About this Site

Contents

Welcome to the course manual for Introduction to Computer Systems in Spring 2022 with Professor Brown.

This class meets TuTH 12:30-1:45 in Engineering Building Room 040.

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

Course Calendar on BrightSpace

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

Navigating the Sections

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

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

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

Reading each page

All class notes can be downloaded in multiple formats, including as a notebook. Some pages of the syllabus and resources are also notebooks, if you want to see behind the curtain of how I manage the course information.

> **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.

> **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. Short questions will be in the margin note

> **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.

> **Think Ahead**
>
> Think ahead boxes will guide you to start thinking about what can go into your portfolio to build on the material at hand.

Syllabus

Welcome to CSC302: Introduction to Computer Systems.

In this syllabus you will find an overview of the course, information about your instructor, course policies, restatements of URI policies, reminders of relevant resources, and a schedule for the course.

This is a live document that will change over time, but a pdf copy is available for direct download or to view on GitHub. Note that this will become outdated over time.

Basic Facts

Introduction to Computer Systems

This new course links together different ideas that you have encountered but not covered deeply in other courses. We'll learn about tools used in programming and how they work. The goal of this course is to help you understand how your computer and programming environment work so that you can debug and learn independently more confident.

Quick Facts

- **Course time:** Spring 2022, TuTh 12:30PM - 1:45PM
- **Credits:** 4

To request a permission number complete this google form **you must be signed into your URI google account to access the form**

Why Take this course

1. use and understand git/ GitHub
2. make sense of cryptic compiler messages
3. understand how file organization impacts programming
4. fulfill your 300 level CSC elective requirement
5. preview ideas that will be explored in depth in 411 & 412

Topics covered

this is a partial list

- git and other version control
- bash and other shell scripting
- filesystems
- basics of hardware
- what happens when you compile code
- what are the different types of software on your computer

Catalog Description

How the history and context of computing impacts the practice of computing today. Tools used in programming and computational problem solving. How programming works from high level languages to hardware. Survey of computer hardware and representation of information. Pre: CSC110, any 200 level CSC course, or equivalent.

Learning Outcomes

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

1. Differentiate the different classes of tools used in computer science in terms of their features, roles, and how they interact and justify positions and preferences among popular tools
2. Identify the computational pipeline from hardware to high level programming language
3. Discuss implications of choices across levels of abstraction
4. Describe the context under which essential components of computing systems were developed and explain the impact of that context on the systems.

About this syllabus

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. That will be linked when sovle and you will get a notification at that time.

About your instructor

Name: Dr. Sarah M Brown Office hours: TBA via zoom, link on BrightSpace

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

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

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

Tools and Resources

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

All of the tools and resources below are either:

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

BrightSpace

> **Note**
>
> Seeing the BrightSpace site requires logging in with your URI SSO and being enrolled in the course

This will be the central location from which you can access all other materials. Any links that are for private discussion among those enrolled in the course will be available only from our course Brightspace site .

This is also where your grades will appear and how I will post announcements.

For announcements, you can customize how you receive them.

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.

Course Manual

The course manual 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. In order to use the command line with https, you will need to create a Personal Access Token for each device you use. In order to use the command line with SSH, set up your public key.

Programming Environment

In this course, we will use several programming environments. In order to 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)
- Git
- A bash shell
- A web browser compatible with Jupyter Notebooks
- nano text editor

> **Note**
>
> all Git instructions will be given as instructions for the command line interface and GitHub specific instructions via the web interface. You may choose to use GitHub desktop or built in IDE tools, but the instructional team may not be able to help.

> **Warning**
>
> Everything in this class will be tested with the up to date (or otherwise specified) version of Jupyter Notebooks. Google Colab is similar, but not the same, and some things may not work there. It is an okay backup, but should not be your primary work environment.

Recommendation:

- Install python via Anaconda
- if you use Windows, install Git and Bash with GitBash (video instructions).
- if you use MacOS, install Git with the Xcode Command Line Tools. On Mavericks (10.9) or above you can do this by trying to run git from the Terminal the very first time.`git --version`
- if you use Chrome OS, follow these instructions:

1. Find Linux (Beta) in your settings and turn that on.
2. Once the download finishes a Linux terminal will open, then enter the commands: `sudo apt-get update` and `sudo apt-get upgrade`. These commands will ensure you are up to date.
3. Install tmux with:

```
sudo apt -t stretch-backports install tmux
```

