5544cce7bd94d903975cee58b
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?

Now we can look at what it did

```
ls
```

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

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

```
ls _build/
```

Skip to main content

```
html            jupyter_execute
```

and in the html folder:

```
ls _build/html/
```

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

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

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

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

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

7.8. Ignoring Built files

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

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

Git helps us with this with the .gitignore

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

Now we check with git status

```
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)
```

only the gitingore file itself is listed! just as we want.

Skip to main content

```
git add .
```

and finally commit

```
git commit -m 'ignore built site'
```

```
[main 844f6e4] ignore built site
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

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

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

First let's confirm

```
git push
```

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

    git remote add <name> <url>

and then push using the remote name

    git push <name>
```

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

Right now, we do not have any remotes

```
git remote
```

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

More generally, you can create a repo

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

Now we add the remote

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

Then we can try to push

```
git push
```

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

    git push --set-upstream origin main

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

we get an error, becuse we need to link the branch locally to a remote branch

```
git push --set-upstream origin main
```

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

now we get an error because the remote and local have different commits

We follow the instruction again, and pull

```
git pull
```

```
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 5 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (5/5), 1.70 KiB | 348.00 KiB/s, done.
From https://github.com/compsys-progtools/tiny-book-brownsarahm
 * [new branch]      feedback   -> origin/feedback
 * [new branch]      main       -> origin/main
There is no tracking information for the current branch.
Please specify which branch you want to merge with.
See git-pull(1) for details.

    git pull <remote> <branch>

If you wish to set tracking information for this branch you can do so with:

    git branch --set-upstream-to=origin/<branch> main
```

we see the fetch part works, but then the linking fails

We need to tell it how to link on pull

```
git pull origin main
```

Skip to main content

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

and we get an error about how they have different histories.

we tell it how we want it to resolve that

```
git pull origin main --rebase
```

```
From https://github.com/compsys-progtools/tiny-book-brownsarahm
 * branch            main       -> FETCH_HEAD
Successfully rebased and updated refs/heads/main.
```

and success!

and now we can actually push

```
git push --set-upstream origin main
```

```
Enumerating objects: 15, done.
Counting objects: 100% (15/15), done.
Delta compression using up to 8 threads
Compressing objects: 100% (12/12), done.
Writing objects: 100% (14/14), 16.53 KiB | 8.26 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/compsys-progtools/tiny-book-brownsarahm.git
   e5287d8..22f2227  main -> main
branch 'main' set up to track 'origin/main'.
```

7.10. Prepare for next class

1. Think through and make some notes about what you have learned about design so far. Try to answer the questions below in design_before.md. If you do not now know how to answer any of the questions, write in what questions you have.

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

Skip to main content

7.11. Badges

Review | Practice

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book. Set it so that the `_build` directory is not under version control. Complete basic customization setps for the necessary files and ensure that you do not add template files to your KWL repo.
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.

build idea build a sphinx extension that adds a particular feature to a documentation website. You can start a proposal and discuss ideas with Dr. Brown

7.12. Experience Report Evidence

Link to your tinybook repo. in the experience report PR

7.13. Questions After Today's Class

7.13.1. Can you create a new remote through the terminal without using the github ui?

If you knew the URL of the remote, yes, if the remote is on GitHub, you have to use either github.com in your browser or use the `gh cli` tool's command `gh repo create`

7.13.2. How is jupyterbook different than other ides or editors?

`jupyter-book` is not an editor or IDE, it is a tool for building websites or pdfs.

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

Jupyter-book is an opinionated distribution of sphinx which can also be used to document other languages like C++

Skip to main content

7.13.3. What is the .doctrees/ folder inside _build/ with files like environment.pickle?

That is an intermediate step between the markdown and the final HTML.

There is api level docs

7.13.4. Why is building necessary? Couldn't it just be a part of compiling?

Building is a more general process of transforming from source to an output format. Compling is a specific step withing building for some programming languages. We will learn more about the build process for C later.

What we did today was building, but not compiling.

7.13.5. What other uses are there for jupyter notebook

The course website is an example. They also maintain a gallery of jupyter books

7.13.6. Did we create a repo locally?

yes `git init .` created a repo locally

7.13.7. How to add more files and sections to your Jupyter Books website

Create more files in the folder and then update the `_toc.yml`. For example, this whole website is jupyter book, see how many files are in the repo

7.13.8. how will we continue to use jupyter books going forward in this class?

Your badge task is to convert your KWL repo to one. The course website is also a jupyter-book, so making more complex contributions to the site is one way to practice with it.

For example you might do an explore or build that contributes to the course website.

Also, most builds will require documentation using *a* documentation builder, which you might choose jupyter-book or to use a different one, but some of what we saw here will apply, making it easier to learn another *similar* tool.

7.13.9. how do we use the developer tools we talked about, for example the bibliography one?

Within jupyter-book, their documentation includes a tutorial on references. This is a good high quality source because it is written by developers of a tool and describes in concrete terms what you need to do, with appropriate context.

To get the bibtex formatted reference for a source, you can use this generator for informal sources(like websites) and google

More generally, bibtex was designed to be used with latex which is easiest to get started with using the cloud-hosted, live collaboration, version in overleaf.

8. How do I choose a Programming Language for a project?

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.

8.1. Admin

This is a posted class with opportunities for Experience, Review, and Practice badges. The practice badge also has ideas in it for an explore badge.

This is a fully asynchronous activity that you can complete alone and has options for how to dicuss it with classmates. You can choose either version.

> **Important**
>
> Future classes do not directly build on top of steps in this activity so this can be made up later without putting you behind.

8.2. Set up for credit

Start your experience badge at the start of working on this activity and then you can add your work to it throughout.

You may work alone or with classmates.

> **Important**
>
> Use the makeup workflow with an ISO formated date (YYYY-MM-DD).

> **Important**
>
> This PR should auto-request @brownsarahm; do not change it

8.3. 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

```
| feature | <language> | <language></language> |
| ------ | ------ | ------ |
| use of whitespace     | Text     | Text     |
| list/array types |   | |
| variable typing |  | |
| memory usage | | |
```

Post your table on GitHub in the language comparisons discussion](orgs/compsys-progtools#10). Reply to at least two peers offering either advice or asking additional questions.

> **Note**
>
> Link to your post and comments in your experience PR

8.4. Learn more

Skim through these resources:

- intro to PL
- categorizing progrmaming languages

While you read look for the following key points:

- What other properties besides the syntax and efficiency criteria above can you use to compare programming languages
- how do types vs paradigms compare as ways to categorize languages
- how do interpretation and compilation compare?

8.5. PL in Developer Survey

Read Carefully the developer survey// languages section for 2023 (2024 will not occur until late spring/early summer)

Include answers to the following questions in your experience report for today. You may work with classmates and use additional sources to find answers to these questions. If they are not basic facts about the langague (eg compiled vs interpretted; open source vs not; libarary ecosystem) include links to where you found the information.

- 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 over the years of the survey?

8.6. Prepare for next class

No additional preparation,the prep for 2024-02-20 is now for 2024-02-22.

8.7. Badges

Review **Practice**

1. Choose two languges with different standings in the desired admired list. THis could be a highly admired and least desired; it could be one with a small gap and one with a large gap. Read a few posts about each language and try to figure out why it is/not admired or desired. Summarize your findings. Include links to all of the posts you read in a section titled `## Sources` in your markdown file. For each source, make a bulleted list with some notes about the author's background and any limitations that might put on the scope of their opinions. (for example, a data 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`.

8.8. Explore badge option

Actually try out one of the langauges above in a new repo created from this asssignment.

work on a branch and commit after each step

1. Setup the devcontainer to provide the compiler/interpreter for the language of your choice. See features
2. Add top extensions related to the language to the devcontainer.
3. Add a gitignore file for that language
4. Write a hello world type program in the language and make sure it runs.
5. Use the official documentation to figure out a common task in the language and write an example program that completes something typical for that language (eg a sort data analysis in R). You may use GitHub Co-pilot in the code space to help write the code.
6. Test your understanding of the code by trying to modify it.
7. Fill in the prompts in the README (follow the comments)
8. Answer the questions in the README of the template repo.

8.9. Experience Report Evidence

If in class you should have the alternate file we made. If for makeup, your report should have all of the content above.

8.10. Questions After Today's Class

> **Important**
>
> These questions are from past semesters, you can add your own via a PR to the course website for a community badge or leave it in your experience report and I will add it later

8.10.1. Is it possible to only work with one language and never even consider using other languages?

Maybe if you work at one company forever, but not easy.

8.10.2. Are deep learning libraries that are changed from python to another language open source?

It's not so much that they're changed from Python to other langauges as far as I know. For example tensorflow has a C++ API and the optimized parts of it are written in C++. It is a Python library, but more than half of the content in the repo is C++. It is open source though.

8.10.3. What typically makes languages go obsolete?

Some stay necessary in a small sense forever (as far as we have experienced so far) because something critical is written in it. For example COBOL has been basically considered irrelevant by computer scientists the whole time, but it is still what 95% of ATM swipes rely on.

More often I would say a language might never become popular and then becoe obsolete or a version of a language can become obsolete.

More research on this is a good explore badge.

8.10.4. Is it worthwhile to learn how Docker works in order to be more employable?

Docker is worth learning.

8.10.5. How often should I be weighing my options for the programming language I'm going to use for a project?

If you are starting a project from scratch it is worth thinking about it.

8.10.6. What class would give you a license to MATLAB so you can get experience

for professional use?

Here is URI's MATLAB access information. You do not need to be in a specific class.

9. Why are these tools like this?

> **Important**
>
> Remember: this is an upper level course; I am not looking for you to give me back the things I say to you.
>
> I am looking to see that you can use and apply the things we do in other contexts.
>
> - sometimes this is practical: eg git & bash
> - sometimes this is conceptual: eg PL & today
>
> This means I look for you to draw connections between ideas.

9.1. Setup

Let's take some time to se up what we are going to talk about today.

We will talk about two main things:

- Unix Philosophy
- Strategies to study systems

In CS we tend to be more implicit about design, but that makes it hard to learn and keep track of.

Today's goal is to be more explicit, disucssing some principles and seeing how you have already interacted wit them.

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

Understanding the context and principles will help you:

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

9.1.1. Why should we study design?

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

9.1.2. Why are we studying developer tools?

Recall:

> The best way to learn design is to study examples [Schon1984, Petre2016], and some of the best examples of software design come from the tools programmers use in their own work.

Software design by example

note

- we will talk about some history in this course
- I will not take a “great men of history” approach, where I have you learn a bunch of people and what each one of them did

This is because:

- I think that history is important context for making decisions
- Many of these “great men” are actually, in many ways, Not Great^TM
- It *is* important to remember that all of this work was done by *people*
- all people are imperfect
- when people are deeply influential, ignoring their role in history is not effective, we cannot undo what they did
- we do not have to admire them or even say their names to acknowledge the work
- computing technology has been used in Very Bad ways and in Definitely Good ways

9.2. Unix Philosophy

sources:

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

The tenets:

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

For better or worse unix philosophy is dominant, so understanding it is important.

This critique is written that unix is not a good system for “normal folks” not in its effectiveness as a context for *developers* which is where *nix (unix, linux) systems remain popular

Skip to main content

context:
“normal folks” is the author’s term; my guess is that it is attempting to describe a typical, nondeveloper computer user terminology to refer to people has changed a lot over time; when we study concepts from primary sources, we have to interpret the document through the lens of what was normal at the time the document was written, not to excuse bad behavior but to not be distracted and see what is there

In what ways might it make sense to design developer tools differently than everyday user tools?

9.2.1. Philosophy in practice, using pipes

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

If you *do not* have `gh` CLI working locally, open a codespace on the main branch of your KWL repo

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

To check if your `gh` CLI is working:

```
gh
```

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

USAGE
  gh <command> <subcommand> [flags]

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

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

EXTENSION COMMANDS
  classroom:   Extension classroom

ALIAS COMMANDS
  co:          Alias for "pr checkout"

ADDITIONAL COMMANDS
  alias:       Create command shortcuts
  api:         Make an authenticated GitHub API request
  completion:  Generate shell completion scripts
  config:      Manage configuration for gh
  extension:   Manage gh extensions
  gpg-key:     Manage GPG keys
  label:       Manage labels
  ruleset:     View info about repo rulesets
  search:      Search for repositories, issues, and pull requests
```

Skip to main content

```
  ssh-key:     Manage SSH keys
  status:      Print information about relevant issues, pull requests, and notifications across repositor
  variable:    Manage GitHub Actions variables

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

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

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

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

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

We will navigate to the kwl repo

```
cd kwl-sp24-brownsarahm/
```

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

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

```
name: Create badge issues (Do not run manually)
on:
  workflow_dispatch

# once you edit, change the name

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

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        # prepare badge lines
```

Skip to main content

```
        # review badge lines
        rtitle="review-"$(sysgetbadgedate --review)
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        # practice badge lines
        pratitle="practice-"$(sysgetbadgedate --practice)
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file
      env:
        GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
#  edit the run step above for the level(s) you want.
#  You should keep the prepare, because they are required for experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

We see that several things, but this output is also kind of long,

We can cut only a part of the output by using a pipe. A pipe connects the output of one command to the input of the next. It uses stdout and stdin files to do this.

The `head` command returns the top of a file and its `-n` option lets us choose how many lines at the top of the file, so we will use this to get the first 26 lines.

```
cat .github/workflows/getassignment.yml | head -n 26
```

(I picked 26 in advance to get the right number for this example, this)

```
name: Create badge issues (Do not run manually)
on:
  workflow_dispatch

# once you edit, change the name

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

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        # prepare badge lines
        pretitle="prepare-"$(sysgetbadgedate --prepare)
        sysgetassignment --type prepare | gh issue create --title $pretitle --label prepare --body-file -
```

Here we see a few key things:

- on the last line it uses a pipe `|` to connect a command `sysgetassignment` to the `gh issue create` command.
- the `gh issue create` command uses the `--body-file` option with a value `-`; this uses std in, but since this is to the right of the pipe, it puts the output of the first command into this option
- the 2nd to last line creates a variable (we will learn this more later) and the last line uses that variable

Skip to main content

We know that this action is what is used to create badge issues, so this custom code must be what gets information from the course website to get the assignments and prepare them for your badge issues.

9.2.2. Installing from source

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

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

```
Collecting git+https://github.com/compsys-progtools/courseutils@main
  Cloning https://github.com/compsys-progtools/courseutils (to revision main) to /private/var/folders/8g/
  Running command git clone --filter=blob:none --quiet https://github.com/compsys-progtools/courseutils /
  Resolved https://github.com/compsys-progtools/courseutils to commit 2220b76eba38662aa1a6172fdfd46700e3f
  Preparing metadata (setup.py) ... done

Successfully installed syscourseutils-0.4.0
```

This shows at the top it clones the repo to a temporary location on your system (or codespace) and then installs it from the source code.

> **Note**
>
> I truncated this output for the website

Now we can use a command from the packaage.

```
sysgetassignment
```

```
404: Not Found
```

With default settings, it says 404.

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

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

```
sysgetassignment --help
```

```
Usage: sysgetassignment [OPTIONS]

  get the assignment text formatted (CLI entrypoint)

Options:
  --type TEXT  type can be prepare, review, or practice
  --date TEXT  date should be YYYY-MM-DD of the tasks you
```

Skip to main content

```
              want
  --help      Show this message and exit.
```

Now we can see that it can take some options.

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

We can use the options to get the last posted practice badge

```
sysgetassignment --date 2024-02-20 --type practice
```

```
- [ ] Describe a type of project where it would be worth it for you to learn a language you have never us
- [ ] Learn about one of the following languages that you have not used before: [R](https://www.r-project

    ```
    - [ ] What is this language designed for?
    - [ ] What Programming paradigm(s) does it support?
    - [ ] Give at least 1 example of a programming language that experience in would make it **easy** to
    - [ ] Give at least 1 example of a programming language that experience in would make it **hard** to
    - [ ] What is its most unique feature(s)?
    ```

hint: for questions 3 & 4, for example learning Python first does not make it easy to learn C++, but lear

Explore badge option

Actually try out one of the langauges above in a new repo created from [this asssignment](https://classro

work on a branch and commit after each step

- [ ] Setup the devcontainer to provide the compiler/interpreter for the language of your choice. See [fe
- [ ] Add top extensions related to the language to the devcontainer.
- [ ] Add a [gitignore](https://github.com/github/gitignore) file for that language
- [ ] Write a hello world type program in the language and make sure it runs.
- [ ] Use the official documentation to figure out a common task in the language and write an example pro
- [ ] Test your understanding of the code by trying to modify it.
- [ ] Fill in the prompts in the README (follow the comments)
- [ ] Answer the questions in the README of the template repo.
```

and it works!

By calling without the the type, we can see what it gives as the default,

```
sysgetassignment --date 2024-02-20
```

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

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

which we see is prepare

Skip to main content

We can also install from a different branch

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

```
Collecting git+https://github.com/compsys-progtools/courseutils@sub_command
  Cloning https://github.com/compsys-progtools/courseutils (to revision sub_command) to /private/var/fold
  Running command git clone --filter=blob:none --quiet https://github.com/compsys-progtools/courseutils /
  Running command git checkout -b sub_command --track origin/sub_command
  Switched to a new branch 'sub_command'
  branch 'sub_command' set up to track 'origin/sub_command'.
  Resolved https://github.com/compsys-progtools/courseutils to commit bbf6192f911f7ce295e4cc6da4b011c5320
  Preparing metadata (setup.py) ... done
Requirement already satisfied: Click in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (from s
Requirement already satisfied: pandas in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (from
Requirement already satisfied: lxml in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (from sy
Requirement already satisfied: numpy in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (from s
Requirement already satisfied: requests in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (fro
Requirement already satisfied: html5lib in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (fro
Requirement already satisfied: six>=1.9 in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages (fro
Requirement already satisfied: webencodings in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages
Requirement already satisfied: python-dateutil>=2.8.2 in /Users/brownsarahm/anaconda3/lib/python3.11/site
Requirement already satisfied: pytz>=2020.1 in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages
Requirement already satisfied: tzdata>=2022.1 in /Users/brownsarahm/anaconda3/lib/python3.11/site-package
Requirement already satisfied: charset-normalizer<4,>=2 in /Users/brownsarahm/anaconda3/lib/python3.11/si
Requirement already satisfied: idna<4,>=2.5 in /Users/brownsarahm/anaconda3/lib/python3.11/site-packages
Requirement already satisfied: urllib3<3,>=1.21.1 in /Users/brownsarahm/anaconda3/lib/python3.11/site-pac
Requirement already satisfied: certifi>=2017.4.17 in /Users/brownsarahm/anaconda3/lib/python3.11/site-pac
Building wheels for collected packages: syscourseutils
  Building wheel for syscourseutils (setup.py) ... done
  Created wheel for syscourseutils: filename=syscourseutils-1.0.2-py3-none-any.whl size=16292 sha256=10ce
  Stored in directory: /private/var/folders/8g/px8bm7bj0_j31j71yh6mfd_r0000gn/T/pip-ephem-wheel-cache-mme
Successfully built syscourseutils
Installing collected packages: syscourseutils
  Attempting uninstall: syscourseutils
    Found existing installation: syscourseutils 0.4.0
    Uninstalling syscourseutils-0.4.0:
      Successfully uninstalled syscourseutils-0.4.0
Successfully installed syscourseutils-1.0.2
```

Note the version updates, and goes from 0.4.0 to 1.0.2. I made some breaking changes on this branch, so I used semantic versioning to jump the major.

It now has an overall command with subcommands. we can use that command, cspt in order to get the list of other commands.

```
cspt
```

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

Options:
  --help  Show this message and exit.

Commands:
  createtoyfiles  from a source file create a set of...
  earlybonus      check if early bonus is met from...
  exportac        export ac files for site from lesson
  exporthandout   export prismia version of the content
  exportprismia   export prismia version of the content
  getassignment   get the assignment text formatted
```

Skip to main content

```
kwlcsv          generate the activity file csv file...
mkchecklist     transform input file to a gh markdown...
parsedate       process select non dates
prfixlist       check json output for titles that...
processexport   transform output from mac terminal...
progressreport  list PR titles from json or - to use...
titlecheck
```

This command should now work for you in your KWL, but we will revisit this in lab.

```
gh pr list -s all  --json title,latestReviews | cspt prfixlist - | cspt mkchecklist -
```

9.3. How do we Study (computer) Systems

Important

This section has more detail than we did in class

When we think of something as a system, we can study it different ways:

- input/output behavior
- components

For complex systems, we also use *abstraction*

These basic ideas apply whether a computer system or not. We can probe an object that we want to udnerstand 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 in this course.

When we study a system we can do so in three main main ways. We can look at the input/output behavior of the system as a whole, we can look at the individual components, or we can adopt the abstractions that were developed.

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.

Skip to main content

But that fact motivates another way to study a complex system, across levels of abstraction. You can abstract away details and focus on one representation. This can be tied literally to components, but it can also be conceptual. For example, in CSC211 you use a model of stack and heap for memory. It's useful for understanding programming, but is not exactly what the hardware does. At times, it is even more useful though than understanding exactly what the hardware does. These abstractions also serve a social, practical purpose. In computing, and society at large really, we use *standards* these are sets of guidelines for how to build things. Like when you use a function, you need to know it's API and what it is supposed to do in order to use it. The developers could change how it does that without impacting your program, as long as the API is not changed and the high level input/output behavior stays the same.

9.3.1. 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).

Run the following command and describe what happens

This gives a list of PRs that you might want to check on

use pipes to produce list of bad titles, pipe into checker

9.3.2. Components

> Take it apart, asess the pieces

Skip to main content

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.

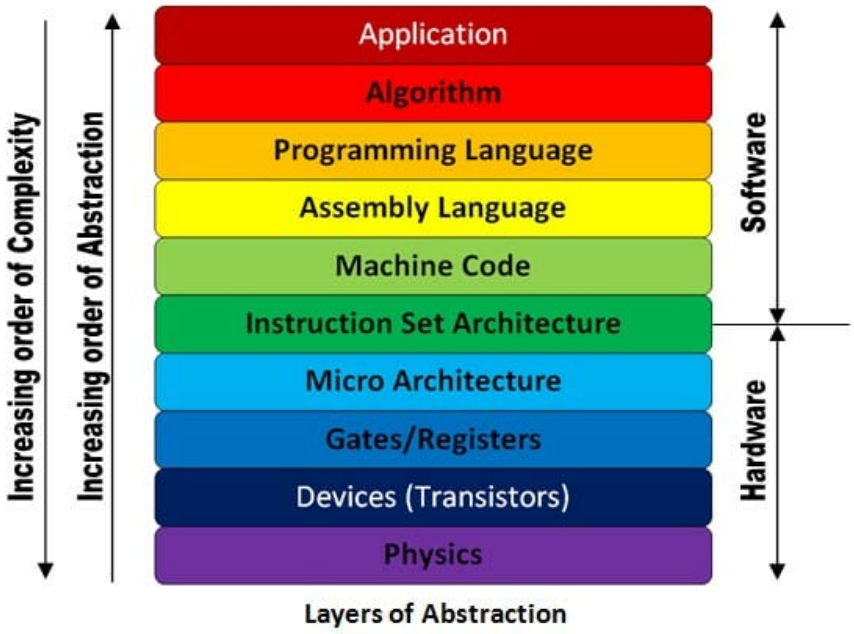
Let's break the command down and examine each piece. What piece should we start from?

9.3.3. Abstraction

> use a model

As we talked about the behavior and abstraction, we talked some about software and some about hardware. These are the two coarsest ways we can think about a computer system at different levels of abstraction. We can think about it only in physical terms and examine the patterns of electricity flow or we can think about only the software and not worry about the hardware, at a higher level of abstraction.

However, two levels is not really enough to understand how computer systems are designed.



Application - the software you run.

Skip to main content

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 dit.

The microarchitecture is the specific circuits: networks of smaller individual components. Again, we can treat the components as blocks and focus on how they work together. At this level we still have calculations like add, multiply, compare, negate, and we can store values and read them. That is all we have at this point though. At this level there are all binary numbers.

The actual gates (components that implement logical operations) and registers (components that hold values) break everything down to logical operations. Instead of adding, we have a series of `and`, `nand`, `or`, and `xor` put together over individual bits. Instead of numbers, we have `registers` that store individual zeros or ones. In a modern digital, electrical computer, at this level we have to actually watch the flow of electricity through the circuit and worry about things like the number of gates and whether or not the calculations finish at the same time or having other parts wait so that they are all working together. We will see later that when we try to allow multiple cores to work independently, we have to handle these timing issues at the higher levels as well. However, a register and gate can be implemented in different ways at the device level.

The device (or transistor in modern electrical digital computers) level, is where things transition between analog and digital. The world we live in is actually all analog. We just pay attention to lots of things at a time scale at which they appear to be be digital. Over time devices have changed from mechanical switches to electronic transistors. Material science innovations at the physics level have improved the transistors further over time, allowing them to be smaller and more heat efficient. Because of abstraction, these changes could be plugged into new hardware without having to make any changes at any software levels.

Skip to main content

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.

9.4. Prepare for this class

1. review the notes on what is a commit. In gitdef.md on the branch for this issue, try to describe git in the four ways we described a commit. **the point here is to think about what you know for git and practice remembering it, not "get the right answer"; this is prepare work, we only check that it is complete, not correct**
2. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file below. We will come back to these notes in class later, but it is best to record over a time period instead of trying to remember at that time. Store your notes in your kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for after spring break, it does not go in the Feb 27 experience branch; keep this branch open until it is specifically asked for**

9.5. Badges

Review | **Practice**

1. Read today's notes when they are posted.
2. Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Review the open source project you looked at for the `software.md` file from before and decide if it more adheres to or deviates from the unix philosophy. Add a `## Unix Philosophy <Adherance/Deviation` section to your software.md, setting the title to indicate your decision and explain your decision in that section. Provide at least two specific examples, using links to specific lines of code or specific sections in the documentation that support your claims.
3. create methods.md and answer the following:

```
- Reflecting on your own programming practice, is initial understanding of concepts or debugging later mo
- which of the three methods for studying a system do you use most often in your chosen context? Give an
- do you think using a different strategy might help you in your chosen context sometimes? why or why not
```

Skip to main content

9.6. Experience Report Evidence

9.7. Questions After Today's Class

9.7.1. when will we be starting the collabrative work?

Next week in lab, you'll have time to form groups and start working together

9.7.2. Are the `sysgetassignment` and `cspt` stuff just to work on badges locally?

And they're how I get the issues created in your repos. This exercise was to show you behind the scenes how the course itself applies some of these concepts.

9.7.3. Would it make sense to make terminals more user friendly?

In a sense, we have done some; there is still more of what is called "cognitive load" or how much a person has to hold in memory, but at the same time, the interactive nature helps. It means you can, for example, run a command with the help option and then figrue out what you need. There are added colors, and more advanced tab complete options.

You can also customize your terminal locally and add aliases, to autmate things you personally do often or make it more like how you want it to be.

explore ideas: (do one of the following and document/write it up)

- customize your prompt on your main terminal to align with how your think or your personal preferences
- create an alias to make something you do often easier
- customize your vscode experience and share your customized settings as a recommendation to others with explanation of what you like and why

9.7.4. What is your definition of abstraction?

I use it in different contexts, in slightly different ways (eg teaching this class is very different than my research collaboration with affective neuroscientists writing a theoretical piece on a new approach to psychological science). The most concise, useful definition I like is that to abstract is to hide or summarize a level of detail that is not useful for the taks at hand.

9.7.5. What are More differences between monolothic and composable designs?

This is a good explore badge topic.

Skip to main content

10. What *is* git?

10.1. Admin

> **Important**
>
> These notes are a bit different than the ones I made in class

- Please start commenting/expressing interest on build/explore ideas. Next lab will be focused on that.
- I will be merging the courseutils update soon, so you can see the docs more easily
- creating a codetour will be a part of many builds
- creating documentation websites is also a part of many builds

10.1.1. Support for build and explore proposals

- build proposal template
- explore proposal instructions

10.1.2. Build and Explore ideas

- build - high level
- build courseutils
- explore - high level
- explore- courseutils

10.1.3. get credit for this class toward your major

The department has decided that this course is acceptable for the 300+ level elective degree requirement in the CS major, but it requires that you complete a form called a “Curriculum Modification”.

I added a template form from Professor DiPippo (CS Undergraduate Chair) to the course website repo in the `img` folder, it is a pdf.

Complete this form and send it to Professor DiPippo for her signature to make this course count toward your major requirements.

the form indicates a different email address to send it to, but that is after Professor DiPippo signs the form, do not send it directly to the Dean’s office (casforms)

10.2. Why define git now?

Two weeks ago learned about what a commit is and then we took a break from how git works, to talk more about how developers communicate about code, programming languages, and the unix philosophy

study tip

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

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

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

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

Today, we are going to learn *what* git is and what its parts are. Next we will see in greater detail *how* it works.

git book is the official reference on git.

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

10.3. Git Definition

From here, we have the full definition of git

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

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

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

Let’s break down the definition

10.3.1. Git is a File system

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

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

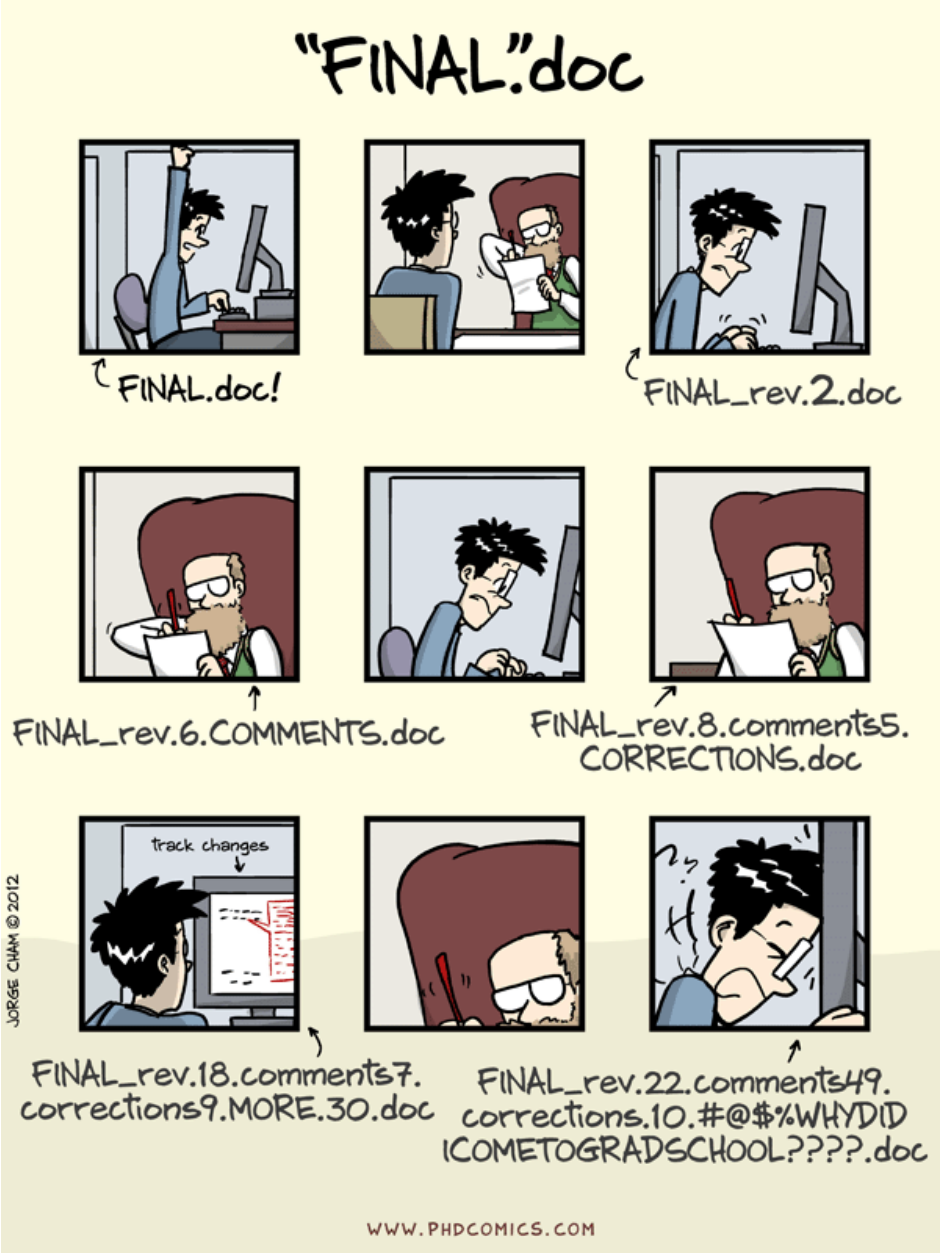
- python dictionaries
- pointers (address,content)
- parameter, passed values

Skip to main content

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.

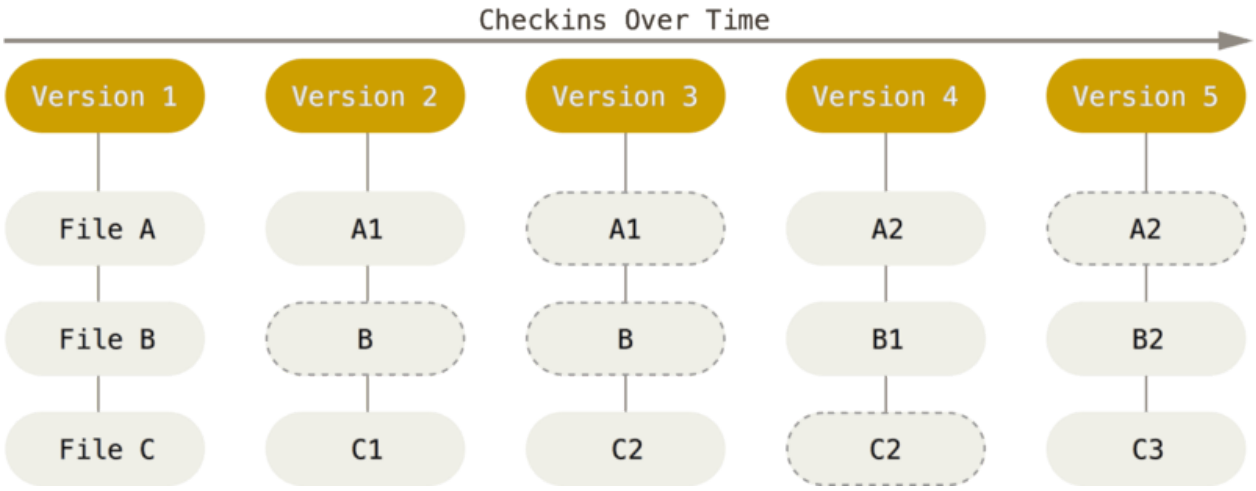
10.3.2. Git is a Version Control System

In the before times

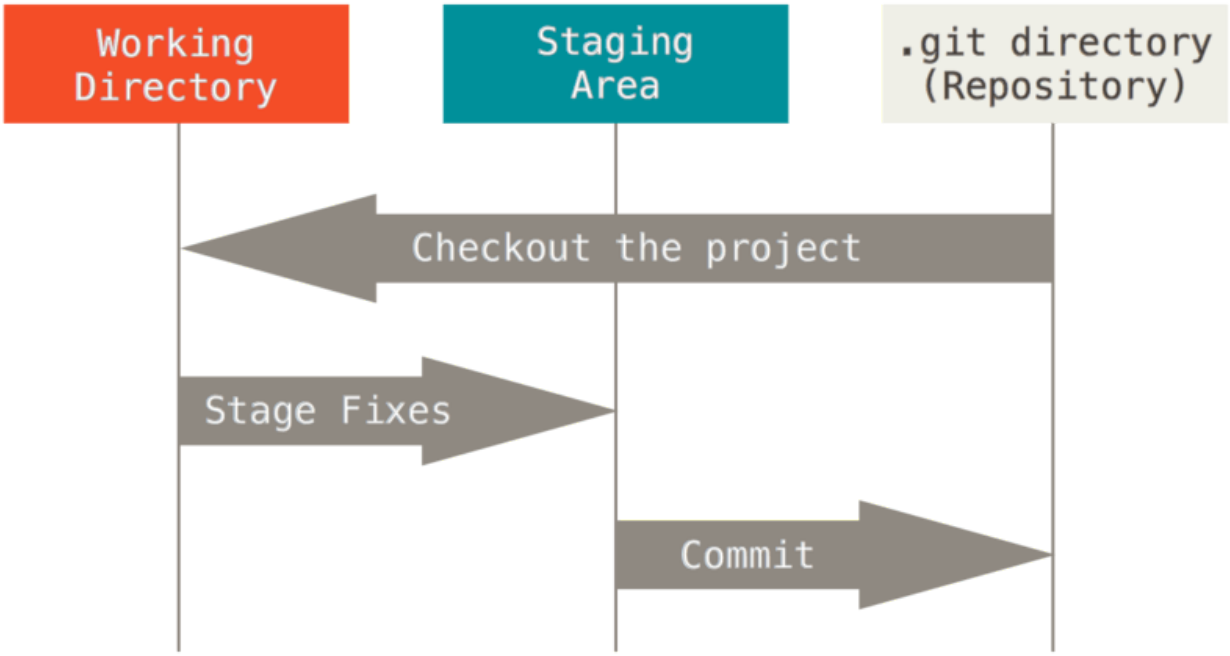


git stores **snapshots** of your work each time you commit.

Skip to main content



it uses 3 stages:



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

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

10.4. 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

Skip to main content

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.

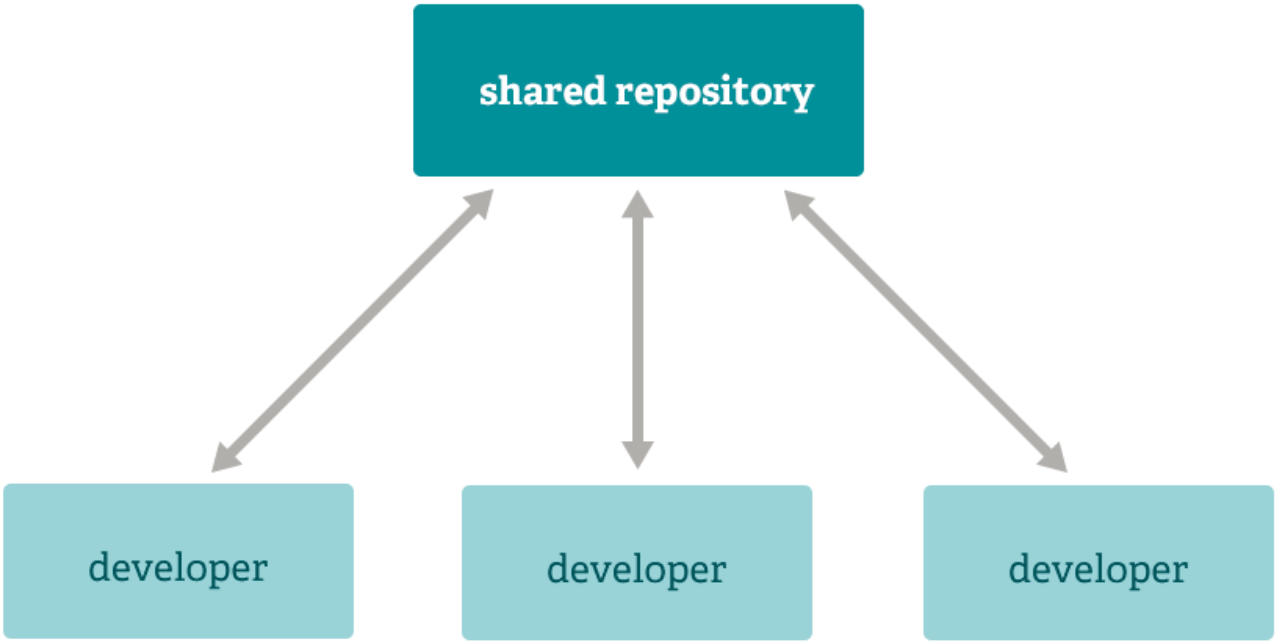
10.5. Git is distributed

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.