4. Next you will install nodejs, to do this, use the following commands:

```
curl -sL https://deb.nodesource.com/setup_14.x | sudo -E bash
sudo apt-get install -y nodejs
sudo apt-get install -y build-essential.
```

5. Next install Anaconda's Python from the website provided by the instructor and use the top download link under the Linux options.
6. You will then see a .sh file in your downloads, move this into your Linux files.
7. Make sure you are in your home directory (something like home/YOURUSERNAME), do this by using the `pwd` command.
8. Use the `bash` command followed by the file name of the installer you just downloaded to start the installation.
9. Next you will add Anaconda to your Linux PATH, do this by using the `vim .bashrc` command to enter the .bashrc file, then add the `export PATH=/home/YOURUSERNAME/anaconda3/bin/:$PATH` line. This can be placed at the end of the file.
10. Once that is inserted you may close and save the file, to do this hold escape and type `:x`, then press enter. After doing that you will be returned to the terminal where you will then type the source .bashrc command.
11. Next, use the `jupyter notebook –generate-config` command to generate a Jupyter Notebook.
12. Then just type `jupyter lab` and a Jupyter Notebook should open up.

Video install instructions for Anaconda:

- Windows
- Mac

On Mac, to install python via environment, this article may be helpful

- I don't have a video for linux, but it's a little more straight forward.

Zoom (backup only & office hours only, Spring 2022 is in person)

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 class best if you download the Zoom client on your computer. Please log in and configure your account. Please add a photo of yourself 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.

Schedule

Overview

The following is a rough 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 quetions than we can tackle in one class, but will give the general idea of how the class will go.

This plan accounts for 1 less week than we actually have. We will either go over somewhere or we'll use the last week for sharing projects, reflection, or an additional topic that comes up during the semester.

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 a bit of the history.

How do all of these topics relate?

approximatley two weeks

> **Tip**
>
> We will integrate history throughout the whole course. Connecting ideas to one another, and especially in a sort of narrative form can help improve retention of ideas. My goal is for you to learn.
> We'll also come back to different topics over and over again with a slightly different framing each time. This will both connect ideas, give you chance to practice recalling (more recall practice improves long term rentention of things you learn), and give you a chance to learn things in different ways.

We'll spend a few classes doing an overview where we go through each topic in a litte more depth than an introduction, but not as deep as the rest of the semester. In this section, we will focus on how the different things we will see later all relate to one another more than a deep understanding of each one. At the end of this unit, we'll work on your grading contracts.

We'll also learn more key points in history of computing to help tie concepts together in a narrative.

Topics:

- bash
- man pages (built in help)
- terminal text editor
- git
- survey of hardware
- compilation
- information vs data

What tools do Computer Scientists use?

approximately four weeks

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.

Topics:

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

What Happens When I run code?

approximately five weeks

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

Finalized Order

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

Date	Key Question	Prepation	Activities
2021-01-25	What are we doing this semester?	Create GitHub and Prismia accounts, take stock of dev environments	introductions, tool practice
2021-01-27	How does knowledge work in computing?	Read through the class site, notes, reflect on a thing you know well	course FAQ, knowledge discussion
2021-02-01	How do I use git offline?	review notes, reflect on issues, check environment, map cs knowledge	cloning, pushing, terminal basics
2021-02-03	Why do I need to use a terminal?	review notes, practice git offline 2 ways, update kwl	bash, organizing a project
2021-02-08	What are the software parts of a computer system?	practice bash, contribute to the course site, examine a software project	hardware simulator
2021-02-10	What are the hardware parts of a computer system?	practice, install h/w sim, review memory	hardware simulation
2021-02-15	How does git really work?	practice, begin contract, understand git	grading contract Q&A, git diff, hash
2021-02-17	What happens under the hood of git?	things	git plumbing and more bash (pipes and find)
2021-02-22	Why are git commit numbers so long?	review, map git	more git, number systems
2021-02-24	How can git help me when I need it?	reveiw numbers and hypothesize what git could help with	git merges
2021-03-01	How do programmers build documentation?	review git recovery, practice with rebase, merge, revert, etc; confirm jupyterbook is installed	templating, jupyterbook
2021-03-03	How do programmers auotmate mundane tasks?		shell scripting, pipes, more redirects, grep
2021-03-08	How do I work remotely ?		ssh/ ssh keys, sed/ awk, file permissions
2021-03-10	How do programmers keep track of all these tools?		IDE anatomy
2021-03-22			
2021-03-24			
2021-03-29			
2021-03-31			
2021-04-05			
2021-04-07			
2021-04-12			
2021-04-14			
2021-04-19			
2021-04-21			
2021-04-26			
2021-04-28			

Table 1 Schedule

Grading

This section of the syllabus describes the principles and mechanics of the grading for the course.