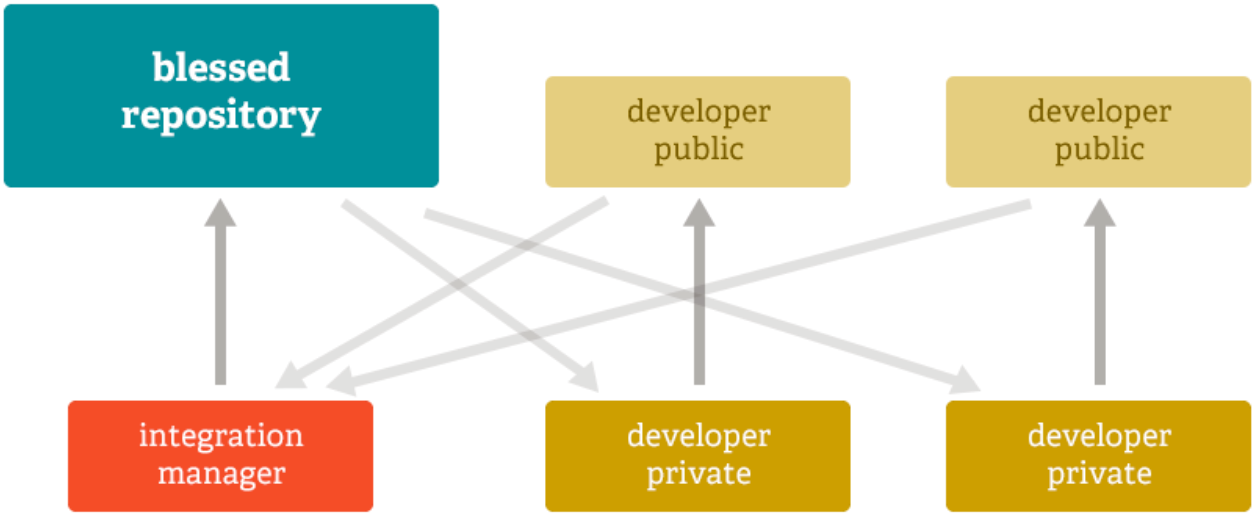
For human reasons, we like to have one “official” copy and treat the others as local copies, but that is a social choice, not a technological requirement of git. Even though we will typically use it with an offical copy and other copies, having a tool tht does not care, makes the tool more flexible and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

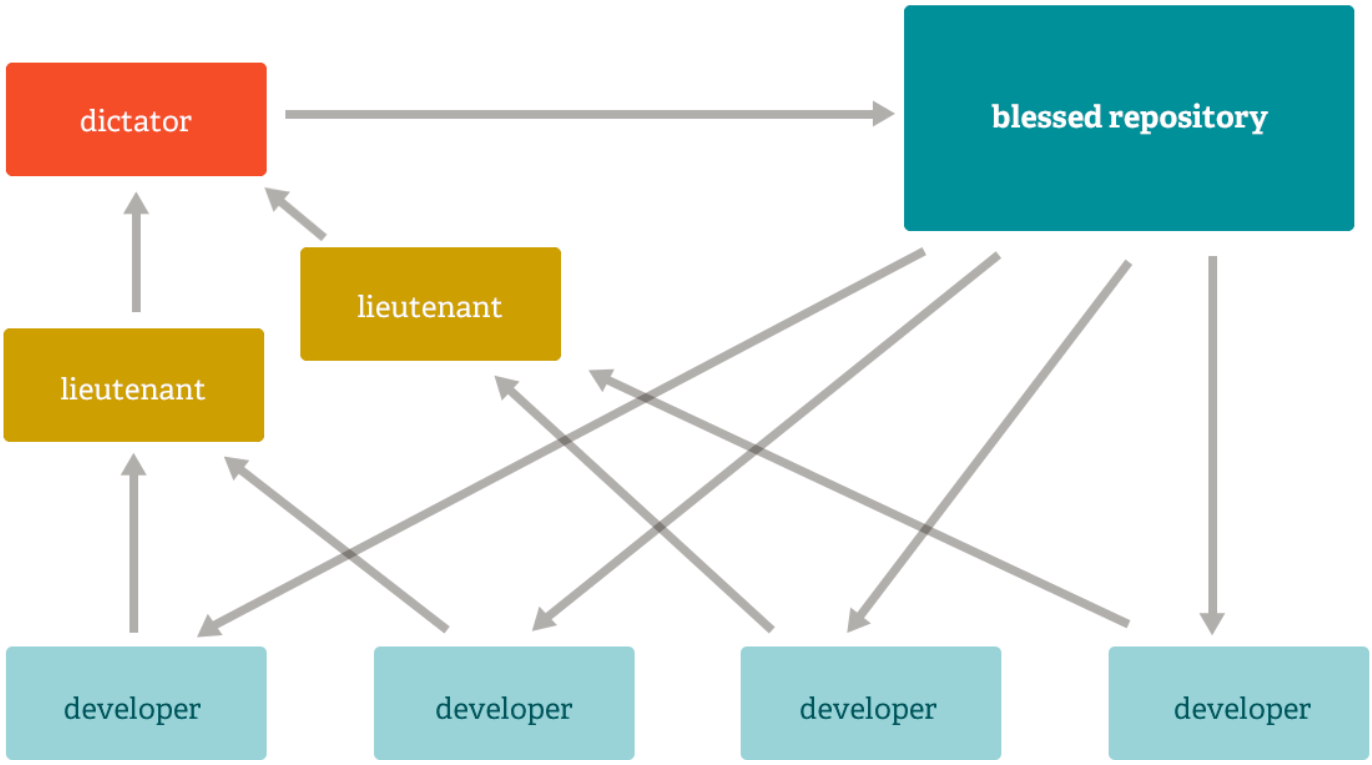
10.5.1. Subversion WOrkflow



10.5.2. Integration Manager



10.5.3. dictator and lieutenants



10.6. How does git do all these things?

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

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

Skip to main content

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

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

```
.git/objects//0d/6ef1d3db0729088d515b35b588f39af5b770fd
.git/objects//0c/67356c8bb0c86efa2c42cb9db9837312559020
.git/objects//9b/87264e80fd24198ff8df6d071eaccc6a913b20
.git/objects//9e/ce5efa25710c8fad7d9f210928785b5362b06f
.git/objects//9e/a743825de328435ce38670898e4109619a4d15
.git/objects//3c/7ace1d193b78129c10d61609cbe97b2686f202
.git/objects//67/1765c67072a10ea8fc89704a8f66d504fcf80f
.git/objects//93/c08483f44ebdf5ce10e6c0002e641aa0cc8844
.git/objects//05/1781ee70ce3f052d0a233d2efc5a34f921877e
.git/objects//9d/6ffa6ded47d8b6df13ed60e482b188015ee499
.git/objects//d8/7bf4a5641e0429fd3c371bd2b19d755105ca92
.git/objects//ab/e403c82238c3877b3437c1ab3e32499a60e24f
.git/objects//fc/a59e8cca05bb0861f9348a40fe8300b3d55637
.git/objects//f5/b2c8dd1d307b1114d70a7e5fe7dc37cc429d35
.git/objects//e3/b192aa0cd490226e8adcd81d3d0b95adb5676b
.git/objects//c8/dc2d1d10e6c27cad4f3be278016f3cf90ac1ca
.git/objects//ed/7b329f735e91c76a22236dc728034914ec29ee
.git/objects//pack/pack-ca5f27a0787f6fe7c793b73ecd54664cc2f6af14.rev
.git/objects//pack/pack-ca5f27a0787f6fe7c793b73ecd54664cc2f6af14.idx
.git/objects//pack/pack-ca5f27a0787f6fe7c793b73ecd54664cc2f6af14.pack
.git/objects//29/ffc88519103085ed3a2ab01cffb3c99d70fc6a
.git/objects//42/f16e84e01cc75ddf80fd19b150de5818db596c
.git/objects//1f/6ef3acccb91bc0c33661854903ec5e7d666fb5
.git/objects//74/3db376fa76bb3611cfac6935938d179330c7eb
.git/objects//7e/78493266aeef733137c5193cc0a0e465e4093d
.git/objects//26/0c9c309922970f80bfa2c93cc23bcfbb962740
.git/objects//81/c6f187f146caaaf43d97bc1bb8ed237142f4c3
.git/objects//86/c90cf823148f4a3b53b04ab5498605cbf6ccb1
.git/objects//9f/399466ad6a1ad572e104209f4469eb8cd48516
.git/objects//6b/eb67c7f42a4829d5b0512a19efefe8de7f9eec
.git/objects//07/897fdccf1f5e214e33ab58bc755be90edbc7e2
.git/objects//07/959c0c04e0d4a4b2df24ab5bd08b17e3e58938
.git/objects//00/4e5366de9503d7e3fe263171a7bcb4e0af88eb
.git/objects//3a/4533a3abbc749f5e1905b30eb187a7350ae71a
.git/objects//5b/03a3e101e049f7a0184c6b0a1adc4787056af0
.git/objects//39/d5708913a6c708d1a505cde6da544785c086a6
.git/objects//63/063aaa7d1dad782e0349d897fcf33ff16a7e5d
.git/objects//d3/980efccf4856f0c61a6a16ed40be534c5230a5
.git/objects//d5/c7c4543f48ca7fb5cf75881fd0dbe21d75f51c
.git/objects//aa/02c633322970a4095e564aeb8fd96a3d4f31ab
.git/objects//b0/20e21272ba05cff974b1f789bdf5cfd2f5d355
.git/objects//f9/e70e5b8173525188a6b10ce5979972de4e0d9f
.git/objects//e0/f340c90f46a0103ee3dac9cfdc5520e4848731
.git/objects//79/8561538cfdec1f1da1eb1271f453fc06a40ca4
.git/objects//79/4a48e2f15fd84d1c60e2cf38ce0b4497e3f2ad
.git/objects//2d/232a2231c650dc4094606797fe0bd3e0ce4c65
.git/objects//1e/2a45fbca5ce7bf775827f5f4dbe23b6561cff4
.git/objects//1e/97f0d64fc62f40bbeb0eba19fa696fe47b7ccc
.git/objects//8c/3cc97ca6446c270ca0b8f7d4ce640a6e81e468
.git/objects//76/2f01b5cf84f39096d55ca95e46f0519d8cae48
.git/objects//2b/95364a88284fac03923db4408737c65d842465
.git/objects//7a/1650f9ccb43520716b69be9da7142beeade018
.git/objects//8e/c01b21b8d76092341a9b91e067f3526c3d45db
```

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

We can see that by looking at the working directory with `ls`

Skip to main content

```
ls
```

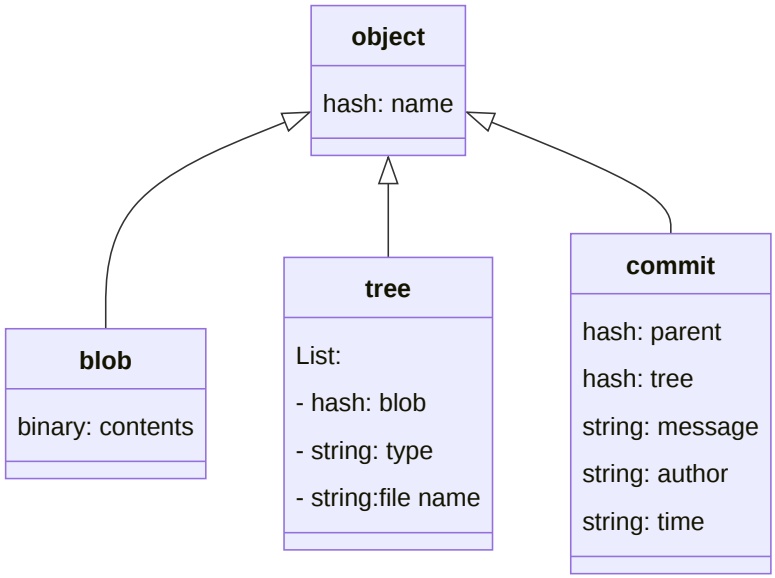
```
API.md          README.md       example.md      scratch.ipynb
CONTRIBUTING.md about.md        my_sectrest     setup.py
LICENSE.md      docs            pkg             tests
```

This is a consequence of git taking snap shots and tracking both the actual contents of our working directory **and** our commit messages and other meta data about each commit.

10.6.1. 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



Let's inspect the last item from the list of objects above,

Mine was:

```
.git/objects//8e/c01b21b8d76092341a9b91e067f3526c3d45db
```

so the minimum 4 characters that I can use to check what type it is 83c0.

```
git cat-file -t 8ec0
```

```
tree
```

Skip to main content

```
git cat-file -p 8ec0
```

```
040000 tree 95b60ce8cdec1bc4e1df1416e0c0e6ecbd3e7a8c    .github
100644 blob 3a4533a3abbc749f5e1905b30eb187a7350ae71a    API.md
100644 blob 9ece5efa25710c8fad7d9f210928785b5362b06f    CONTRIBUTING.md
100644 blob 2d232a2231c650dc4094606797fe0bd3e0ce4c65    LICENSE.md
100644 blob c8dc2d1d10e6c27cad4f3be278016f3cf90ac1ca    README.md
100644 blob 1e97f0d64fc62f40bbeb0eba19fa696fe47b7ccc    _config.yml
100644 blob e0f340c90f46a0103ee3dac9cfdc5520e4848731    _toc.yml
100644 blob 9ea743825de328435ce38670898e4109619a4d15    about.md
100644 blob 9d6ffa6ded47d8b6df13ed60e482b188015ee499    abstract_base_class.py
100644 blob 762f01b5cf84f39096d55ca95e46f0519d8cae48    alternative_classes.py
100644 blob 93c08483f44ebdf5ce10e6c0002e641aa0cc8844    example.md
100644 blob f9e70e5b8173525188a6b10ce5979972de4e0d9f    helper_functions.py
100644 blob 762f01b5cf84f39096d55ca95e46f0519d8cae48    important_classes.py
100644 blob 3c7ace1d193b78129c10d61609cbe97b2686f202    philosophy.md
100644 blob d87bf4a5641e0429fd3c371bd2b19d755105ca92    scratch.ipynb
100644 blob 39d5708913a6c708d1a505cde6da544785c086a6    setup.py
100644 blob 1f6ef3acccb91bc0c33661854903ec5e7d666fb5    test_abc.py
100644 blob 63063aaa7d1dad782e0349d897fcf33ff16a7e5d    tests_alt.py
100644 blob abe403c82238c3877b3437c1ab3e32499a60e24f    tests_helpers.py
100644 blob b020e21272ba05cff974b1f789bdf5cfd2f5d355    tests_imp.py
```

I see the tuples: type, hash of content, file name of the file for each item.

10.7. Git Variables

the program `git` does not run continously the entire time you are using it for a project. It runs quick commands each time you tell it to, it's goal is to manage files, so this makes sense. This also means that important information that `git` needs is also saved in files.

We can see the files that it has by listing the directory:

```
ls .git
```

```
COMMIT_EDITMSG  REBASE_HEAD     index           packed-refs
FETCH_HEAD      config          info            refs
HEAD            description     logs
ORIG_HEAD       hooks           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 .git/HEAD
```

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

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

Skip to main content

other things.

> **Important**
>
> We did some exercises at this point in class, you can revieew the questions on prismia

We saw the local HEAD already, so we can also compare to the origin manually, by using `FETCH_HEAD` which is updated on each fetch or pull

> **Important**
>
> This is corrected relative to what was said in class, I had previously used a different source that called `ORIG_HEAD` as related to origin, but it is orig as in original. It is created when you move the head dramatically and holds the previous value before that change is done so you can go back. source

```
cat .git/FETCH_HEAD
```

```
e3b192aa0cd490226e8adcd81d3d0b95adb5676b
```

but we see that origin head is to a commit, not a branch, so we have to follow the pointer from the `HEAD` to see what commit it was at.

So, remember that branches are pointers and look at what commit the current branch points to

```
cat .git/refs/heads/organization
```

```
e3b192aa0cd490226e8adcd81d3d0b95adb5676b
```

here we can now see they are the same currently, but this is not necessarily true, it can be pointed at even different branches so these are often not the same.

10.8. Collaboration Tools

Let's look at the gh inclass repo in browser

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

```
Opening github.com/compsys-progtools/gh-inclass-sp24-brownsarahm in your browser.
```

We added a collaborator to each of your gh inclass repos and followed the steps in the "add a collaborator" issue.

If you are making up class, ping @brownsarahm in a comment on the issue to get assigned a partner.

10.9. Prepare for next class

Create a file `gitcommandsbreakdown.md` and for each command in the template below break down what steps it must do based on what we have learned so far about git objects. I started the first one as an example based on what we discussed today, fill in the others. In class on Thursday, we will make a commit using plumbing commands, so thinking about what you already know about commits will prepare you to learn this material.

```
# What git commands do

## `git status`

- check the branch of the HEAD pointer
- compare the HEAD pointer to the FETCH_HEAD, if different trace back through parent commits to find out
- compare the snapshot at the HEAD pointer's commit to the current working directory
- if staging is not empty, compare that to the working directory

## `git commit`

-

## `git add`

-
```

10.10. Badges

Review | **Practice**

1. Read about different workflows in git and describe which one you prefer to work with and why in favorite_git_workflow.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. In commit_contents.md, redirect the content of your most recent commit, its tree, and the contents of each tree and blob in that tree to the same file. Edit the file or use `echo` to put markdown headings between the different objects. Add a title `# Complete Commit` to the file and at the bottom of the file add `## Reflection` subheading with some notes on how, if at all this excercise helps you understand what a commit is.

10.11. Experience Report Evidence

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

Skip to main content

10.12. Questions After Today's Class

10.12.1. Why is python so hard to install on a computer?

The complexity is that there can be multiple versions and there are multiple ways to do things.

10.12.2. What is a pack?

a pack is a compressed chunk of content created when you push or occasionally if you have a lot of content.

You may have seen one after running the command, `find .git/objects/ -type f`. It would look something like: `.git/objects/pack/pack-ef31f9fc1517ed75ff4829e12728cdc8884be12e.idx`

you can read more in the git internals chapter of the book

10.12.3. What are the exact differences between HEAD, ORIG_HEAD, and FETCH_HEAD?

All three are pointers to a commit, (or branch)

- `HEAD` is the locally checked out branch (or commit).
- `ORIG_HEAD` is where your head was before the last operation. source and `FETCH_HEAD` is the last head you checked out.

10.12.4. Why hasn't a more user friendly interface been implemented for git?

There are a lot of attempts, but they are all severely limited in function and generally pretty strongly disliked by people who understand how git works. I would not teach any of them because I have seen that they reinforce misconceptions and incorrect understanding of git in ways that prevent people from learning the more advanced features.

Also, user friendly is relative. Relative to the audience and to the alternative. To experienced devs familiar with other VCS, git *is* user friendly.

10.12.5. How does git know what files are in the staging area and what files are not?

The index file, which is not easily human readable, contains the contents of the staging area. We will learn more about this next class. This is a **great** questions

10.12.6. When we switch branches how does it update that list of files in the staging area?

No, the staging area is separate from branches. This is useful though. For example, you could add files to the staging area, then use git status and realize you are stil on main when you wanted to commit to a branch. You can then create and switch to

10.12.7. if github didn’t exist, where would git be used?

Other popular git hosts are bitbucket and gitlab, but there are lots and you can host your own.

this is a good explore topic

10.12.8. Does the HEAD just point to the most recent commit for the entire repository?

not always, but most of the time yes. It points to what is currently checked out.

We will be working with this more over the next few classes, so this is okay to be a little unclear because there will be more time to work with this.

11. How does git make a commit?

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

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

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

this is also to serve as an example method you could apply in understanding another complex system

Inspecting a system’s components is a really good way to understand it and correctly understanding it will impact your ability to ask good questions and even look up the right thing to do when you need to fix things.

Also, looking at the parts of git is a good way to reinforce specific design patterns that are common in CS in a practical way. This means that today we will also:

- review and practice with the bash commands we have seen so far
- see a practical example of hashing
- reinforce through examples what a pointer does

navigate to your github inclass repo

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

For example:

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

11.1. Creating a repo from scratch

We will start in the top level course directory.

```
cd inclass/systems
ls
```

```
gh-inclass-sp24-brownsarahm     tiny-book
kwl-sp24-brownsarahm
```

Yours should also have your kwl repo, gh inclass repo, course website clone, etc.

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

Last time we used an existing directory like `git init .` because we were working in the directory that already existed

Today we will create a new directory called `test`

```
git init test
```

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

we get this message again, see context from last week

Some people did not get the message

We can see what it did by first looking at the working directory

```
ls
```

```
gh-inclass-sp24-brownsarahm     test
kwl-sp24-brownsarahm            tiny-book
```

it made a new folder named as we said

and we can go into that directory

Skip to main content

```
cd test/
```

and then rename the branch

```
git branch -m main
```

To clarify we will look at the status

```
git status
```

Notice that there are no commits, and no origin.

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

Some people did not get the error message, so I decided to investigate. The message says that the warnign will only show if you do not set a default. So it is possible that some students set that.

-

First, though we checked to ensure that no onr was on an older version

```
git --version
```

```
git version 2.41.0
```

```
ls
```

it has no files

but with `-a`

```
ls -a
```

we see it does have the .git folder

```
.       ..      .git
```

```
ls .git
```

Skip to main content

```
config          hooks           objects
```

we can see the basic requirements of an empty repo here.

11.2. Searching the file system

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

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

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

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

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

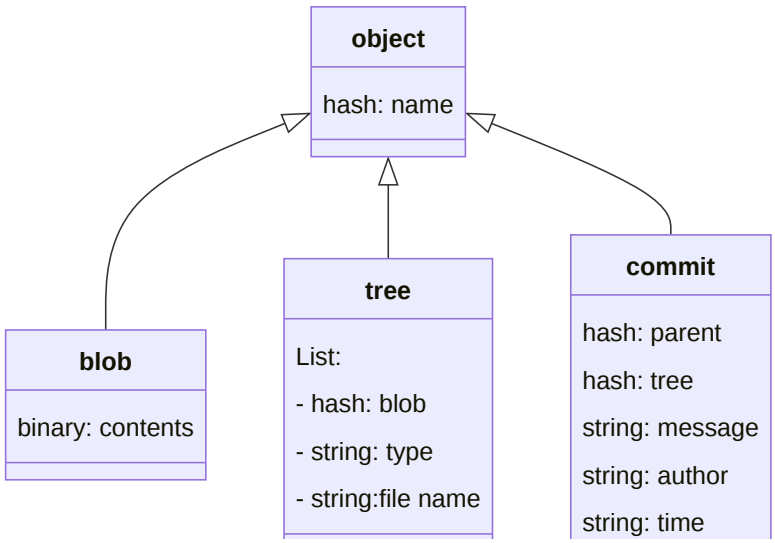
```
find .git/objects/ -type f
```

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

11.3. Git Objects

Remember our 3 types of objects

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



Skip to main content

11.3.1. How to create an object

All git objects are files stored with the name that is the hash of the content in the file

Remember git is a content-addressable file systsem… so it uses key- value pairs.

Let's create our first git object. git uses hashes as the key. We give the hashing function some content, it applies the algorithm and returns us the hash as the reference to that object. We can also write to our .git directory with this.

The `git hash-object` command works on files, but we do not have any files yet. We can create a file, but we do not have to. Rememmer, **everything** is a file.

When we use things like `echo` it writes to the stdout file.

```
echo "test content"
```

```
test content
```

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

pipes are an important content too. we're seeing them in context of real uses, and we will keep seing them. Pipes connect the std out of one command t othe std in of the next.

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

We can break down this command:

- git hash-object would take the content you handed to it and merely return the unique key
- `--stdin` option tells git hash-object to get the content to be processed from stdin instead of a file
- the `|` is called a pipe (what we saw before was a redirect) it pipes a *process* output into the next command
- `echo` would write to stdout, withthe pip it passes that to std in of the `git-hash`

we get back the hash:

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we can check if it wrote to the directory.

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

and it shows nothing

Now let's run it again with a slight modification. `-w` option tells the command to also write that object to the database

Skip to main content

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

we get the same hash back again

```
d670460b4b4aece5915caf5c68d12f560a9fe3e4
```

and we can check if it wrote to the directory.

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

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

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

11.3.2. Viewing git objects

We can try with `cat`

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

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

This is binary output that we cannot understand. Fortunately, git provides a utility. We can use `cat-file` to use the object by referencing at least 4 characters that are unique from the full hash, not the file name. (`7046` will not work, but `d670` will)

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

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

```
blob
```

we see that it is a blob.