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. Differentiate the different classes of tools used in computer science in terms of their features, roles, and how they interact and justify positions and preferences among popular tools
2. Identify the computational pipeline from hardware to high level programming language
3. Discuss implications of choices across levels of abstraction
4. Describe the context under which essential components of computing systems were developed and explain the impact of that context on the systems.

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

Principles of Grading

Learning happens through practice and feedback. My goal as a teacher is for you to learn. The grading in this course is based on your learning of the material, rather than your completion of the activities that are assigned.

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 things.

- Earning a C in this class means you have a general understanding; you will know what all the terms mean and could follow along if in a meeting where others were discussing systems concepts.
- Earning a B means that you could apply the course concepts in other programming environments; you can solve basic common errors without looking much up.
- Earning an A means that you can use knowledge from this course to debug tricky scenarios and/or design aspects of systems; you can solve uncommon error while only looking up specific syntax, but you have an idea of where to start.

No Grade Zone

At the beginning of the course we will have a grade free zone where you practice with both course concepts and the tooling and assingment types to get used to expectations. You will get feedback on lots of work and begin your Know, Want to know, Learned (KWL) Chart in this period.

Grading Contract

In about the third week you will complete, from a provided template, a grading contract. In that you will state what grade you want to earn in the class and what work you are going to do to show that. If you complete all of that work to a satisfactory level, you will get that grade. The grade free zone is a chance for you to get used to the type of feedback in the course and the grading contract template will have example specifications to meet.

The finalized grading contract will include the specification that each piece of work has to adhere to.

All contracts will include maintaining a KWL Chart for the duration of the semester and consistent responses in class.

Notes

- Keep your deeper explorations and more practice task content in your KWL chart repository.
- Link approved PRs to your grading contract for record keeping.

Grading Contract Instructions

> **Important**
>
> this includes minor corrections relative to the readme in the template provided

Grading Policies

Late Work

You will get feedback on items at the next feedback period.

Regrading

Re-request a review on your Feedback Pull request.

For general questions, post on the conversation tab of your Feedback PR with your request.

For specific questions, reply to a specifc comment.

If you think we missed *where* you did something, add a comment on that line (on the code tab of the PR, click the plus (+) next to the line) and then post on the conversation tab with an overview of what you're requesting and tag @brownsarahm

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

General URI Policies

Anti-Bias Statement:

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

Disability Services for Students Statement:

Your access in this course is important. Please send me your Disability Services for Students (DSS) accommodation letter early in the semester so that we have adequate time to discuss and arrange your approved academic accommodations. If you have not yet established services through DSS, please contact them to engage in a confidential conversation about the process for requesting reasonable accommodations in the classroom. DSS can be

reached by calling: 401-874-2098, visiting: web.uri.edu/disability, or emailing: dss@etal.uri.edu. We are available to meet with students enrolled in Kingston as well as Providence courses.

Academic Honesty

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

- Using material, directly or paraphrasing, from published sources (print or electronic) without appropriate citation
- Claiming disproportionate credit for work not done independently
- Unauthorized possession or access to exams
- Unauthorized communication during exams
- 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

URI COVID-19 Statement

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

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

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

If you are unable to attend class, please notify me at brownsarahm@uri.edu. We will work together to ensure that course instruction and work is completed for the semester.

Course Communications

Help Hours

TBA

Day	Time	Location	Host
Tuesday	4-5pm	online	Dr. Brown
Wednesday	1-2pm	online	Dr. Brown
Friday	11am-1pm	online	Mark

Online office hours locations are linked in the #help channel on slack

Tips

For assignment help

- **send in advance, leave time for a response** I check e-mail/github a small number of times per day, during work hours, almost exclusively. You might see me post to this site, post to BrightSpace, or comment on your assignments outside of my normal working hours, but I will not reliably see emails that arrive during those hours. This means that it is important to start assignments early.

Using issues

- use issues for content directly related to assignments. If you push your code to the repository and then open an issue, I can see your code and your question at the same time and download it to run it if I need to debug it
- use issues for questions about this syllabus or class notes. At the top right there's a GitHub logo that allows you to open a issue (for a question) or suggest an edit (eg if you think there's a typo or you find an additional helpful resource related to something)

For E-mail

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

1. Introduction

1.1. What is a System?

> **Tip**
>
> You can contribute or fix things on this page (and anywhere else in this site) by clicking on "suggest an edit" under the GitHub menu in the top right.

1.2. What are we going to learn? and Editing on GitHub

We initialized your KWL Chart. You will keep this chart up to date over the course of the semester. Mostly it will be prompted when you should fill it in, but you can add to it whenever you would like.

> **Further Reading**
>
> GitHub itself provides pretty good documentation, full of screenshots for things in browser. editing a file pull request

1.3. For next class

> **Note**
>
> This section will contain more detail, but a short list of what you need will always be in the schedule

- More practice with GitHub terminology. Accept this assignment, read through it, and follow the instructions at the end.
- Review these notes, bring any questions you have to class
- Read the syllabus, explore this whole website. Bring questions about the course. Be prepared for a scavenger hunt that asks you not to recall every fact about the course, but to know where to find informatio.
- Think about one thing you've learned really well (computing or not) and how do you know that you know it? (bring your example)

1.4. Questions After Class

1.4.1. What physical code will we be writing this semester?

1.4.2. How would committing, pull requests, branches, etc work if you wanted to work on something on your computer?

1.4.3. Why does github seem so simple on a surface level? but to actually use it requires much deeper knowledge…

1.4.4. Will I be able to view my answers on prismia chat for the future? Will it erase?

1.4.5. What about the feedback page? How to merge it or not merge it at all?

2. How does learning and knowledge work in computing?

2.1. Git review

- Make sure you get GitHub terminology down and use this “assignment” to practice.

2.2. Scavenger Hunt

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

Where can you find when office hours are?

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

Where is the regrading policy?

There's a term you don't recognize in an activity, where should you look?

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

2.3. Recall, Systems

"Systems" in computing often refers to all the parts that help make the "more exciting" algorithmic parts work. Systems is like the magic that helps you get things done in practice, so that you can shift your attention elsewhere.

In intro courses, we typically give you an environment to hide all the problems that could occur at the systems level.

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

2.4. Mental Models and Learning

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

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

a mental model is how you think about a concept and your way of relating it. Novices have sparse mental models, experts have connected mental models.

When we first learn new things, we first get the basic concepts down, but we may not know how they relate.

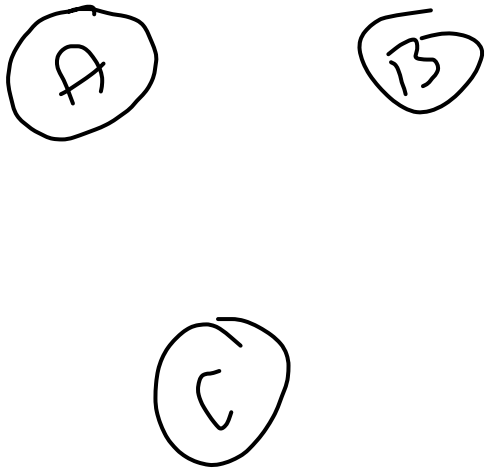


Fig. 2.1 a novice mental model is disconnected and has few concepts

As we learn more, they become more connected.

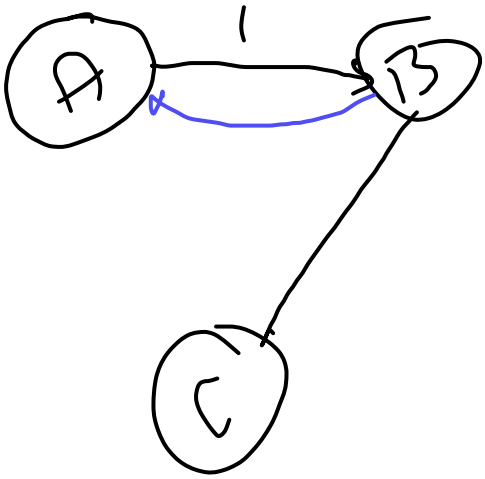


Fig. 2.2 a compententmental model starts to have some connections, with relationships between the concepts.