Then we can do it with the `-p` option instead to see the content

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

```
test content
```

where we see the content we put in to the hashing function

11.3.3. Hashing a file

let's create a file

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

and store it, by hashing it

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

```
83baae61804e65cc73a7201a7252750c76066a30
```

we can look at what we have.

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

we see two objects as expected

```
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Now this is the status of our repo.

d67046
+ "test content"
+(blob)

83baae
+ Version 1
+(blob)

We can check the type of files with `-t` and `git cat-file`

```
git cat-file -t 83baa
```

```
blob
```

it is a blob object as expected

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

```
ls
```

```
test.txt
```

Skip to main content

it is the one test.txt, the first blob we made had no file in the working directory associated to it.

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

```
git status
```

```
On branch main

No commits yet

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

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

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

11.3.4. Updating the Index

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

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

Again, we check in with status

```
git status
```

```
On branch main

No commits yet
```

Skip to main content

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

We have the files staged as expected

11.3.5. A file can have multiple statuses

Now the file is staged.

Let's edit it further.

```
echo "version 2" >> test.txt
```

We can look at the content to ensure it as expected

```
cat test.txt
```

```
version 1
version 2
```

So the file has two lines

Now check status again.

```
git status
```

```
On branch main

No commits yet

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

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

We added the first version of the file to the staging area, so that version is ready to commit but we have changed the version in our working directory relative to the version from the hash object that we put in the staging area so we *also* have changes not staged.

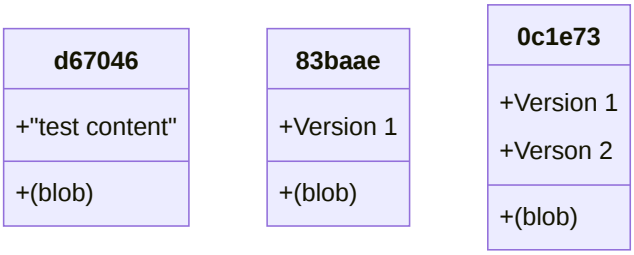
We can hash and store this version too.

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

We can then look again at our list of objects.

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```



So now our repo has 3 items, all blobs

```
git status
```

```
On branch main

No commits yet

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

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

hashing the object does not impact the index, which is what git status uses

11.3.6. Preparing to Commit

When we work with porcelain commands, we use add then commit. We have staged the file, which we know is what happens when we add. What else has to happen to make a commit.

We know that commits are comprised of:

- a message
- author and times stamp info
- a pointer to a tree
- a pointer to the parent (except the first commit)

We do not have any of these items yet.

Let's make a tree next

Skip to main content

Now we can write a tree from the index,

```
git write-tree
```

```
d8329fc1cc938780ffdd9f94e0d364e0ea74f579
```

and we get a hash

Lets examine the tree, first check the type

```
git cat-file -t d832
```

```
tree
```

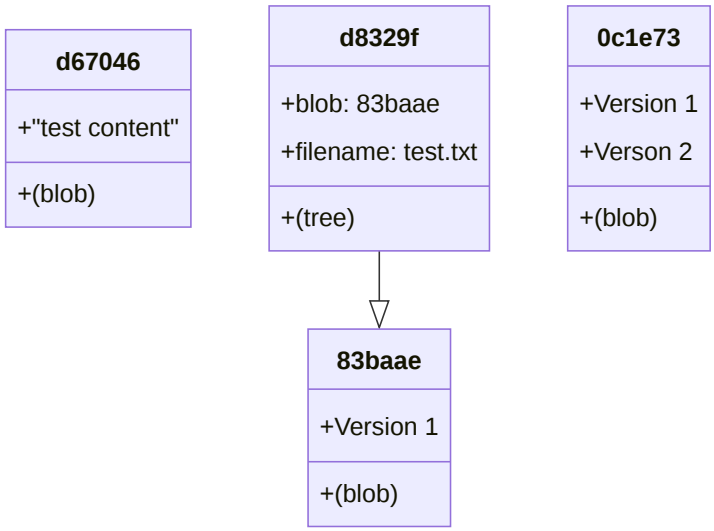
it is as expected

and now we can look at its contents

```
git cat-file -p d832
```

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

Now this is the status of our repo:



Again, we will check in with git via `git status`

```
git status
```

```
On branch main
```

Skip to main content

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

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

Nothing has changed, making the tree does not yet make the commit

11.3.7. Creating a commit manually

We can echo a commit message through a pipe into the commit-tree plumbing function to commit a particular hashed object.

the `git commit-tree` command requires a message via stdin and the tree hash. We will use stdin and a pipe for the message

```
echo "first commit" | git commit-tree d832
```

```
e3ba10cb02de504d4f48b9af4934ddcc4d0be3df
```

and we get back a hash. But notice that this hash is unique for each of us. Because the commit has information about the time stamp and our user.

We can also look at its type

```
git cat-file -t e3ba
```

```
commit
```

and we can look at the content

```
git cat-file -p e3ba
```

```
tree d8329fc1cc938780ffdd9f94e0d364e0ea74f579
author Sarah M Brown <brownsarahm@uri.edu> 1709231797 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1709231797 -0500

first commit
```

Now we check the final list of objects that we have for today

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

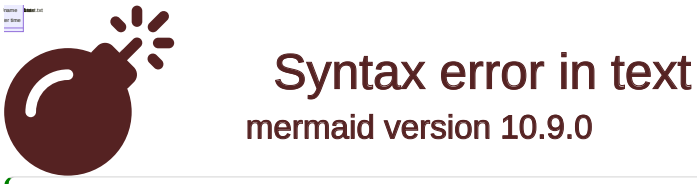
```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
```

Skip to main content

```
.git/objects//e3/ba10cb02de504d4f48b9af4934ddcc4d0be3df
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

> **Important**
>
> Check that you also have 5 objects and 4 of them should match mine, the one you should not have is the `e3ba10` one but you should have a different one in its place.

Visually, this is what our repo looks like:




```
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
```

11.4. What does git status do?

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

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

```
git status
```

we see it is “on main” this is because we set the branch to main , but since we have not written there, we have to do it directly. Notice that when we use the porcelain command for commit, it does this automatically; the porcelain commands do many things.

Notice, git says we have no commits yet even though we have written a commit.

In our case because we made the commit manually, we did not update the branch.

Skip to main content

We will pick up from here next class.

11.5. Prepare for this class

1. Read the notes from February 29th. We will build on these directly in the future. **You need to have the `test` repo with the same status for lab on class on 3/5** Make sure you have completed all of the steps in the github inclass repo.

11.6. Badges

Review | Practice

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plubming and porcelain. The table should have two columns, one for each type of command (plubming and porcelain). Each row should have one git porcelain command and at least one of the corresponding git plumbing command(s). Include two rows: `add` and `commit`.

11.7. 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.

11.8. Questions After Today's Class

11.8.1. After finalizing the repo, how do we go about uploading it to GitHub? Do we push it like we would new content for cloned local repos?

You could add a remote and then push; we will not push this one, but that is possible.

11.8.2. Is there any benefit to using git like this over how we normally use it with git add and git commit?

It is useful for edge cases and inspection. It will not be the way you use it day to day, but it would be useful, if you were working on an applicaiton the built on top of git. It is also helpful to go through this way to see how git really works.

11.8.3. What would happen if a commit is made and a blob or tree object is later deleted?

This is a good explore badge idea. My guess is that it would not do anything immediately, but if you tried to check out that commit it would have an issue and complain about a corrupted repo.

11.8.4. what does the git index do I'm slightly confused about the functionality of this?

The index is the staging area, it is where we store information necessary to make a commit while we get ready to make a commit.

11.8.5. How are hashes made?

git uses the SHA-1 hardened algorithm. We will learn more about that algorithm soon.

11.8.6. Why do four of the objects match for everyone?

The core objects are all the same, so the hashes are the same because the hashes are a function of the content. Only the commit has unique information in it.

11.8.7. using cat on the "test object" gave me `xK▒▒OR04f(I-.QH▒▒+I▒+▒K▒`, your computer output the same thing but rendered the `▒` as `?` and other actual symbols, why?

They are both binary content, but we are using different terminal applications that attempt to render it using different character sets.

11.8.8. did any of what we did today synch with remote servers on github?

No, this repo has no remotes.

11.8.9. How do we upload the new repository and connect it to others

add a remote. like how we did with our tinv-book

Skip to main content

11.8.10. Why is it so hard to have a repositry inside of another one?

One is that it is often just not what you want.

Two then you have to manage which repo git is working on when you commit.

Learning about uses and best practices for git submodules is a good explore.

11.8.11. I thought we psuh into remote before we staged the content?

We can only push content that hash been added to the .git directory, so without staging, we would't have the tree.

11.8.12. Do remote repositories like github store the repo in a similar .git folder

yes! that is the content that gets compressed and sent back and forth.

11.8.13. Why would manually doing the commands be useful or is this just to help us understand how they work?

Mostly just to understand, but plumbing commands can be helpful in fixing if weird stuff happens.

11.8.14. What happens if there is a collision with hashes? Especially if you have a massive repository with thousands of files?

(we will come back to this later)

git used SHA-1 initially which is vulnerable to collision attacks, so now it uses a hardened one, it knows which are possible to collide andif one of those occurs, it changes.

git is working toward changing to SHA-256 to make collisisons much less likely.

11.8.15. Why would we hash an object but not add it to a regular commit?

We would not typically do this on purpose, but we do hash and store objects when we use `git add` before they are added to the index, they must be hashed. So if you do atypical things at that point you could end up with something hashed but not linked to a commit, long term instead of only temporary.

12. Why did we learn the plubming commands?

> **Important**
>
> If you miss class or lab for any reason you are responsible for making it up per the instructions in the syllabus. You do not need to notify me why you are missing class and I do not judge your reasons for missing class for short absences. For longer absences you may email me to make a plan.
>
> For lab makeup this includes attending office hours for the checkout step.
>
> For class makeup this includes an alternative path to your experience badge. Note that in class actitivies may build, so it is important to review the notes ASAP so that you are ready to participate in the next class you attend.

You will not typically use them on a day to day basis, but they are a good way to see what happens at the interim steps and make sure that you have the right understanding of what git does.

A **correct** understanding is essential for using more advanced features

While there is of course some content that we want you to know after this course, my goal is also to teach you process, by modeling it.

No one will every know all of the things but you can be fast or slow at finding answers.

And you can find correct answers, incorrect answers, or looks-okay-but-you-will-regret-this-later answers.

My goal is that you get good at *quickly* finding **correct** answers.

Large language models will not do that for you.

Note I checked and confirmed that some of the assignments will be answered incorrectly by LLMs because they are about very specific details that most people who write blogs and code excerpts on stack overflow etc do incorrectly.

12.1. Today's Questions

- What are references?
- How can can I release and share my code?
- How else can git help me?

12.2. What does git status do?

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

- we cansee the working directory with: `ls`
- we can see the active branch in the `HEAD` file

Skip to main content

Recall that we:

- crated a blob objct directly
- createaed a file
- hashed the file, to create its blob object
- added the file to the index
- wrote the tree
- created a commit

Let's use inspection to review where our repo is left off from last week.

```
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 that git status still thinks the file is only staged, but not committed.

From this we learn that index is not emptied when we create a commit.

We see that the file is both staged and has changes not staged because we edited the file after we had staged it.

This reminds us that the staging area is manual, like all git operations. git does not run continously, which makes it faster and less resource intensive.

We also note that this says we are on the main branch , and that we have no commits yet. Let's try to understand why it says we have no commits when we thougth we made one

12.3. Branches are references

branches are not git objects, they are references.

We can see that by where they are stored.

How can we inspect to see where references are stored?

using ls and/or find

Skip to main content

12.3.1. Why is our commit missing?

We can see our list of object with `find`

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

```
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//d6/70460b4b4aece5915caf5c68d12f560a9fe3e4
.git/objects//d8/329fc1cc938780ffdd9f94e0d364e0ea74f579
.git/objects//e3/ba10cb02de504d4f48b9af4934ddcc4d0be3df
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

Since we made most of these objects as not commits, the hashes are mostly shared, but one of the hashes is unique so we worked together to identify that one.

Then for that unique hash, we confirmed it was a commit using `git cat-file` to view its type

```
git cat-file -t e3ba
```

```
commit
```

as expected

Now let's look at what the `HEAD` pointer says to try to understand why it does not see that commit, since we know that git status works from `HEAD`

```
cat .git/HEAD
```

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

this is consistent with the `git status` said, so this is not yet explaining

now we can check what the branch points to

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

as expected when the tab complete did not work

```
cat: .git/refs/heads/main: No such file or directory
```

it has not even made a file to track what commit the main branch points to.

Skip to main content

Important

git creates updates the file for the branch each time you add a commit with `git commit`, the first time `git commit` is run it also creates the file

If we look in the folder, we can see what is in there

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

but it is empty

12.3.2. Updating a branch manually

The branch file is named as the branch (here `main`) and stored in the `.git/refs/heads/` folder and contains the full hash of the commit that branch is pointing to.

Since we made the commit manually, we need to move the branch manually too. We will use echo, by copyting the full hash from above (I copied the part after the last `/` in the list of objects above and then typed the `e3` before pasting).

Mine looks like:

```
echo e3ba10cb02de504d4f48b9af4934ddcc4d0be3df > .git/refs/heads/main
```

but your commit hash will be different than mine.

Now we check with git again:

```
git status
```

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

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

and now we can see that it no longer sees the staged file and *does* see our commit.

So we now have HEAD-> main and main -> our commit -> tree –> blob.

12.4. Git tags

The other type of git reference is called a tag. There are two types of tags:

- a lightweight tag is like a branch that does not move

Skip to main content

We can see where they are stored in the `refs` folder:

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

```
heads   remotes tags
```

Remember:

- heads are branches (like `main`)
- remotes are servers that we can push to (eg `https://github.com/...`)

Before we make a tag, let's make sure we are on the most up to date main. First we check the branch and working directory:

```
git status
```

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

nothing to commit, working tree clean
```

main, clean working tree is good.

Next we pull to update from github

```
git pull
```

```
remote: Enumerating objects: 4, done.
remote: Counting objects: 100% (4/4), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 1), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), 955 bytes | 136.00 KiB/s, done.
From https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm
 * [new branch]      1-create-a-add-a-classmate -> origin/1-create-a-add-a-classmate
Already up to date.
```

and we get the updates

Now, to make a tag we use `git tag <name of tag>` to create a lightweight tag

```
git tag v1
```

This command has no output, but we will inspect

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

```
v1
```

Skip to main content

we see this created a file for the tag,

we will look at this too

```
cat .git/refs/tags/v1
```

```
0d6ef1d3db0729088d515b35b588f39af5b770fd
```

the lightweight tag is just like a branch, another pointer to the commit.

The difference is that this does not move when we create new commits, so this is like a shortcut to a specific commit.

We can see the tag relative to branches in the log as well:

```
git log
```

```
commit 0d6ef1d3db0729088d515b35b588f39af5b770fd (HEAD -> main, tag: v1, origin/main, origin/HEAD)
Merge: 9f39946 e3b192a
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 8 13:43:19 2024 -0500

    Merge pull request #6 from compsys-progtools/organization

    first pass organizing

commit e3b192aa0cd490226e8adcd81d3d0b95adb5676b (origin/organization, organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:41:53 2024 -0500

    oranizizng

commit 260c9c309922970f80bfa2c93cc23bcfbb962740
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:06:20 2024 -0500

    describe the files

commit 29ffc88519103085ed3a2ab01cffb3c99d70fc6a
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:59:20 2024 -0500
```

If we push:

```
git push
```

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

we see nothing happens initially.

Tags have to be pushed explicitly

```
```

Skip to main content

```
Total 0 (delta 0), reused 0 (delta 0), pack-reused 0
To https://github.com/compsys-progtools/gh-inclass-sp24-brownsarahm.git
 * [new tag]         v1 -> v1
```

then it sends the information to GitHub

Next we looked at how tags are represented on github. After opening the repo up in browser:

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

Tags enable releases, where are defined on github as:

> Releases are deployable software iterations you can package and make available for a wider audience to download and use.

We can create a relese for the tag we made.

12.5. Debugging with git

git will help you do a binary search to find what commit introduced a bug.

we have to tell it to start, then tell it that the current commit is bad, then tell it the last good commit (or a reference to it)

To do this we need to have to tell it what commit was good, this is one of the reasons that using tags can be helpful

there are multiple ways to refer to a point in the history

revision selection

First let's choose a point to use as our last good commit by scanning the previous commits.

```
git log
```

```
commit 0d6ef1d3db0729088d515b35b588f39af5b770fd (HEAD -> main, tag: v1, origin/main, origin/HEAD)
Merge: 9f39946 e3b192a
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 8 13:43:19 2024 -0500

    Merge pull request #6 from compsys-progtools/organization

    first pass organizing

commit e3b192aa0cd490226e8adcd81d3d0b95adb5676b (origin/organization, organization)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:41:53 2024 -0500

    oranizizng

commit 260c9c309922970f80bfa2c93cc23bcfbb962740
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 13:06:20 2024 -0500
```

Skip to main content

```
    describe the files

commit 29ffc88519103085ed3a2ab01cffb3c99d70fc6a
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:59:20 2024 -0500

    add to readme

commit 9f399466ad6a1ad572e104209f4469eb8cd48516
Merge: 1e2a45f fca59e8
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:43:18 2024 -0500

    Merge pull request #5 from compsys-progtools/organizing_ac

    2/6 in class activity

commit fca59e8cca05bb0861f9348a40fe8300b3d55637 (origin/organizing_ac)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 11:21:21 2024 -0500

    add files for organizing

commit 1e2a45fbca5ce7bf775827f5f4dbe23b6561cff4 (my_branch_cehckoutb, my_branch)
Merge: faef6af 81c6f18
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 1 12:51:17 2024 -0500

    Merge pull request #4 from compsys-progtools/2-create-an-about-file

    create about file close s #2

commit 81c6f187f146caaaf43d97bc1bb8ed237142f4c3 (origin/2-create-an-about-file, 2-create-an-about-file)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Jan 30 13:33:54 2024 -0500

    create about file close s #2
:

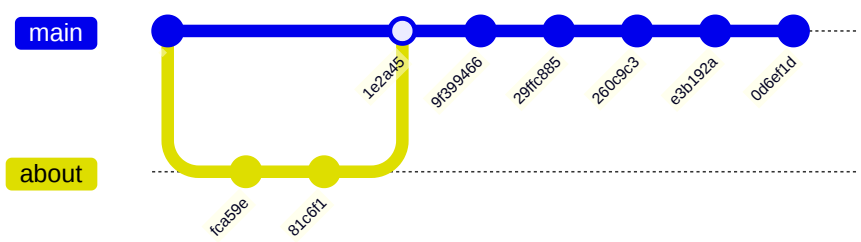
    2/6 in class activity

commit fca59e8cca05bb0861f9348a40fe8300b3d55637 (origin/organizing_ac)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 11:21:21 2024 -0500

    add files for organizing
```

Let's choose the commit where we left the `create-an-about` branch.

This log shows history like this



We will start `git bisect` first, this is our search for the "bad commit"

```
git bisect start
```

Skip to main content

```
status: waiting for both good and bad commits
```

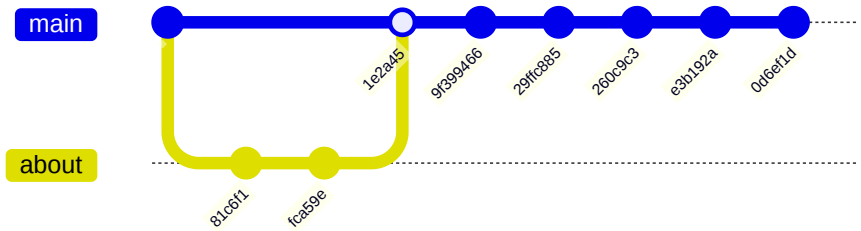
it tells us its progress at each step and what we need to do next, tell it the good and bad commits

Now, lets tell it that the current commit is bad, this reprsents that we just rcevied the bug report.

```
git bisect bad
```

```
status: waiting for good commit(s), bad commit known
```

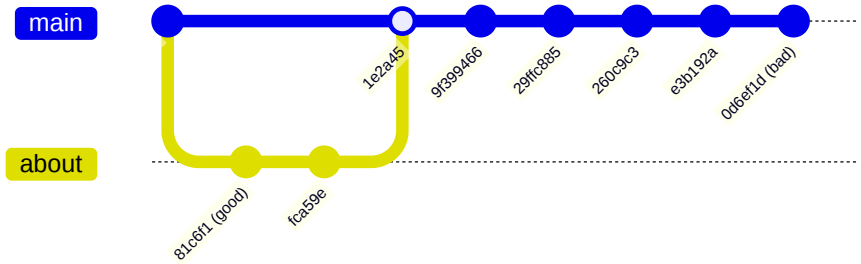
now we get the updated status, so now it has labeled the last commit on the main branch as bad



next we tell it what was the last known, not necessarily the last, good commit. We'll use the about branch commit. This would represent a case where maybe we switched to to that branch locally, ran a test for the the new bug and it passed.

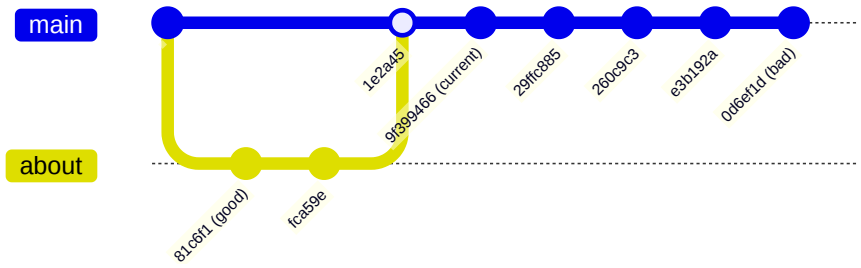
my commit was `81c6`, yours will be different

```
git bisect good 81c6
```



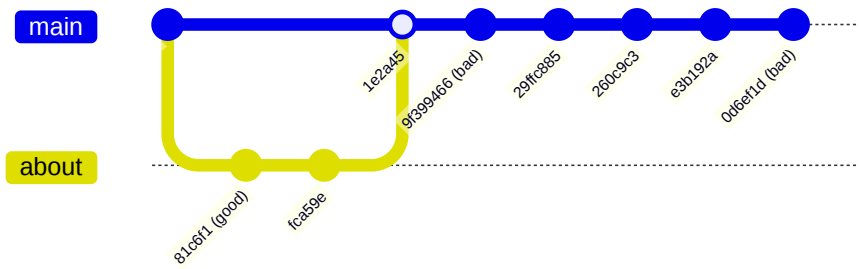

```
Bisecting: 3 revisions left to test after this (roughly 2 steps)
[9f399466ad6a1ad572e104209f4469eb8cd48516] Merge pull request #5 from compsys-progtools/organizing_ac
```

It tells us where it checked out



Skip to main content

```
git bisect bad
```



Now git labels all of the commits between the good commit and bad commit as bad:

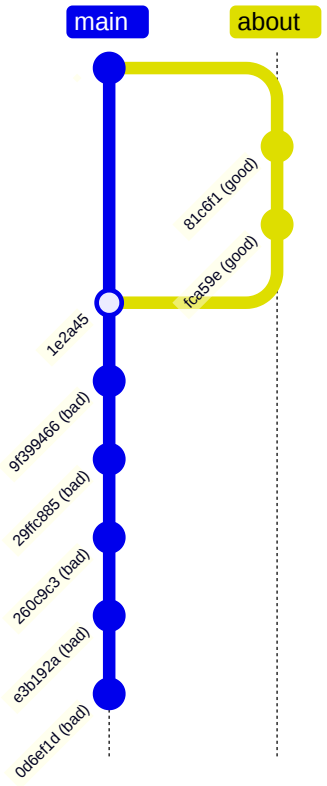
and it checks out the next commit we need to look at:

```
Bisecting: 0 revisions left to test after this (roughly 1 step)
[fca59e8cca05bb0861f9348a40fe8300b3d55637] add files for organizing
```

We will label this as good:

```
git bisect good
```

Skip to main content



Now git tells us the first bad commit, and a bunch of details about that first bad commit:

```
9f399466ad6a1ad572e104209f4469eb8cd48516 is the first bad commit
commit 9f399466ad6a1ad572e104209f4469eb8cd48516
Merge: 1e2a45f fca59e8
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Feb 6 12:43:18 2024 -0500

    Merge pull request #5 from compsys-progtools/organizing_ac

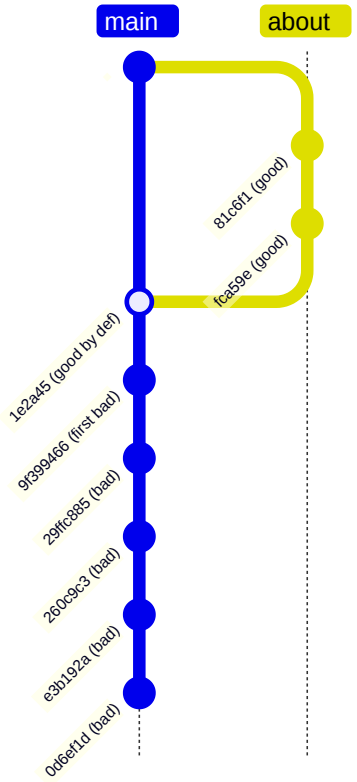
    2/6 in class activity

 API.md                 | 1 +
 CONTRIBUTING.md        | 1 +
 LICENSE.md             | 1 +
 _config.yml            | 1 +
 _toc.yml               | 1 +
 abstract_base_class.py | 1 +
 alternative_classes.py | 1 +
 example.md             | 1 +
 helper_functions.py    | 1 +
 important_classes.py   | 1 +
 philosophy.md          | 1 +
 scratch.ipynb          | 1 +
 setup.py               | 1 +
 test_abc.py            | 1 +
 tests_alt.py           | 1 +
 tests_helpers.py       | 1 +
 tests_imp.py           | 1 +
 17 files changed, 17 insertions(+)
 create mode 100644 API.md
 create mode 100644 CONTRIBUTING.md
 create mode 100644 LICENSE.md
 create mode 100644 _config.yml
 create mode 100644 _toc.yml
 create mode 100644 abstract_base_class.py
 create mode 100644 alternative_classes.py
```

Skip to main content

```
 create mode 100644 important_classes.py
 create mode 100644 philosophy.md
 create mode 100644 scratch.ipynb
 create mode 100644 setup.py
 create mode 100644 test_abc.py
 create mode 100644 tests_alt.py
 create mode 100644 tests_helpers.py
 create mode 100644 tests_imp.py
```

Note our final labeling of commits is:



We did not need to inspect the merge commit because git could tell it was only a merge commit and did not add any changes to files; if it had been a merge conflict resovled with the merge commit, we would have had to check it.

Here we can then inspect what changes were made and figure out how to fix the bug.

Let's dig a bit futher into what the current status is.

```
git status
```

```
HEAD detached at fca59e8
You are currently bisecting, started from branch 'main'.
  (use "git bisect reset" to get back to the original branch)

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

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

Skip to main content

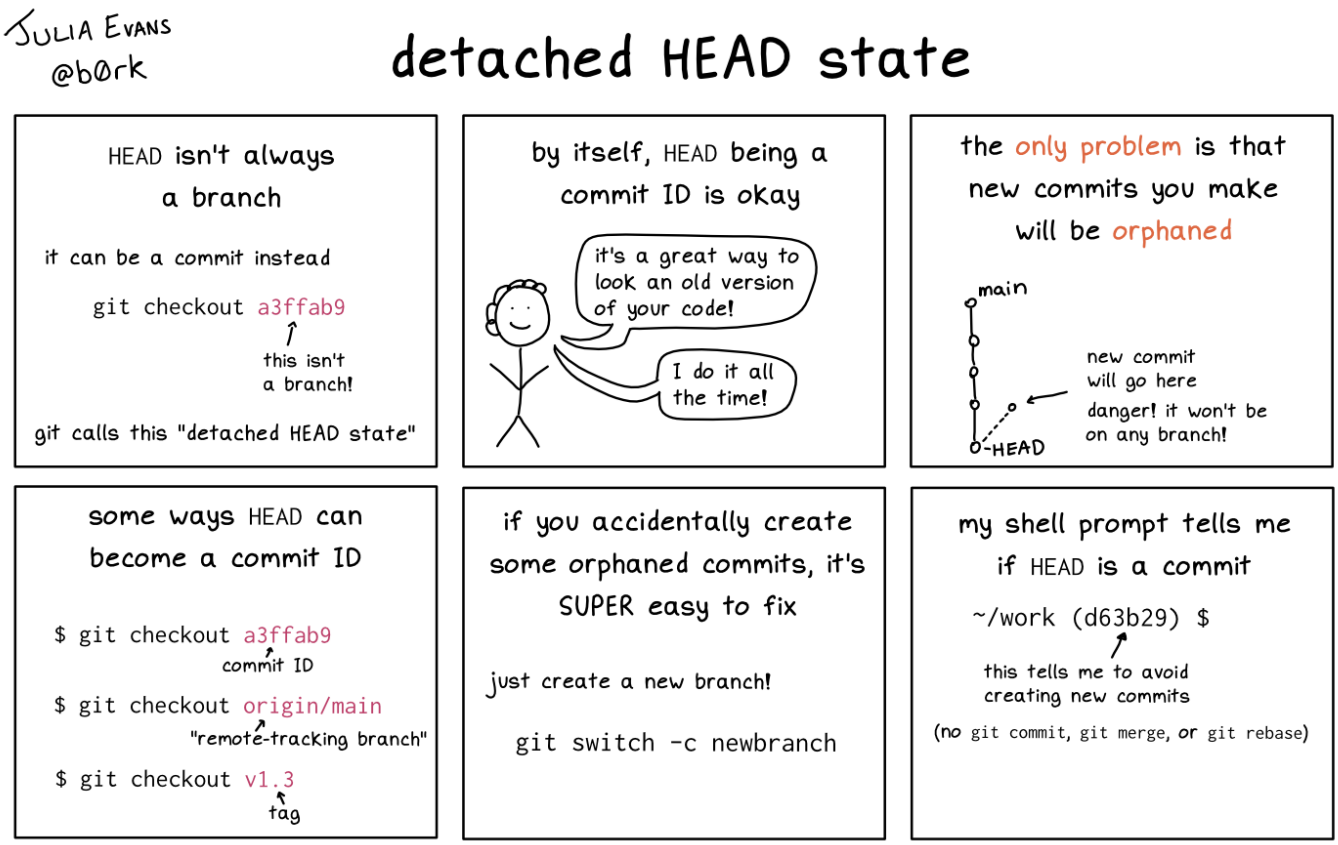


Fig. 12.1 A comic from wizard zines that has detailed alt text at the “Read the transript” link on the source page.

We also see some untracked files. This is because at the tip of the main branch, where we were previously, these files existed and were listed in the `.gitignore`

We can see more about what bisect did by looking in the .git directory

```
ls .git/
```

```
BISECT_ANCESTORS_OK     FETCH_HEAD              index
BISECT_EXPECTED_REV     HEAD                    info
BISECT_LOG              ORIG_HEAD               logs
BISECT_NAMES            REBASE_HEAD             objects
BISECT_START            config                  packed-refs
BISECT_TERMS            description             refs
COMMIT_EDITMSG          hooks
```

it created files for all of the steps in the active bisect

Now we go back to whre we were before the biset

```
git bisect reset
```

Skip to main content

```
Previous HEAD position was fca59e8 add files for organizing
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

in this case, main

```
git status
```

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

nothing to commit, working tree clean
```

here we see the gitignroe is back

12.6. Any commit can be checked out

We can also check out the error prone commit direct if we need to go back to fix again.

```
git checkout fca59
```

```
Note: switching to 'fca59'.

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 fca59e8 add files for organizing
```

This way git gives us advice on how to handle our detached head.

12.7. Annotated commits create objects

Maybe, we do not know how to fix this , but we want to make the commit easy for a colleague to find.

```
git tag -a -m 'introduced imaginary bug' v0.1
```

this time we will use `-a` to make an annotated tag and pass the message for is with `-m`

Skip to main content

We can

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

```
v0.1    v1
```

we see it created a file like before

So we can look at the file again

```
cat .git/refs/tags/v0.1
```

```
6b459a1dcf4e2a0ecec1ec41b03d60af4043e9a3
```

but this time the file points to a different object, not the commit.

We can inspect this object with `git cat-file`:

```
git cat-file -t  6b45
```

```
tag
```

it is a new type: tag

and check its content

```
git cat-file -p  6b45
```

```
object fca59e8cca05bb0861f9348a40fe8300b3d55637
type commit
tag v0.1
tagger Sarah M Brown <brownsarahm@uri.edu> 1709663041 -0500

introduced imaginary bug
```

This is a lot like the content of a commit, but instead of pointing to a tree it points to the a commit.

12.8. How can git help with work in Progress?

git stash

What if you have worked on something part way, but not finished it, so you do not want to make a commit yet, but you need to switch and work on something else?

Skip to main content

```
git checkout main
```

```
Previous HEAD position was fca59e8 add files for organizing
Switched to branch 'main'
Your branch is up to date with 'origin/main'.
```

and create some work in progress:

```
echo 'not done yet'>>example.md
```

we will look at our file

```
cat example.md
```

```
myst notebook example of using the code for doc websitenot done yet
```

and check in 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:   example.md

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

Now lets consider that we have some other work we need to do. We also do not want to lose this work, or make a commit for it yet. What can we do?

This is what the git “stash” is for. Stash is a first

`git stash` stores temporary changes without making a commit so that we can come back to them.

```
git stash
```

```
Saved working directory and index state WIP on main: 0d6ef1d Merge pull request #6 from compsys-progtools
```

We can see the impact of this like :

```
git status
```

Skip to main content

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

nothing to commit, working tree clean
```

Then we can do something else

```
echo "important " >> about.md
```

check in again

```
git status
```

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

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

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

make a new branch

```
git checkout -b important_about
```

```
Switched to a new branch 'important_about'
```

and add

```
git add .
```

we'll check agian

```
git status
```

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

our stashed content is still not interfering

and then commit this

```
```

Skip to main content

```
[important_about c3bee12] add about improtatn fact
 1 file changed, 1 insertion(+)
```

now we can go back to main

```
git checkout main
```

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

we can apply our changes with `git stash apply`

```
git stash apply
```

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

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

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

and we get them back

```
git checkout important_about
```

```
M       example.md
Switched to branch 'important_about'
```

```
git status
```

```
On branch important_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:   example.md

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

```
git add .
```

```

```

```
Saved working directory and index state WIP on important_about: c3bee12 add about improtatn fact
```

```
git stash list
```

```
stash@{0}: WIP on important_about: c3bee12 add about improtatn fact
stash@{1}: WIP on main: 0d6ef1d Merge pull request #6 from compsys-progtools/organization
```

```
git status
```

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

```
git stash pop
```

```
On branch important_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:   example.md

no changes added to commit (use "git add" and/or "git commit -a")
Dropped refs/stash@{0} (4862b97ea83e0ddf04ca2344f33df90b1a78feb5)
```

```
git stash list
```

```
stash@{0}: WIP on main: 0d6ef1d Merge pull request #6 from compsys-progtools/organization
```

```
git add .
```

```

```

```
git status
```

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

```
git checkout main
```

```
M       example.md
```

Skip to main content

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

```
git status
```

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

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

```
## Prepare for Next Class

```{include} ../_prepare/2024-03-07.md
```

12.9. Badges

Review **Practice**

1. Use git bisect to find the first bad commit in the toy bug repo, save the command history and the bad commit hash to git_debug.md
2. Create tagtypeexplore.md with the template below. Determine how many of the tags in the course website are annotated vs lightweight using. (You may need to use `git pull --tags` in your clone of the course website)

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

12.10. Experience Report Evidence

12.11. Questions After Today's Class

12.11.1. Are tags only applicable to commits?

tags are defined relative to a commit.

Skip to main content
```

12.11.2. Is the stash diffrent between all the branches or it the same.

There is one stash per repo, it is a queue shared across branches

12.11.3. with git stash can I create a file on say main stash and then switch to another branch to move that file to that new branch instead?

Yes, sort of, you can create a while while main is checked out and then stash it instead of committing it. Since it gets stashed, not committed, the file is not “on” the branch.

THen you can switch branches and apply or pop from stash and then commit.

12.11.4. Is the point of stash area just in case you do work in the wrong branch?

No, it actually cannot help if you commit to the wron branch and if you do not commit, you do not need it.

It is so that you can save work without commiting, get a clean working directory to do something else and then add the work in .

12.11.5. while debugging and you accidently call a good commit a bad commit is there a way to reverse that or no?

It looks like you would have to start over, but you could print the log first to remember what you had already figured out.

12.11.6. Where does git stash store its stack, is it a file inside of the .git folder?

it hashes the content as blobs and stores the pointer to that blob in `.git/refs/stash`

12.11.7. how often are these commands used in a professional environment

this is hard to know, a lot of people are git novices, but I want you to have these as tools so you can be more efficient.

Being efficient with your time in a salaried job often means you can work fewer hours total or you can get more work done and earn promotions faster.

12.11.8. why wouldn’t i just make a new branch instead of leaving the changes floating in space?

One time to use stash is when you realize you started working without pulling. If you make a commit you will have to deal with the divergent branches, but if you stash, pull, then apply, no such problem.

Another case is that in some projects, there are standards to not have too many too small commits, so this is a way to avoid that

13. What is a commit number?

Important

- spring break is like a time pause (you get an extra week on things assigned last week and this week)

13.1. What is a hash?

a hash is:

- a fixed size value that can be used to represent data of arbitrary sizes
- the *output* of a hashing function
- often fixed to a hash table

Common examples of hashing are lookup tables and encryption with a cyrptographic hash.

A hashing function could be really simple, to read off a hash table, or it can be more complex.

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.

For example:

Hash	content
0	Success
1	Failure

In a more complex scenario, imagine trying to hash all of the new terms you learn in class.

A table would be hard for this, because until you have seen them all, you do not know how many there will be. A more effective way to hash this, is to derive a *hashing function* that is a general strategy.

A *cyrptographic* hash is additionally:

- unique
- not reversible
- similar inputs hash to very different values so they appear uncorrelated

Now lets go through each of these properties

13.1.1. Hashes are fixed length

This is good for memory allocation reasons.

13.1.2. Hashes are unique

This means that two different values we put in should give different results.

For this property alone, a simple function could work:

```
def basic_unique(input):
    return input
```

but this is not a hash because its length would not be constant and not a cryptographic has because it is easily reversible.

13.1.3. Hashes are not reversible

This means that given the hash, we cannot compute the input.

An example of a nonreversible function is modulus:

```
13%3
```

```
1
```

```
10%3
```

```
1
```

It can be any function that gives the same output for two (or more) values.

13.1.4. Similar inputs yeild uncorrelated outputs

We can see this using git:

```
echo "it's almost spring" | git hash-object --stdin
```

```
1327354a0d47111e361ec52acc46b9509bed69a0
```

and we get a hash

and then we change it just a little bit:

```
echo "it's almsot spring" | git hash-object --stdin
```

Skip to main content

```
f798473089b4a00684cd0f13a6f2d1d7a0e033ba
```

and now the hash is completely different

13.2. How can hashes be used?

Hashes can then be used for a lot of purposes:

- message integrity (when sending a message, the unhashed message and its hash are both sent; the message is real if the sent message can be hashed to produce the same has)
- password verification (password selected by the user is hashed and the hash is stored; when attempting to login, the input is hashed and the hashes are compared)
- file or data identifier (eg in git)

13.2.1. Hashing in passwords

Passowrds can be encrypted and the encrypted information is stored, then when you submit a candidate password it can compare the hash of the submitted password to the hash that was stored. Since the hashing function is nonreversible, they cannot see the password.

Some sites are negligent and store passwords unencrypted, if your browser warns you about such a site, proceed with caution and definitely do not reuse a password you ever use. (you *should never* reuse passwords, but especially do not if there is a warning)

An attacker who gets an encyrpted database, cannot actually read the passwords, but they could build a lookup table. For example, “password” is a bad password because it has been hashed in basically every algorithm and then the value of it can be reversed. Choosing an uncommon password makes it less likely that your password exists in a lookup table.

For example, in SHA-1 the hashing algorithm that git uses

```
echo "password" | git hash-object --stdin
```

```
f3097ab13082b70f67202aab7dd9d1b35b7ceac2
```

13.2.2. Hashing in Git

In git we hash both the content directly to store it in the database (.git) directory and the commit information.

Recall, when we were working in our toy repo we created an empty repository and then added content directly, we all got the same hash, but when we used git commit our commits had different hashes because we have different names and made the commits at different seconds. We also saw that *two* entries were created in the `.git` directory for the commit.

Git was originally designed to use SHA-1.

Skip to main content

Git switched to hardened HSA-1 in response to a collision.

> In that case it adjusts the SHA-1 computation to result in a safe hash. This means that it will compute the regular SHA-1 hash for files without a collision attack, but produce a special hash for files with a collision attack, where both files will have a different unpredictable hash. from.

they will change again soon

git uses the SHA hash primarily for uniuqeness, not privacy

It does provide some *security* assurances, because we can check the content against the hash to make sure it is what it matches.

This is a Secure Hashing Algorithm that is derived from cryptography. Because it is secure, no set of mathematical options can directly decrypt an SHA-1 hash. It is designed so that any possible content that we put in it returns a unique key. It uses a combination of bit level operations on the content to produce the unique values.

This means it can produce 2^160 different hashes. Which makes the probability of a collision very low.

> The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about 2^{80} (the formula for determining collision probability is $p = (n(n-1)/2) * (1/2^160)$). 2^{80}) is 1.2 x 1024 or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.
>
> —A SHORT NOTE ABOUT SHA-1 in the Git Documentation

13.3. Working 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

we'll see this in our github inclass repo

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

by using some options on `git log` to make it easier to read a set of them

```
git log --abbrev-commit --pretty=oneline
```

```
9f39946 (HEAD) Merge pull request #5 from compsys-progtools/organizing_ac
```

```
81c6f18 (origin/2-create-an-about-file, 2-create-an-about-file) create about file close s #2
faef6af start readme, closes #3
98cff65 Initial commit
```

For most project 7 characters is enough and by default, git will give you 7 digits if you use `--abbrev-commit` and git will automatically use more if needed.

13.4. How are hashes computed?

> **Important**
>
> I did not have this explicitly in class, but it was implied and I thought it was better to add it here

Hashes are computed on the binary representation of the data, most treat that as numbers and then perform mathematical operations.

you can see the psuedocode to SHA-1

we'll work a small example, Python can convert characters to ints:

```
ord('s')
```

```
115
```

and ints to binary

```
bin(ord('s'))
```

```
'0b1110011'
```

with some extra at the start, so if we just want the binary string

```
str(bin(ord('s')))[2:]
```

```
'1110011'
```

We can use this to make a function that performs an xor based hash.