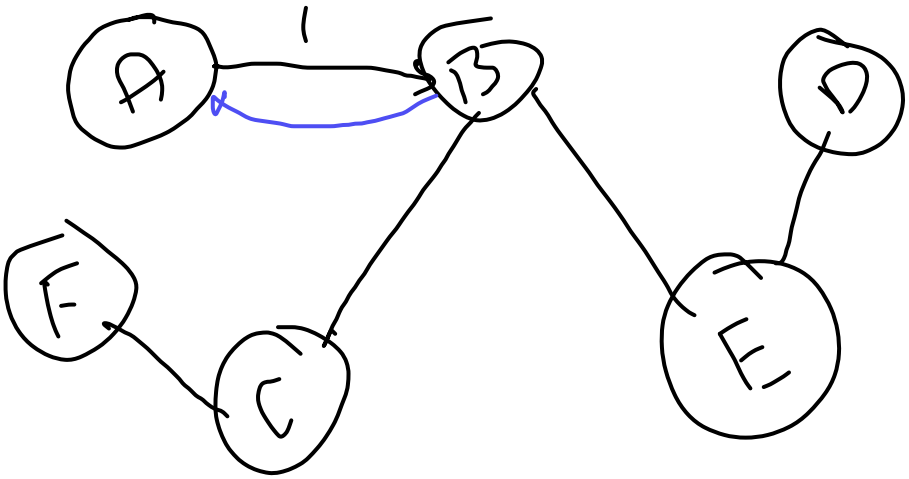


Fig. 2.3 an expert mentla model is densley connected and has more concepts in it.

We can visualize with concept maps. Which connect the ideas using relationships on the arrows.

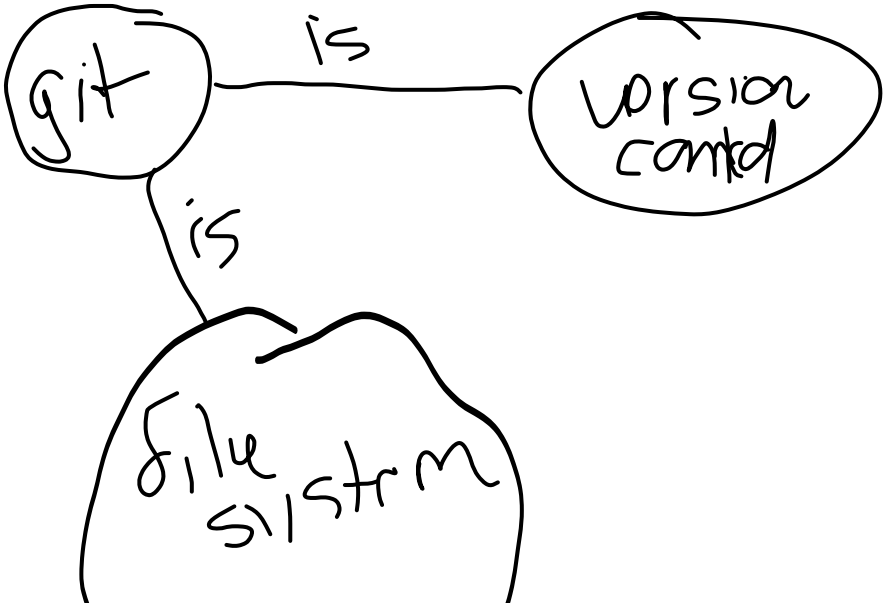


Fig. 2.4 a small concept map showing that git is an instance of both a file system and a version control system.

2.5. Why do we need this for computer systems?

> **Attention**
>
> This section contain points added here that were not discussed directly in class, but are important and will come back up

2.5.1. Systems are designed by programmers

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

> Historically, Computer Science Departments were often initally formed by professors in math creating a new department or, sometimes, making a new degree programs without even creating a new department at first. In some places, CS degree programs also grew within or out of Electrical Engineering. At URI, CS grew out of math.