```
def xor_hash(message,n):
    '''
    a simple hashing algorithm based on xor that can take a message of
    any length and return a version of it with a specific bit length n
    '''
    bitstring = ''.join([str(bin(ord(c)))[2:] for c in message])

    # zero pad to make it a multiple of n
    extra = len(bitstring)%n
    num_pad = (n-extra)*int(extra >0)
    padded_bitstring = bitstring + num_pad*'0'

    # split into length n chunks
    split_bitstring = [padded_bitstring[i:i+n] for i in range(0, len(padded_bitstring), n)]

    # now make them back to integers so we can xor
    bin_int_strings = [int("0b" + bits,2) for bits in split_bitstring]

    # now xor
    out_hash_int = 0
    for n_int in bin_int_strings:
        out_hash_int = out_hash_int^n_int

    # we would need to left zero pad the binary to keep it correct
    bin_str = str(bin(out_hash_int))[2:]
    extra_out = len(bin_str)%n
    num_pad_out = (n-extra_out)*int(extra_out >0)
    padded_out =  num_pad_out*'0' + bin_str
    return padded_out
```

You can see this visualized on python tutor

We can test that we get something the right length

```
xor_hash('hello',8)
```

```
'00001101'
```

for different lengths of input

```
xor_hash('it is almost break',8)
```

```
'00110000'
```

This is a sort of meta function, so we could make simpler, more specific functions for each length or we can use it with different ones to see

```
xor_hash('it is almost break',13)
```

```
'0010111001001'
```

This is not cryptographic though because similar inputs have correlated outputs

Skip to main content

13.5. How do we get to the alphanumeric hashes we see?

The actual hashing algorithms are mathematical, they work by treating the binary input

Let’s build this up with some review

13.5.1. What is a Number ?

a mathematical object used to count, measure and label

13.5.2. 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

13.5.3. Hindu Arabic Number system

In modern, western cultures, our number system is called the hindu-arabic system, it consists of a set of **numerals**: 0,1,2,3,4,5,6,7,8,9 and uses a **place based** system with **base 10**.

13.5.3.1. Where does the name come from?

- invented by Hindu mathematicians in India 600 or earlier
- called “Arabic” numerals in the West because Arab merchants introduced them to Europeans
- slow adoption

13.5.3.2. Numerals

are the visual characters used to represent quantities

13.5.3.3. Place based

We use a **place based** system. That means that the position or place of the symbol changes its meaning. So 1, 10, and 100 are all different values. This system is also a decimal system, or base 10. So we refer to the places and the ones (10^0), the tens (10^1), the hundreds(10^2), etc for all powers of 10.

Number systems can use different characters, use different strategies for representing larger quantities, or both.

Skip to main content

13.5.4. Roman Numerals

is a number system that uses different numerals and is not a place based system

There are symbols for specific values: 1=I, V=5, X=10, L =50, C = 100, D=500, M = 1000.

Not all systems are place based, for example Roman numerals. In this system the subsequent symbols are either added or subtracted, with no (nonidentity) multipliers based on position. Instead if the symbol to right is the same or smaller, add the two together, if the left symbol is smaller, subtract it from the one on the right.

roman numerals	translation	value (hindu-arabic)
III	1+1+1	3
IV	-1 + 5	4
VI	5 +1	4
XLIX	-10 + 50 -1 + 10	49

13.6. Comparing Bases

13.6.1. Decimal

To represent larger numbers than we have digits on we have a base (10) and then.

$$10 = 10 * 1 + 1 * 0$$

$$22 = 10 * 2 + 1 * 2$$

we have the ones (10^0) place, tens (10^1) place, hundreds (10^2) place etc.

13.6.2. Binary

Binary is any base two system, and it can be represented using any different characters.

Binary number systems have origins in ancient cultures:

- Egypt (fractions) 1200 BC
- China 9th century BC
- India 2nd century BC

In computer science we use binary because mechanical computers began using relays (open/closed) to implement logical (boolean) operations and then digital computers use on and off in their circuits.

Skip to main content

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.

$$10 => 1 * 2^1 + 0 * 2^0 = 1 * 2 + 0 * 1 = 2$$

so 10 in binary is 2 in decimal

Convert 1001 in binary to decimal

$$1001 => 1 * 8 + 0 * 4 + 0 * 2 + 1 * 1 = 9$$

13.6.3. Octal

Is base 8.

We use hindu-arabic symbols, 0,1,2,3,4,5,6,7 (the first eight). Then nine is represented as 11.

In computer science, this numbering system was popular in 6 bit and 12 bit computers, but is has origins before that.

Native Americans using the Yuki Language (based in what is now California)used an octal system because they count using the spaces between fingers and speakers of the Pamean languages in Mexico count on knuckles in a closed fist. Europeans debated using decimal vs octal in the 1600-1800s for various reasons because 8 is better for math mostly. It is also found in Chinese texts dating to 1000BC.

Some examples: $$10 => 1 * 8^1 + 0 * 8^0 = 1 * 8 + 0 * 1 = 8$$ so 10 in octal is 8 in decimal

$$401 => 4 * 8^2 + 0 * 8^1 + 1 * 8^0 = 4 * 64 + 1 * 0 = 257$$

In computer science we use octal a lot because it reduces every 3 bits of a number in binary to a single character. So for a large number, in binary say `101110001100` we can change to `5614` which is easier to read, for a person.

13.6.4. Hexadecimal

base 16, common in CS because its 4 bits. we use 0,1,2,3,4,5,6,7,8,9,A,B,C,D,E,F.

This is commonly used for representing colors

Some examples:

$$A5 ==> (9 + 1) * 16^1 + 5 * 16^0 = 10 * 16 + 5 * 1 = 165$$

Skip to main content

$$E40 ==> 14 * 16^2 + 4 * 16^1 + 0 * 16^0 = 14 * 256 + 4 * 16 + 0 * 1 = 3648$$

It is also 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.

13.7. Prepare for Next Class

Be sure to get some rest during the break. Come to class ready to learn.

If you do work during the break, be sure to update your ide notes.

13.8. Badges

Review | **Practice**

1. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
2. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include links to your sources and be sure that the sources are trustworthy.
3. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts.md with a title `# Git counts`. Describe the scenario that would get you to that number of objects with the maximum *or* minimum number of commits in terms of what types of objects would exist. Assume normal git use with porcelain commands, not atypical cases with plubming commands. *If you get stuck, outline what you know and then request a review.*

13.9. gittransition

```
# transition questions

1. Why make the switch? (in detail, not just *an attack*)
2. What impact will the switch have on how git works?
3. Which developers will have the most work to do because of the switch?
```

13.10. Experience Report Evidence

13.11. Questions After Today's Class

13.11.1. Will we be approved for requests to join teams for builds from the discussion posts made?

13.11.2. want to learn more about how to use sha-256

This is a good community badge or explore option.

You can start with the git transition plan

13.11.3. What are all the parts of the SHA hashing algroithm?

You can see SHA-1 psuedocode on wikipedia

13.11.4. What makes git's algorithm "hardened"?

It uses the regular hashing algorithm, then checks if the resulting hash is vulnerable with a detection algorithm and if so, it computes a different hash. accodring to shattered.

13.12. Are there any benefits of a number system like Roman Numerals over the Hindu-Arabic system?

It is just different, it was something the Romans developed before Europeans had encountered a system like the Hindu-Arabic system.

13.12.1. Does git have a plan to change its hashing algorithm in the future?

Yes and a detailed transition plan

13.12.2. what is the most secure hash algorithm?

This is a good community badge as is, or by comparing several, can be an explore

13.12.3. Do you think it would be as easy for us to do math with a octal (base 8) system as it is with a decimal (base 10) system is if we learned it just as early?

Yes, if that is what you were used to, that would be it.

We would learn times tables in different ways possibly or different tricks because we represented the quantities differently, but yes!

Working out examples of this could be a good explore badge.

13.12.4. Will quantum computing be able to decript things such as hash encriptions ?

13.12.5. Are the git hashes unique universally, or just across the repository?

They only *have* to be unique withing a single repo, for example in the test repo, we all had some identical hashes.

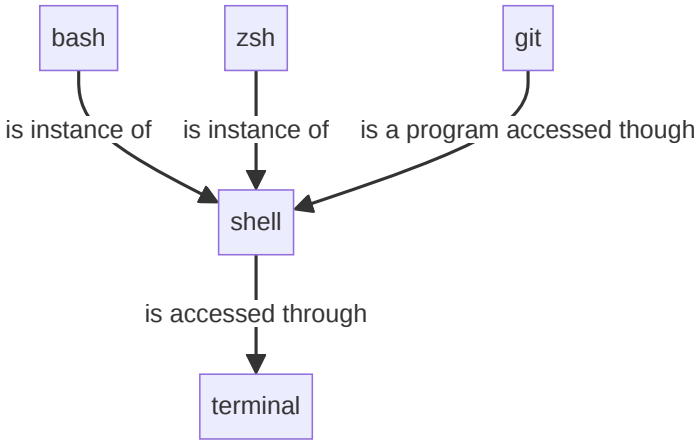
The blob and tree hashes are more likely to repeat across repos than commit hashes though because commit hashes cannot really repeat easily because it requires the same person to commit in two different repos (well the same logged in identity) at the same time.

14. How can I automate things with bash?

So far we have used bash commands to navigate our file system as a way to learn about the file system itself. To do this we used commands like:

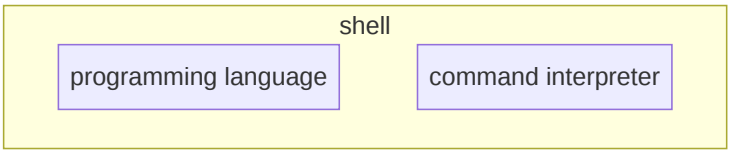
- `mv`
- `cd`
- `pwd`
- `ls`
- `find`
- `rm`

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



Skip to main content

14.1. Inspecting an example script

Today we will start by inspecting the github action file, first we'll navigate to the folder.

```
cd kwl-sp24-brownsarahm/
```

Next we'll look at which files we have

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

```
create_about.yml        experiencereport.yml    getassignment.yml
experienceinclass.yml   forgottenexperience.yml
```

Then we will look at the getassignmet one.

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

```
name: Create badge issues (Do not run manually)
on:
  workflow_dispatch

# once you edit, change the name

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

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        # prepare badge lines
        pretitle="prepare-"$(sysgetbadgedate --prepare)
        sysgetassignment --type prepare | gh issue create --title $pretitle --label prepare --body-file -
        # review badge lines
        rtitle="review-"$(sysgetbadgedate --review)
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        # practice badge lines
        pratitle="practice-"$(sysgetbadgedate --practice)
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file
      env:
        GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
#  edit the run step above for the level(s) you want.
#  You should keep the prepare, because they are required for experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

Skip to main content

14.2. Variables in Bash

We can create variables

```
MYVAR="my_val"
```

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 $MYVAR
```

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.

```
my_val
```

A common mistake is to put a space around the `=` sign, this is actually considered **good style** in many other languages.

```
NAME = "Sarah"
```

```
-bash: NAME: command not found
```

In bash, however, this creates an error. When there is a space after `NAME`, `bash` tried to interpet `NAME` as a bash command, but then it does not find it, so it gives an error.

Removing the space works again:

```
NAME="Sarah"
echo $NAME
```

```
Sarah
```

```
echo $NAME
```

```
Sarah
```

The `$` is essential syntax for recalling variables in bash. If we forget it, it treats it as a literal

```
echo NAME
```

Skip to main content

```
echo NAME
```

```
NAME
```

so we get the variable **name** out instead of the variable **value**

14.3. Environment Variables

Environment variables are global.

A common one is the `PATH`

```
echo $PATH
```

```
/Users/brownsarahm/anaconda3/bin:/Users/brownsarahm/anaconda3/condabin:/usr/local/bin:/usr/bin:/bin:/usr/
```

mine has my anaconda install, yours may be different

You can store environment variables to be set each time you start a terminal in your profile.

- On MacOS this file is: `~/.bash_profile`
- on linux it is `~/.baschrc`

```
cat ~/.bash_profile
```

```
alias pip=pip3
alias python=python3

export PS1="\u@\W $ "

# >>> conda initialize >>>
# !! Contents within this block are managed by 'conda init' !!
__conda_setup="$('/Users/brownsarahm/anaconda3/bin/conda' 'shell.bash' 'hook' 2> /dev/null)"
if [ $? -eq 0 ]; then
    eval "$__conda_setup"
else
    if [ -f "/Users/brownsarahm/anaconda3/etc/profile.d/conda.sh" ]; then
        . "/Users/brownsarahm/anaconda3/etc/profile.d/conda.sh"
    else
        export PATH="/Users/brownsarahm/anaconda3/bin:$PATH"
    fi
fi
unset __conda_setup
# <<< conda initialize <<<
```

A common one you might want to set is:

- `PS1` the primary prompt line. Content you can use is documented in the gnu docs
- `alias` lets you set another name for a program, for exmaple I use `pip` to call `pip3`

Skip to main content

14.4. Bash Loops

We can also make loops like

```
for loopvar in list
do
# loop body
echo $loopvar
done
```

So, for example:

```
for name in Sarah Cam Kai
> do
> echo $name
> done
```

Notes:

- The `>` is not typed, it is what happens since the interpreter knows that after we write the first line, the command is not complete.
- a list in bash is items with spaces, no brackets here (`Sarah Cam Kai`)

```
Sarah
Cam
Kai
```

and this does as expected

When we get the command back with the up arrow key, it puts it all on one line, because it was one command. The `;` (semicolon) separates the “lines”

```
for name in Sarah Cam Kai; do  echo $name; done
```

```
Sarah
Cam
Kai
```

14.5. Nesting commands

Let’s look at the action file again

```
name: Create badge issues (Do not run manually)
on:
  workflow_dispatch
```

Skip to main content

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

    # Install dependencies
    - name: Set up Python 3.9
      uses: actions/setup-python@v4
      with:
        python-version: 3.9

    - name: Install Utils
      run: |
        pip install git+https://github.com/introcompsys/courseutils@main
    - name: Get badge requirements
      run: |
        # prepare badge lines
        pretitle="prepare-"$(sysgetbadgedate --prepare)
        sysgetassignment --type prepare | gh issue create --title $pretitle --label prepare --body-file -
        # review badge lines
        rtitle="review-"$(sysgetbadgedate --review)
        sysgetassignment --type review | gh issue create --title $rtitle --label review --body-file -
        # practice badge lines
        pratitle="practice-"$(sysgetbadgedate --practice)
        sysgetassignment --type practice| gh issue create --title $pratitle --label practice --body-file
      env:
        GH_TOKEN: ${{ secrets.GITHUB_TOKEN }}
#  edit the run step above for the level(s) you want.
#  You should keep the prepare, because they are required for experience badges
#     You may choose to get only the review or only the practice (and change this any time)
```

We can run a command to generate the list by putting it inside `$()` to run that command first. Think kind of like PEMDAS and the `$` in bash is for variables.

This is useful in combination with the the loop

```
for file in $(ls)
> do
> echo $file
> done
```

```
README.md
community_contributions.md
experiences
```

the `$()` tells bash to run that command first and then hold its output as a variable for use elsewhere

If we modify it

```
for file in $(ls -a); do  echo $file ; done
```

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

Skip to main content

```
community_contributions.md
experiences
```

```
test hello
```

14.6. Conditionals in bash

We can also do conditional statements

```
if test -f .git
> 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 returns nothing because it is not a file

If we switch it, we get output:

```
if test -f README.md; then echo "file"; fi
```

```
file
```

We can put the if inside of the loop.

```
if test  .git; then echo "exists"; fi
```

```
exists
```

```
echo $(test hello)
```

```

```

```
for file in $(ls -a); do if test -f $file ;then  echo $file; fi ; done
```

```
README.md
```

```
community_contributions.md
```

14.7. Script files

We can put our script into a file

```
nano filecheck.sh
```

So that the file

```
cat filecheck.sh
```

```
for file in $(ls)
do
  if test -f $file
  then
     echo $file
  fi
done
```

and run it with `bash <filename>`

```
bash filecheck.sh
```

```
README.md
community_contributions.md
filecheck.sh
```

```
cd ../
```

```

```

```
ls
```

```
ex                          spring2024
gh-inclass-sp24-brownsarahm test
kwl-sp24-brownsarahm        tiny-book
```

```
cd spring2024/
```

```

```

Skip to main content

14.8. Prepare for Next Class

Review your `idethoughts.md` from a few weeks ago and add some summary notes.

14.9. Badges

Review | **Practice**

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that updates your badges.json file and then generates your progress report using courseutils *we will use this in lab on 3/25*

14.10. Experience Report Evidence

14.11. Questions After Today's Class

14.11.1. What parts of the terminal are saved local per window and what parts aren't?

local variables local, but there are also global variables

most programs should be per window.

14.11.2. Are there functions in bash scripting?

Yes see the docs

14.11.3. what is YMAL syntax similar to in terms of other programming langauges

It is kind of like a dicionary, but yaml is just a file type, like json or xml, not a a programming language. It is a way to store data that is parse-able.

14.11.4. What kind of action could be useful?

Anything you might do a alot and would prefer to have automated. Fore xample in calss we have the actions for the badges, the market place has examples.

there are also the ideas in the discussions.

Skip to main content

14.11.5. How to stop a loop (break statement)

`break` exists in bash

14.11.6. How to check values (greater than/less than)

those are operators defines on the conditional page

14.11.7. What differentiates a terminal and a shell?

a shell is the language/interpreter terminal is how you access it.

Sometimes they collapse like powershell is a terminal and shell.

14.11.8. How can I change the experience workflow to automatically fill in the date?

This is nontrival because you would need to edit the file instead of just copying it. It is a good build badge.

15. What is an IDE?

captioning

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?

Update your individual IDE notes with 1-2 things you learned from your peers.

15.1.2. Asynchronous

There are questions on the GitHub Discussion. Update your individual IDE notes in your KWL repo with links to your post and replies.

15.2. Learn more

> **Note**
>
> These are for both in person and async, but how you use them is different

- 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. Foe example, share your favorite extensions and configurations
- make a table of how would you evaluate attribute. Which ones would you evaluate by just if it exists or not? Which ones would you evaluate in different degrees, what attributes of them would you evaluate?
- Discuss 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 experience report (makeup) action. Use ISO date format: YYYY-MM-DD for the workflow input.

15.4. Prepare for Next Class

15.5. Badges

Review | **Practice**

1. Try a new IDE and make some notes about how it was to learn in newide.md What is easy? hard? What could you apply from the ones you already use? Were there features you had trouble finding?
2. Add at least one comment on others' posts on the GitHub discussions about IDEs.

15.6. Experience Report Evidence

15.7. 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.

Ti

You
on y

Minimum Rows

```
# KWL Chart


<!-- replace the  _ in the table or add new rows as needed -->
```

Skip to main content

```
| ------| ------- | ------ | ------- |
| Git | _ | _ | _ |
| GitHub | _ | _ | _ |
| Terminal | _ | _ | _ |
| IDE | _ | _ | _ |
| text editors | _ | _ | _ |
|file system | _ | _ |_ |
|bash | _ | _ | _ |
|abstraction | _ | _ | _ |
|programming languages | _ | _ | _ |
|git workflows | _ | _ | _ |
| git branches | _ | _ | _ |
| bash redirects | _ | _ | _ |
|number systems | _ | _ | _ |
| merge conflicts | _ | _ | _ |
| documentation | _ | _ | _ |
| templating | _ | _ | _ |
|bash scripting | _ | _ | _ |
| developer tools | _ | _ | _ |
| networking | _ | _ | _ |
|ssh | _ | _ | _ |
| ssh keys | _ | _ | _ |
|compiling | _ | _ | _ |
| linking   | _ | _ | _ |
| building | _ | _ | _ |
| machine representation  | _ | _ | _ |
| integers   | _ | _ | _ |
| floating point  | _ | _ | _ |
|logic gates | _ | _ | _ |
| ALU | _ | _ | _ |
| binary operations | _ | _ | _ |
| memory | _ | _ | _ |
| cache | _ | _ | _ |
| register | _ | _ | _ |
| clock | _ | _ | _ |
| Concurrency | _ | _ | _ |
```

Required Files

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

Skip to main content

date	file	type
2024-01-25	brain.md	practice
2024-01-30	gitgui.md	review
2024-01-30	gitgui.md	practice
2024-02-01	terminal-vocab.md	prepare
2024-02-01	branches.md	review
2024-02-01	branches-forks.md	practice
2024-02-06	terminal_review.md	review
2024-02-06	terminalpractice.md	practice
2024-02-06	terminal_organization_adv.md	practice
2024-02-06	software.md	prepare
2024-02-08	commit-def.md	review
2024-02-15	docs_ecosystems.md	practice
2024-02-20	newlanguage.md	practice
2024-02-22	software.md` about how that project adheres to and deviates from the unix philosophy. Be specific, using links to specific lines of code or specific sections in the documentation that support your claims. Provide at least one example of both adhering and deviating from the philosophy and three total examples (that is 2 examples for one side and one for the other). You can see what badge `software.md	practice
2024-02-22	methods.md	review
2024-02-22	methods.md	practice
2024-02-27	workflows.md	practice
2024-02-27	idethoughts.md	prepare
2024-02-27	gitdef.md	prepare
2024-02-27	favorite_git_workflow.md	review
2024-02-27	workflows.md	practice
	eview.md	review

Skip to main content

date	file	type
02-29		
2024-02-29	gitplumbingdetail.md	practice
2024-02-29	gitislike.md	practice
2024-03-05	tagtypes.md	practice
2024-03-05	tagtypeexplore.md	review
2024-03-07	gitcounts.md	review
2024-03-07	gitcounts_scenarios.md	practice
2024-03-07	hash_num_prep.md	prepare
2024-03-21	newide.md	practice
2024-03-21	newide.md	review
2024-03-21	settingssync.md	practice
2024-03-21	frequentide.md	practice

Team Repo

> **Warning**
>
> We will not use this in spring 2024

Contributions

Your team repo is a place to build up a glossary of key terms and a “cookbook” of “recipes” of common things you might want to do on the shell, bash commands, git commands and others.

For the glossary, follow the jupyterbook syntax.

For the cookbook, use standard markdown.

to denote code inline `use single backticks`

```
to denote code inline `use single backticks`
```

Skip to main content

```
```
to make a code block use 3 back ticks
```
```

To nest blocks use increasing numbers of back ticks.

To make a link, `[show the text in squarebrackets](url/in/parenthesis)`

Collaboration

You will be in a “team” that is your built in collaboration group to practice using Git Collaboratively.
There will be assignments that are to be completed in that repo as well. These activities will be marked accordingly. You will take turns and each of you is required to do the initialization step on a recurring basis.

This is also where you can ask questions and draft definitions to things.

Peer Review

If there are minor errors/typos, suggest corrections inline.

In your summary comments answer the following:

- Is the contribution clear and concise? Identify any aspect of the writing that tripped you up as a reader.
- Are the statements in the contribution verifiable (either testable or cited source)? If so, how do you know they are correct?
- Does the contribution offer complete information? That is, does it rely on specific outside knowledge or could another CS student not taking our class understand it?
- Identify one strength in the contribution, and identify one aspect that could be strengthened further.

Choose an action:

- If the suggestions necessary before merging, select **request changes**.
- If it is good enough to merge, mark it **approved** and open a new issue for the broader suggestions.
- If you are unsure, post as a **comment** and invite other group members to join the discussion.

Review Badges

Review After Class

After each class, you will need to review the day’s material. This includes reviewing prismia chat to see any questions you got wrong and reading the notes. Most days there will be specific additional activities and questions to answer. These should be in your KWL repo. Review activities will help you to reinforce what we do in class and guide you to practice with the most essential skills of this class.

Skip to main content

2024-01-23

related notes

Activities:

2024-01-25

related notes

Activities:

2024-01-30

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 comment. Be sure it is not inside another repository.
3. Try using setting up git using your favorite IDE's git integration (not its terminal) or GitHub Desktop. Make a file gitgui.md and include some notes of how it went. Give the file a heading like `# Setting up <tool name>`, with the actual tool name you setup in the title (eg Github Desktop or VSCode source control panel or …) Was it hard? easy? what did you figure out or get stuck on? Is the terminology consistent to the terminal or does it use different terms?

2024-02-01

related notes

Activities:

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

2024-02-06

related notes

Skip to main content

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on. In this case, in lab, we will check that you know what to do, but if we want you to do revisions those will be done through the badge.

1. Update your KWL chart with the new items and any learned items.
2. Clone the course website. Append the commands used and the contents of your `fall2023/.git/config` to a terminal_review.md (hint: history outputs recent commands and redirects can work with any command, not only echo). Edit the README.md, commit, and try to push the changes. What happens and what GitHub concept that we have not used yet might fix it? see your `vocab-` repo for a list of key github concepts. (answer in the `terminal_review.md`)
3. **lab** Organize the provided messy folder in a Codepsace (details will be provided in lab time). Commit and push the changes. Answer the questions below in your kwl repo in a file called `terminal_organization.md`
4. clone your `messy_repo` locally and append the `history.md` file to your `terminal_organization.md`

2024-02-08

related notes

Activities:

1. Export your git log for your KWL main branch to a file called gitlog.txt and commit that as exported to the branch for this issue. **note that you will need to work between two branchse to make this happen**. Append a blank line, `## Commands`, and another blank line to the file, then the command history used for this exercise to the end of the file.
2. In commit-def.md compare two of the four ways we described a commit today in class. How do the two descriptions differ? How does defining it in different ways help add up to improve your understanding?

2024-02-15

related notes

Activities:

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book. Set it so that the `_build` directory is not under version control. Complete basic customization setps for the necessary files and ensure that you do not add template files to your KWL repo.
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.

build idea build a sphinx extension that adds a particular feature to a documentation website. You can start a proposal and discuss ideas with Dr. Brown

2024-02-22

related notes

Skip to main content

1. Read today's notes when they are posted.
2. Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Review the open source project you looked at for the `software.md` file from before and decide if it more adheres to or deviates from the unix philosophy. Add a `## Unix Philosophy <Adherance/Deviation` section to your software.md, setting the title to indicate your decision and explain your decision in that section. Provide at least two specific examples, using links to specific lines of code or specific sections in the documentation that support your claims.
3. create methods.md and answer the following:

```
- Reflecting on your own programming practice, is initial understanding of concepts or debugging later mo
- which of the three methods for studying a system do you use most often in your chosen context? Give an
- do you think using a different strategy might help you in your chosen context sometimes? why or why not
```

2024-02-27

related notes

Activities:

1. Read about different workflows in git and describe which one you prefer to work with and why in favorite_git_workflow.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. In commit_contents.md, redirect the content of your most recent commit, its tree, and the contents of each tree and blob in that tree to the same file. Edit the file or use `echo` to put markdown headings between the different objects. Add a title `# Complete Commit` to the file and at the bottom of the file add `## Reflection` subheading with some notes on how, if at all this excercise helps you understand what a commit is.

2024-02-29

related notes

Activities:

1. Make a table in gitplumbingreview.md in your KWL repo that relates the two types of git commands we have seen: plubming and porcelain. The table should have two columns, one for each type of command (plubming and porcelain). Each row should have one git porcelain command and at least one of the corresponding git plumbing command(s). Include two rows: `add` and `commit`.

2024-03-05

related notes

Activities:

1. Use git bisect to find the first bad commit in the toy bug repo, save the command history and the bad commit hash to git_debug.md

Skip to main content

2. Create tagtypeexplore.md with the template below. Determine how many of the tags in the course website are annotated vs lightweight using. (You may need to use `git pull --tags` in your clone of the course website)

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
```

2024-03-07

related notes

Activities:

1. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
2. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include links to your sources and be sure that the sources are trustworthy.
3. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts.md with a title `# Git counts`. Describe the scenario that would get you to that number of objects with the maximum *or* minimum number of commits in terms of what types of objects would exist. Assume normal git use with porcelain commands, not atypical cases with plubming commands. *If you get stuck, outline what you know and then request a review.*

2024-03-19

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that updates your badges.json file and then generates your progress report using courseutils *we will use this in lab on 3/25*

2024-03-21

related notes

Activities:

Skip to main content

1. Try a new IDE and make some notes about how it was to learn in newide.md What is easy? hard? What could you apply from the ones you already use? Were there features you had trouble finding?
2. Add at least one comment on others' posts on the GitHub discussions about IDEs.

Prepare for the next class

These tasks are not always based on things that we have already done. Sometimes they are to have you start thinking about the topic that we are *about* to cover. Getting whatever you know about the topic fresh in your mind in advance of class will help what we do in class stick for you when we start.

The correct answer is not as important for these activities as it is to do them before class. We will build on these in class. These are evaluated on completion only, but we may ask you questions or leave comments if appropriate, in that event you should reply and then we'll approve.

2024-01-25

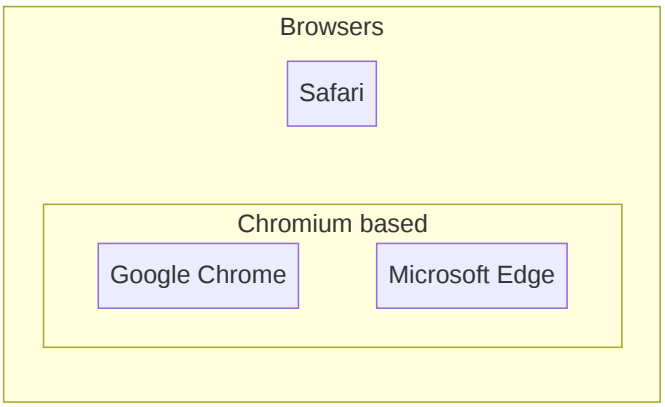
this will actually be done in the first few minutes of class

Think about one thing you've learned really well (computing or not). Be prepared to discuss the following: How do you know that you know it? What was it llike to first learn it?

2024-01-30

1. Find the glossary page for the course website, link it below. Review the terms for the next class: shell, terminal, bash, git, zsh, powershell, GitHub. Make a diagram using mermaid to highlight how these terms relate to one another
2. Check your kwl repo before class and see if you have recieved feedback, reply or merge accordingly.

Example "venn diagram " with mermaid subgraphs



2024-02-01

1. Add file called terminal-vocab.md on a branch linked to this issue. Fill in the template below:

```
# Terminal vocab

## Before

<insert your diagram from 2024-01-30 prepare>

## Reflection

<!-- write a few bullets assessing your diagram above based on the notes from 2024-01-30. was anything ir


## Final thoughts

<!-- write a few bullets/sentences on how trying to make the diagram did/not help you think about the ter
```

2024-02-06

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

```
## Software Reflection

Project : <markdown link to repo>

## README

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

## Contents

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


## Automation

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

## Documentation

<!-- what support for users? what for developers? code of conduct? citation? -->

## Hidden files and support
 <!-- What type of things are in the hidden files? who would need to see those files vs not? -->
```

Some open source projects if you do not have one in mind:

- pandas
- numpy
- GitHub CLI

Skip to main content

- vs code
- Typescript
- Swift
- Jupyter book
- git-novice lesson

2024-02-08

1. Bring git questions or scenarios you want to be able to solve to class on Thursday (in your mind or comment here if that helps you remember)
2. Practice using the new bash commands we have learned to read and understand the workflow files in your KWL repo. Look for bash commands that we have seen before. Try to modify files on a prepare branch so that your name is already filled in when your experience badge (inclass) action runs.

2024-02-13

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.

2024-02-20

related notes

Activities:

1. Think through and make some notes about what you have learned about design so far. Try to answer the questions below in design_before.md. If you do not now know how to answer any of the questions, write in what questions you have.

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

2024-02-27

Skip to main content

Activities:

1. review the notes on what is a commit. In gitdef.md on the branch for this issue, try to describe git in the four ways we described a commit. **the point here is to think about what you know for git and practice remembering it, not "get the right answer"; this is prepare work, we only check that it is complete, not correct**
2. Start recording notes on *how* you use IDEs for the next couple of weeks using the template file below. We will come back to these notes in class later, but it is best to record over a time period instead of trying to remember at that time. Store your notes in your kwl repo in idethoughts.md on an `ide_prep` branch. **This is prep for after spring break, it does not go in the Feb 27 experience branch; keep this branch open until it is specifically asked for**

2024-02-29

related notes

Activities:

Create a file `gitcommandsbreakdown.md` and for each command in the template below break down what steps it must do based on what we have learned so far about git objects. I started the first one as an example based on what we discussed today, fill in the others. In class on Thursday, we will make a commit using plumbing commands, so thinking about what you already know about commits will prepare you to learn this material.

```
# What git commands do

## `git status`

- check the branch of the HEAD pointer
- compare the HEAD pointer to the FETCH_HEAD, if different trace back through parent commits to find out
- compare the snapshot at the HEAD pointer's commit to the current working directory
- if staging is not empty, compare that to the working directory

## `git commit`

-

## `git add`

-
```

2024-03-05

related notes

Activities:

1. Read the notes from February 29th. We will build on these directly in the future. **You need to have the `test` repo with the same status for lab on class on 3/5** Make sure you have completed all of the steps in the github inclass repo.

2024-03-07

Skip to main content

Activities:

1. Take a few minutes to think what you know about hashing and numbers. Create hash_num_prep.md with two sections: `## Hashing` with a few bullet points summarzing key points about hashing, and `## numbers` where you draft a definition of a number
2. Use `git log` to view recent updates to the course website from the release for the Feb 29th class to the release for the March 5 class.

2024-03-19

related notes

Activities:

Be sure to get some rest during the break. Come to class ready to learn.

If you do work during the break, be sure to update your ide notes.

2024-03-21

related notes

Activities:

Review your `idethoughts.md` from a few weeks ago and add some summary notes.

2024-03-26

related notes

Activities:

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.

2024-01-23

related notes

Skip to main content

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

2024-01-25

related notes

Activities:

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

1. review today's notes after they are posted, both rendered and the raw markdown versions. Include links to both views in your badge PR comment. (to review + there are hints for some of the following there)
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. Do you think this information will help you approach learning more effectively? why or why not? How, if at all, does it changes how you think about debugging and learning to program? Give examples of how you have encountered the different types of confusion in your prior programming experiences. (to help you get good strategies and prime for things we will see in the next few weeks)
3. Make a concept map of your current understanding of git and github git-basics-map.md. Use mermaid syntax, to draw your map. GitHub can render it for you including while you work using the preview button. (review what we have learned so far; think about connections + learn a new tool)
4. Read more about version control in general and add a "version control" row to your KWL chart with all 3 columns filled in. (git is version control, but not the only one)

2024-01-30

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 comment. Be sure it is not inside another repository.
3. Try using setting up git using your favorite IDE's git integration (not its terminal) or GitHub Desktop. Make a file gitgui.md and include some notes of how it went. Give the file a heading like `# Setting up <tool name>` with the actual tool name you setup in the title (eg Github Desktop or VSCode source control panel or …). Was it hard? easy? what did you figure out or get stuck on? Is the terminology consistent to the terminal or does it use different terms?
4. Design a small demo to illustrate the difference between git add and git commit and how they impact `git push`. Write your demo in stage_commit.md. Compare what happens based on what you can see on GitHub and what you can see with git status. Denote what you can tell about each case from the terminal and what you can tell from GitHub.com. For this demo, test things either using the files for this badge in your KWL repo or create a new sandbox repo where your account (not the course) is the owner.

Skip to main content

2024-02-01

related notes

Activities:

1. Create a merge conflict in your KWL repo on the branch for this issue and resolve it using your favorite IDE, then create one and resolve it on GitHub in browser (this requires the merge conflict to occur on a PR). Describe how you created it, show the files, and describe how your IDE helps or does not help in merge_conflict_comparison.md. Give advice for when you think someone should resolve a merge conflict in GitHub vs using an IDE. (if you do not regulary use an, IDE, try VSCode) *You can put content in the file for this step for the purpose of making the merge conflicts for this exercise.*
2. Learn about GitHub forks and more about git branches(you can also use other resources)
3. In branches-forks.md in your KWL repo, compare and contrast branches and forks; be specific about their relationship. You may use mermaid diagrams if that helps you think through or communicate the ideas. If you use other resources, include them in your file as markdown links.

2024-02-06

related notes

Activities:

badge steps marked **lab** are steps that you will be encouraged to use lab time to work on. For this one in particular, I am going to give you the messy repo in lab.

1. Update your KWL chart with any learned items.
2. Get set up so that you can pull from the course website repo and push to your own fork of the class website by cloning the main repo, then forking it and adding your fork as an additional remote. Append the commands used and the contents of your `spring2024/.git/config` to a terminalpractice.md (hint: `history` outputs recent commands and redirects can work with any command, not only echo). Based on what you know so far about forks and branches, what advantage does this setup provide? (answer in the `terminal_practice.md`)
3. **lab** Organize the provided messy folder (details will be provided in lab time). Commit and push the changes. Clone that repo locally.
4. Organize a folder on your computer (good candidate may be desktop or downloads folder), using only a terminal to make new directories, move files, check what's inside them, etc. Answer reflection questions in a new file, terminal_organization_adv.md in your kwl repo. Tip: Start with a file explorer open, but then try to close it, and use only command line tools to explore and make your choices. If you get stuck, look up additional commands to do acomplish your goals.

```
# Terminal File moving reflection
1. How was this activity overall Did this get easier toward the end?
2. How was it different working on your own computer compared to the Codespace form?
3. Did you have to look up how to do anything we had not done in class?
4. When do you think that using the terminal will be better than using your GUI file explorer?
5. What questions/challenges/ reflections do you have after this?
6. Append all of the commands you used in lab below. (not from your local computer's history, from the co
```

Skip to main content

2024-02-08

related notes

Activities:

1. Explore the tools for conventional commits and then pick one to try out. Use one of the tools that helps making conventional commits (eg in VSCode or a CLI for it)for a series of commits adding “features” and “bug fixes” telling the story of a code project in a file called commit-story.md. For each edit, add short phrases like ‘new feature 1’, or ‘next bug fix’ to the single file each time, but use conventional commits for each commit. In total make at least 5 different types of changes (types per conventional commits standard) including 2 breaking changes and at least 10 total commits to the file.
2. learn about options for how git can display commit history. Try out a few different options. Choose two, write them both to a file, gitlog-compare.md. Using a text editor, wrap each log with three backticks to make them “code blocks” and then add text to the file describing a use case where that format in particular would be helpful. **do this after the above so that your git log examples include your conventional commits**

2024-02-15

related notes

Activities:

1. Review the notes, jupyterbook docs, and experiment with the `jupyter-book` CLI to determine what files are required to make `jupyter-book build` run. Make your kwl repo into a jupyter book. Set it so that the `_build` directory is not under version control. Complete basic customization setps for the necessary files and ensure that you do not add template files to your KWL repo.
2. Learn about the documentation ecosystem in another language that you know using at least one official source and additional sources as you find helpful. In docs_ecosystems.md include a summary of your findings and compare and contrast it to jupyter book/sphinx. Include a bibtex based bibliography of the sources you used. You can use this generator for informal sources and google scholar for formal sources.

explore idea extend the conversion of your repo into a jupyter book by making links among pages, adding an intro, and adding a github action so that a pdf is generated and added to an orphan branch named `gh-pages`. Plus use at least 2 other jupyter book features from the docs, specify or discuss, your extended features with Dr. Brown in your proposal.

2024-02-20

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.

Skip to main content

2. Learn about one of the following languages that you have not used before: R, Julia, Clojure, Zig, Go, erlang, Elixir or another language you are curious about that appeared in the Developer survey. For example you might be interest in the top paying languages. Use the official documentation. Answer the following questions in languagelearning.md:

```
1. What is this language designed for?
2. What Programming paradigm(s) does it support?
3. Give at least 1 example of a programming language that experience in would make it **easy** to lea
4. Give at least 1 example of a programming language that experience in would make it **hard** to lea
5. What is its most unique feature(s)?
```

hint: for questions 3 & 4, for example learning Python first does not make it easy to learn C++, but learning C makes it very easy to learn C++.

Explore badge option

Actually try out one of the langauges above in a new repo created from this asssignment.

work on a branch and commit after each step

1. Setup the devcontainer to provide the compiler/interpreter for the language of your choice. See features
2. Add top extensions related to the language to the devcontainer.
3. Add a gitignore file for that language
4. Write a hello world type program in the language and make sure it runs.
5. Use the official documentation to figure out a common task in the language and write an example program that completes something typical for that language (eg a sort data analysis in R). You may use GitHub Co-pilot in the code space to help write the code.
6. Test your understanding of the code by trying to modify it.
7. Fill in the prompts in the README (follow the comments)
8. Answer the questions in the README of the template repo.

2024-02-22

related notes

Activities:

1. Most real projects partly adhere and at least partly deviate from any major design philosophy or paradigm. Add a `## Unix Philosophy` section to your software.md about how that project adheres to and deviates from the unix philosophy. Be specific, using links to specific lines of code or specific sections in the documentation that support your claims. Provide at least one example of both adhering and deviating from the philosophy and three total examples (that is 2 examples for one side and one for the other). You can see what badge `software.md` was previously assigned in and the original instructions on the KWL file list.
2. create methods.md and answer the following:

```
- which of the three methods for studying a system do you use most often when debugging? Give an example
                                                                                          ıen you are debugg
```

```
- do you think the ones you use most often are consistently effective? why or why not? When do they work
- what are you most interested in trying that might be different for either learning new things or debugg
```

2024-02-27

related notes

Activities:

1. Read about different workflows in git and add responses to the below in a workflows.md in your kwl repo. Two good places to read from are Git Book and the atlassian Docs
2. Update your kwl chart with what you have learned or new questions in the want to know column
3. Find the hash of the blob object that contains the content of your workflows.md file and put that in the comment of your badge PR for this badge.

```
## Workflow Reflection

1. Why is it important that git can be used with different workflows?
1. Which workflow do you think you would like to work with best and why?
1. Describe a scenario that might make it better for the whole team to use a workflow other than the one
```

2024-02-29

related notes

Activities:

1. Read more details about git internals to review what we did in class in greater detail. Make a file gitplumbingdetail.md and create a 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 (generally more than one each).
2. Create gitislike.md and explain main git operations we have seen (add, commit, push) in your own words in a way that will either help you remember or how you would explain it to someone else at a high level. This might be analogies or explanations using other programming concepts or concepts from a hobby.

2024-03-05

related notes

Activities:

1. Practice using git to fix problem using a git command from the patching or debugging section of the docs. Create a log of what you did using the history or git log into a file git_debug_story.md. If you have a project in another class or another badge in this class that causes you to use one of these features in a real scenario, that can count. If not, for example, you could create a “bug” and then use bisect to find it.

Skip to main content

2. Create tagtypes.md with the template below. Include an experiment that shows which if either type of tag creates a new git object. There are two types, try creating one of each a lightwight tag (provide only the tag name- what we did in class) and an annotated (provide a name and a message with `-m`).
3. Determine how many of the tags in the course website are annotated vs lightweight using. (You may need to use `git pull --tags` in your clone of the course website)

```
# Tags

<!-- short defintion/description in your own words of what a tag is and what it is for -->


## Comparing tag types

<!-- include your experiment terminal history  and interpretation -->

## Inspecting tags

Course website tags by type:
- annoted:
- lightweight:
<!-- include lists of tags for each type -->
```

2024-03-07

related notes

Activities:

3. Analyze the xor hashing algorithm in the notes to determine which properties of a cryptographic hash are/not met. Include your analysis in xorhash.md
4. find 2 more real world examples of using other number systems (either different bases or different symbols and bases) **not mentioned in class** that are currently used. Describe the number system and its usage in numbers.md. Include links to your sources and be sure that the sources are trustworthy.
5. Calculate the maximum number of git objects that a repo can have without requiring you to use more than the minimum number of characters to refer to any object and include that number in gitcounts_scenarios.md with a title `# Git counts`. Describe 3 scenarios that would get you to that number of objects in terms of what types of objects would exist. For example, what is the maximum number of commits you could have without exceeding that number, how many files could you have? How could you get to that number of objects in the fewest number of commits? What might be a typical way to get there? Assume normal git use with porcelain commands, not atypical cases with plubming commands. *If you get stuck, outline what you know and then request a review.*
6. Read about the Learn more about the SHA-1 collision attach
7. Learn more about how git is working on changing from SHA-1 to SHA-256 and answer the transition questions below gittransition.md

gittransition

```
# transition questions
```

```
2. What impact will the switch have on how git works?
3. Which developers will have the most work to do because of the switch?
```

2024-03-19

related notes

Activities:

1. Update your KWL Chart learned column with what you've learned
2. Write a bash script that updates your badges.json file and then generates your progress report using courseutils *we will use this in lab on 3/25*
3. Update your action file for getting assignments to use the new courseutils (at compsys-progtools instead of introcompsys)

2024-03-21

related notes

Activities:

The first three tasks are the practice task, Option A and Option B are pre-approved explore badges. Hover over one of them on your issue to create an independent issue to track it. Title the new issue `Explore: IDE-X` where `X` is `A` or `B` based on your choice. On the explore badge, when it is ready for submission, request a review from Dr. Brown.

1. Examine 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. Try a new IDE and make some notes about how it was to learn in newide.md What is easy? hard? What could you apply from the ones you already use? Were there features you had trouble finding?
3. Configure your VS Code preferences to your github account. add settingssync.md with a description of what settings you customized and synced and reflect on why this is an important feature and what prerequisites to it might be.
4. (explore option A) 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. Describe the procedure you did to compare the IDEs. 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.
5. (explore option B) Create a small repo owned by compsys named `ide-USERNAME` where `USERNAME` is your gh username with some example code, a vscode/codespace devcontainer file that installs CodeTour and your favorite extension(s). Write a CodeTour that walks someone through using your favorite extension to do something with the code. The example code can be any language, can be very simple, can even have a bug in it if that helps your example. You can use an-IDE integrated LLM (eg GitHub Co-pilot, not the chat version) to generate some code for this purpose if you do not have some available to you already, but you cannot share solutions to a course assignment without that instructor's permission.

KWL File List

Explore Badges

> **Warning**
>
> Explore Badges are not required, but an option for higher grades. The logistics of this could be streamlined or the instructions may become more detialed during the penalty free zone.

Explore Badges can take different forms so the sections below outline some options. This page is not a cumulative list of requirements or an exhaustive list of options.

> **Tip**
>
> You might get a lot of suggestions for improvement on your first one, but if you apply that advice to future ones, they will get approved faster.

How do I propose?

Create an issue on your kwl repo, label it explore, and “assign” @brownsarahm.

In your issue, describe the question you want to answer or topic to explore and the format you want to use.

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

Skip to main content

Explore Badge Ideas:

- Extend a more practice:
  - for a more practice that asks you to describe potential uses for a tool, try it out, find or write code excerpts and examine them
  - for a more practice that asks you to try something, try some other options and compare and contrast them. eg “try git in your favorite IDE” -> “try git in three different IDEs, compare and contrast, and make recommendations for novice developers”
- For a topic that left you still a little confused or their was one part that you wanted to know more about. Details your journey from confusion or shallow understanding to a full understanding. This file would include the sources that you used to gather a deeper understanding. eg:
  - Describe how cryptography evolved and what caused it to evolve (i.e. SHA-1 being decrypted)
  - Learn a lot more about a specific number system
  - compare another git host
  - try a different type of version control
- Create a visual aid/memory aid to help remember a topic. Draw inspriation from Wizard Zines
- Review a reference or resource for a topic
- write a code tour that orients a new contributor to a past project or an open source tool you like.

Examples from past students:

- Scripts/story boards for tiktoks that break down course topics
- Visual aid drawings to help remember key facts

For special formatting, use jupyter book’s documentation.

Build Badges

Build may be individual or in pairs.

Proposal Template

If you have selected to do a project, please use the following template to propose a bulid

```
## < Project Tite >

<!-- insert a 1 sentence summary  -->

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build
```

Skip to main content

```
<!-- list what your project will produce with target deadlines for each-->

### Milestones
```

The deliverables will depend on what your method is, which depend on your goals. It must be approved and the final submitted will have to meet what is approved. Some guidance:

- any code or text should be managed with git (can be GitHub or elsewhere)
- if you write any code it should have documentation
- if you do experiments the results should be summrized
- if you are researching something, a report should be 2-4 pages, plus unlimited references in the 2 column ACM format.

This guidance is generative, not limiting, it is to give ideas, but not restrict what you *can* do.

Updates and work in Progress

These can be whatever form is appropriate to your specific project. Your proposal should indicate what form those will take.

Summary Report

This summary report will be added to your kwl repo as a new file `build_report_title.md` where `title` is the (title or a shortened version) from the proposal.

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.

Collaborative Build rules/procedures

- Each student must submit a proposal PR for tracking purposes. The proposal may be shared text for most sections but the deliverables should indicate what each student will do (or be unique in each proposal).
- the proposal must indicate that it is a pair project, if iteration is required, I will put those comments on both repos but the students should discuss and reply/edit in collaboration
- the project must include code reviews as a part of the workflow links to the PRs on the project repo where the code reviews were completed should be included in the reflection
- each student must complete their own reflection. The abstract can be written together and shared, but the reflection must be unique.

Build Ideas

General ideas to write a proposal for

- make a vs code extension for this class or another URI CS course
- port the courseutils to rust. crate clap is like the python click package I used to develop the course utils
- buld a polished documentation website for your CSC212 project with sphinx or another static site generator
- use version control, including releases on any open source side-project and add good contributor guidelines, README, etc

Auto-approved proposals

For these build options, you can copy-paste the template below to create your proposal issue and assign it to `@brownsarahm`.

For working alone there are two options, for working with a partner there is one.

212 Project Solo- Docs focus

Use this option if your team for your 212 project is not currently enrolled in this class or does not want to do a collaborative build. This version focuses on the user docs.

```
## 212 Project Doc & Developer onboarding

Add documentation website and developer onboarding information to your CSC 212 project.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will

This project will provide information for a user to use the data structure implemented for a CSC 212 proj

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build
1. ensure there is API level documentation in the code files
1. build a documentation website using [jupyterbook/ sphinx/doxygen/] that includes setup instructions ar
1. configure the repo to automatically build the documentation website each time the main branch is updat


### Deliverables


- link to repo with the contents listed in method in the reflection file


### Milestones

<!-- give a target timeline -->
```

Use this option if your team for your 212 project is not currently enrolled in this class or does not want to do a collaborative build. This version focuses on the contributor experience.

```
## 212 Project Doc & Developer onboarding

Add documentation website and developer onboarding information to your CSC 212 project.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will

This project will provide information for a user to use the data structure implemented for a CSC 212 proj

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build

1. ensure there is API level documentation in the code files
1. add a license, readme, and contributor file
1. add [code tours](https://marketplace.visualstudio.com/items?itemName=vsls-contrib.codetour) that help
1. set up a PR template
1. set up 2 issue templates: 1 for feature request and 1 for bug reporting


### Deliverables


- link to repo with the contents listed in method in the reflection file


### Milestones

<!-- give a target timeline -->
```

212 Project Pair

Use this option if your teammate for your 212 project is in this class and wants to do a collaborative build.

```
## 212 Project Doc & Developer onboarding

Add documentation website and developer onboarding information to your CSC 212 project.

### Objectives

<!-- in this section describe the overall goals in terms of what you will learn and the problem you will

This project will provide information for a user to use the data structure implemented for a CSC 212 proj

### Method

 <!-- describe what you will do , will it be research, write & present? will there be something you build
1. ensure there is API level documentation in the code files
1. build a documentation website using [jupyterbook/ sphinx/doxygen/] that includes setup instructions ar
1. configure the repo to automatically build the documentation website each time the main branch is updat
1. add [code tours](https://marketplace.visualstudio.com/items?itemName=vsls-contrib.codetour) that help
1. set up a PR template
1. set up 2 issue templates: 1 for feature request and 1 for bug reporting



### Deliverables
```

```
- link to repo with the contents listed in method in the reflection file

### Milestones

<!-- give a target timeline -->
```

Syllabus and Grading FAQ

How much does activity x weigh in my grade?

How do I keep track of my earned badges?

Also, when are each badge due, time wise?

Who should I request to review my work?

Will everything done in the penalty free zone be approved even if there are mistakes?

Once we make revisions on a pull request, how do we notify you that we have done them?

What should work for an explore badge look like and where do I put it?

Git and GitHub

I can’t push to my repository, I get an error that updates were rejected

My command line says I cannot use a password

Help! I accidentally merged the Badge Pull Request before my assignment was graded

Skip to main content

Skip to main content

Skip to main content

Skip to main content

Skip to main content

For an Assignment, should we make a new branch for every assignment or do everything in one branch?

```
Doing each new assignment in its own branch is best practice. In a typical software development flow once
```

Other Course Software/tools

Courseutils

This is how your badge issues are created. It also has some other utilities for the course. It is open source and questions/issues should be posted to its issue tracker

Jupyterbook

Glossary

> **Tip**
>
> Contributing glossary terms or linking to uses of glossary terms to this page is eligible for community badges

absolute path

the path defined from the root of the system

add (new files in a repository)

the step that stages/prepares files to be committed to a repository from a local branch

argument

input to a command line program

bash

bash or the bourne-again shell is the primary interface in UNIX based systems

bitwise operator

an operation that happens on a bit string (sequence of 1s and 0s). They are typically faster than operations on whole integers.

branch

a copy of the main branch (typically) where developmental changes occur. The changes do not affect other branches

Compiled Code

code that is put through a compiler to turn it into lower level assemlby language before it is executed. must be compiled and re-executed everytime you make a change.

detached head

a state of a git repo where the head pointer is set to a commit without a branch also pointing to the commit

directory

a collection of files typically created for organizational purposes

divergent

git branches that have diverged means that there are different commits that have same parent; there are multipe ways that git could fix this, so you have to tell it what strategy to use

fixed point number

the concept that the decimal point does not move in the number. Cannot represent as wide of a range of values as a floating point number.

floating point number

the concept that the decimal can move within the number (ex. scientific notation; you move the decimal based on the exponent on the 10). can represent more numbers than a fixed point number.

git

a version control tool; it's a fully open source and always free tool, that can be hosted by anyone or used without a host, locally only.

GitHub

a hosting service for git repositories

.gitignore

a file in a git repo that will not add the files that are included in this .gitignore file. Used to prevent files from being unnecessarily committed.

git objects

FIXME something (a file, directory) that is used in git; has a hash associated with it

git Plumbing commands

low level git commands that allow the user to access the inner workings of git.

git Workflow

a recipe or recommendation for how to use Git to accomplish work in a consistent and productive manner

HEAD

a file in the .git directory that indicates what is currently checked out (think of the current branch)

merge

merge conflict

when two branches to be merged edit the same lines and git cannot automatically merge the changes

mermaid

mermaid syntax allows user to create precise, detailed diagrams in markdown files.

hash function

the actual function that does the hashing of the input (a key, an object, etc.)

hashing

transforming an input of arbitrary length to a unique fixed length output (the output is called a hash; used in hash tables and when git hashes commits).

integrated development environment

also known as an IDE, puts together all of the tools a developer would need to produce code (source code editor, debugger, ability to run code) into one application so that everything can be done in one place. can also have extra features such as showing your file tree and connecting to git and/or github.

interpreted code

code that is directly executed from a high level language. more expensive computationally because it cannot be optimized and therefore can be slower.

issue

provides the ability to easily track ideas, feedback, tasks, or bugs. branches can be created for specific issues. an issue is open when it is created. pull requests have the ability to close issues. see more in the docs

Linker

a program that links together the object files and libraries to output an executable file.

option

also known as a flag, a parameter to a command line program that change its behavior, different from an argument

path

the “location” of a file or folder(directory) in a computer

pointer

a variable that stores the address of another variable

pull (changes from a repository)

download changes from a remote repository and update the local repository with these changes.

pull request

allow other users to review and request changes on branches. after a pull request recieves approval you can merge the changed content to the main branch.

PR

Skip to main content

push (changes to a repository)

to put whatever you were working on from your local machine onto a remote copy of the repository in a version control system.

relative path

the path defined **relative** to another file or the current working directory; may start with a name, includes a single file name or may start with `./`

release

a distribution of your code, related to a git tag

remote

a copy of the repository hosted on a server

repository

a project folder with tracking information in it in the form of a .git directory in it

ROM (Read-Only Memory)

Memory that only gets read by the CPU and is used for instructions

SHA 1

the hashing function that git uses to hash its functions (found to have very serious collisions (two different inputs have same hashes), so a lot of software is switching to SHA 256)

sh

abbr. see shell

shell

a command line interface; allows for access to an operating system

ssh

allows computers to safely connect to networks (such as when we used an ssh key to clone our github repos)

templating

templating is the idea of changing the input or output of a system. For instance, the Jupyter book, instead of outputting the markdown files as markdown files, displays them as HTML pages (with the contents of the markdown file).

terminal

a program that makes shell visible for us and allows for interactions with it

tree objects

type of git object in git that helps store multiple files with their hashes (similar to directories in a file system)

yml

see YAML

Skip to main content

a file specification that stores key-value pairs. It is commonly used for configurations and settings.

zsh

zsh or z shell is built on top of the bash shell and contains new features

General Tips and Resources

This section is for materials that are not specific to this course, but are likely useful. They are not generally required readings or installs, but are options or advice I provide frequently.

on email

- how to e-mail professors

How to Study in this class

In this page, I break down how I expect learning to work for this class.

Begin a great programmer does not require memorizing all of the specific commands, but instead knowing the common patterns and how to use them to interpret others' code and write your own. Being efficient requires knowing how to use tools and how to let the computer do tedious tasks for you. This is how this course is designed to help you, but you have to get practice with these things.

Using reference materials frequently is a built in part of programming, most languages have built in help as a part of the language for this reason. These tools can help you when you are writing cod eand forget a specific bit of syntax, but these tools will not help you *read* code or debug environment issues. You also have to know how to effectively use these tools. Knowing the common abstractions we use in computing and recognizing them when they look a little bit differently will help you with these more complex tasks. Understanding what is common when you move from one environment to another or to This course is designed to have you not only learn the material, but also to build skill in learning to program. Following these guidelines will help you build habits to not only be successful in this class, but also in future programming.

Why this way?

Learning requires iterative practice. In this class, you will first get ready to learn by preparing for class. Then, in class, you will get a first experience with the material. The goal is that each class is a chance to learn by engaging with the ideas, it is to be a guided inquiry. Some classes will have a bit more lecture and others will be all hands on with explanation, but the goal is that you *experience* the topics in a way that helps you remember, because being immersed in an activity helps brains remember more than passively watching something. Then you have to practice with the material

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.

A new bo
programn
Program
available
that linke

Skip to main content

Learning in class

> **Important**
>
> My goal is to use class time so that you can be successful with *minimal frustration* while working outside of class time.

Programming requires both practical skills and abstract concepts. During class time, we will cover the practical aspects and introduce the basic concepts. You will get to see the basic practical details and real examples of debugging during class sessions. Learning to debug something you've never encountered before and setting up your programming environment, for example, are *high frustration* activities, when you're learning, because you don't know what you don't know. On the other hand, diving deeper into options and more complex applications of what you have already seen in class, while challenging, is something I'm confident that you can all be successful at with minimal frustration once you've seen basic ideas in class. My goal is that you can repeat the patterns and processes we use in class outside of class to complete assignments, while acknowledging that you will definitely have to look things up and read documentation outside of class.

Each class will open with some time to review what was covered in the last session before adding new material.

To get the most out of class sessions, you should have a laptop with you. During class you should be following along with Dr. Brown. You'll answer questions on Prismia chat, and when appropriate you should try running necessary code to answer those questions. If you encounter errors, share them via Prismia chat so that we can see and help you.

After class

After class, you should practice with the concepts introduced.
This means reviewing the notes: both yours from class and the annotated notes posted to the course website.
When you review the notes, you should be adding comments on tricky aspects of the code and narrative text between code blocks in markdown cells. While you review your notes and the annotated course notes, you should also read the documentation for new modules, libraries, or functions introduced in that class.

If you find anything hard to understand or unclear, write it down to bring to class the next day or post an issue on the course website.

GitHub Interface reference

This is an overview of the parts of GitHubt from the view on a repository page. It has links to the relevant GitHub documentation for more detail.

Top of page

The very top menu with the logo in it has GitHub level menus that are not related to the current repository.

Repository specific page

Code | Issues | Pull Requests | Actions | Projects | Security | Insights | Settings

This is the main view of the project

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

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

File panel

the header in this area lists who made the last commit, the message of that commit, the short hash, date of that commit and the total number of commits to the project.

If there are actions on the repo, there will be a red x or a green check to indicate that if it failed or succeeded on that commit.

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

Releases mark certain commits as important and give easy access to that version. They are related to git tags

Packages are out of scope for this course. GitHub helps you manage distributing your code to make it easier for users.

Environments are a tool for dependency management. We will cover thigns that help you know how to use this feature indirectly, but probably will not use it directly in class. This would be eligible for a build badge.

the header in this area lists who made the last commit, the message of that commit, the short hash, date of that commit and the total number of commits to the project.

If there are actions on the repo, there will be a red x or a green check to indicate that if it failed or succeeded on that commit. ^^^ file list: a table where the first column is the name, the second column is the message of the last commit to change that file (or folder) and the third column is when is how long ago/when that commit was made

README file

The bottom of the right panel has information about the languages in the project

Skip to main content

Language/Shell Specific References

- bash
- C
- Python

Bash commands

command	explanation
`pwd`	print working directory
`cd <path>`	change directory to path
`mkdir <name>`	make a directory called name
`ls`	list, show the files
`touch`	create an empty file
`echo 'message'`	repeat ‘message’ to stdout
`>`	write redirect
`>>`	append redirect
`rm file`	remove (delete) `file`
`cat`	concatenate a file to standard out (show the file contents)

Skip to main content

git commands

command	explanation
`status`	describe what relationship between the working directory and git
`clone <url>`	make a new folder locally and download the repo into it from url, set up a remote to url
`add  <file>`	add file to staging area
`commit -m 'message'`	commit using the message in quotes
`push`	send to the remote
`git log`	show list of commit history
`git branch`	list branches in the repo
`git branch new_name`	create a `new_name` branch
`git checkout -b new_Name`	create a `new_name` branch and switch to it
`git pull`	apply or fetch and apply changes from a remote branch to a local branch
`git commit -a -m 'msg'`	the `-a` option adds modified files (but not untracked)

Getting Help with Programming

This class will help you get better at reading errors and understanding what they might be trying to tell you. In addition here are some more general resources.

Skip to main content

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.

Skip to main content

File structure

I recommend the following organization structure for the course:

```
CSC3392
  |- kwl-
  |- gh-inclass
  |- semYYYY
  |- ...
```

This is one top level folder will all materials in it. A folder inside that for in class notes, and one folder per repository.

Please **do not** include all of your notes or your other assignments all inside your portflio, it will make it harder to grade.

Finding repositories on github

Each assignment repository will be created on GitHub with the `compsys-progtools` organization as the owner, not your personal acount. Since your account is not the owner, they do not show on your profile.

If you go to the main page of the organization you can search by your username (or the first few characters of it) and see only your repositories.

More info on cpus

Resource	Level	Type	Summary
What is a CPU, and What Does It Do?	1	Article	Easy to read article that explains CPUs and their use. Also touches on “buses” and GPUs.
Processors Explained for Beginners	1	Video	Video that explains what CPUs are and how they work and are assembled.
The Central Processing Unit	1	Video	Video by Crash Course that explains what the Central Processing Unit (CPU) is and how it works.

Windows Help & Notes

CRLF Warning

This is GitBash telling you that git is helping. Windows uses two characters for a new line `CR` (cariage return) and `LF` (line feed). Classic Mac Operating system used the `CR` character. Unix-like systems (including MacOS X) use only the `LF` character. If you try to open a file on Windows that has only `LF` characters, Windows will think it's all one line. To help you, since git knows people collaborate across file systems, when you check out files from the git database (`.git/` directory) git replaces `LF` characters with `CRLF` before updating your working directory.

When working on Windows, when you make a file locally, each new line will have `CRLF` in it. If your collaborator (or server, eg GitHub) runs not a unix or linux based operating system (it almost certainly does) these extra characters will make a mess and make the system interpret your code wrong. To help you out, git will automatically, for Windows users, convert `CRLF` to `LF` when it adds your work to the index (staging area). Then when you push, it's the compatible version.

git documentation of the feature