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

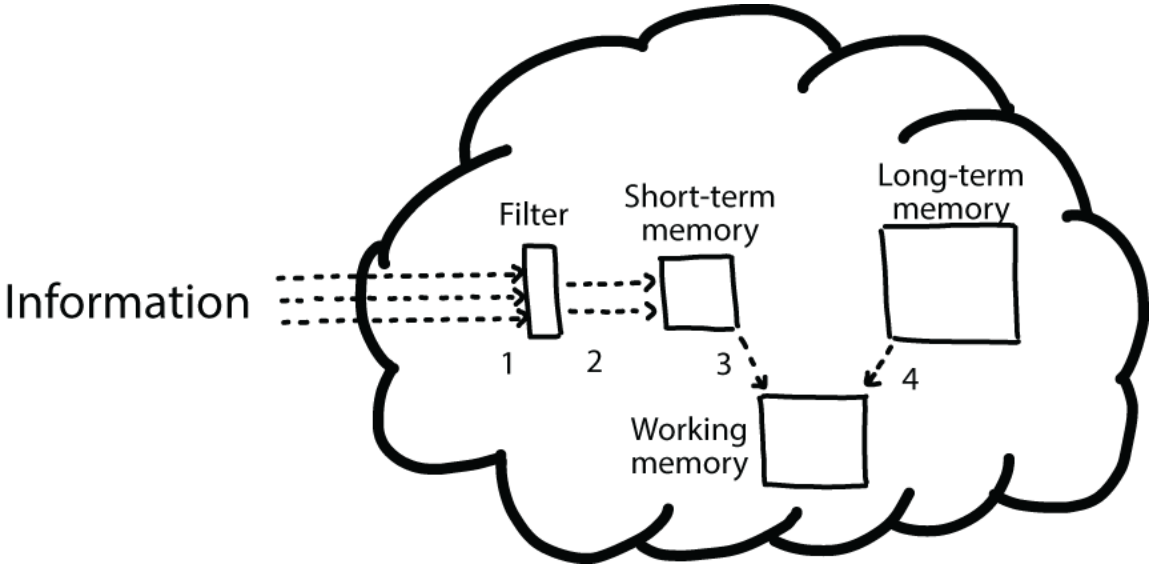


Fig. 2.5 An overview of the three cognitive processes that this book covers: STM, LTM, and working memory. The arrows labeled 1 represent information coming into your brain. The arrows labeled 2 indicate the information that proceeds into your STM. Arrow 3 represents information traveling from the STM into the working memory, where it's combined with information from the LTM (arrow 4). Working memory is where the information is processed while you think about it.

2.5.2. Context Matters

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

2.5.3. Optimal is relative

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

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

2.6. How I expect this to work

2.7. For next class

> **Note**
>
> This is what is required, before the next class and will be checked or if you don't do it you will have trouble participating in class

1. Review these notes, both rendered as html and the raw markdown in the repository.
2. find 2-3 examples of things in programming you have got working, but did not really understand. this could be errors you fixed, or something you just know you're supposed to do, but not why
3. map out your computing knowledge and add it to your kwl chart repo. this can be an image that you upload or a text-based outline.
4. Make sure you have a working environment for next week. Use slack to ask for help.
    - check that you have Python installed with Jupyter, ideally with Anaconda
    - install jupyter book
    - install GitBash on windows (optional for others)
    - make sure you have Xcode on MacOS
    - install the GitHub CLI on all OSs

2.8. More Practice

Note

Activities in this section are optional, but things that may help you prepare, or (in future classes) extend the idea.

- (optional) try mapping out using mermaid syntax, we'll be using other tools that will faciltate rendering later, or try getting it to render on your own.
- (optional) read chapter 1 the programmmer's brain. Some of the ideas we talked about today are mentioned there, and it relates to where you're suppoed to be looking for things that you have done, but didn't really understand.
- try adding something to this page or the glossary of the course site or link a glossary term to an occurence of it on the site.

Hint

terms on this page that could be added to the glossary include filesystem and operating system. The jupyter book docs show how to add to a glossary and link to the glossary from another page.

2.9. Questions After Class

2.9.1. How would I learn more about version control systems?

2.9.2. How to use github to make more meaningful repositories, instead of just a mess of files that are not properly uploaded?

2.9.3. Are there any benefits to using git offline vs using it only in conjunction with github?

2.9.4. When will we be establishing the grade contracts?

2.9.5. How can I be better at communicating documentation

2.9.6. How often are we supposed to update our KWL Charts?

2.9.7. Do we get a notification when you post the notes, or do we just check periodically?

---

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

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

3. How do I use git offline

3.1. Todays Goals

- just enough bash
- offline git basics
- practice with issues as something to *do* while we work with git offline

3.2. Closing an Issue with a commit

We can close issues with commits, we'll first review making commits in browser to see how that works, then we will do it offline again.

Use the create a test repo for today's class it will have some issues it in upon creation.

Notice what happened:

- the file is added and the commit has the the message
- the issue is closed
- if we go look at the closed issues, we can see on the issue that it was linked to the commit
- from the issue, we can see what the changes were that made are supposed to relate to this

> **Note**
>
> we can still comment on an issue that is already closed.

> **Try it Yourself**
>
> We can also re-open issues. Try that out and then make a new commit to close it again. Why is this a useful feature for GitHub?

3.3. Getting Set up Locally

Opening different terminals

- default terminal on mac, ue the `bash` command to use bash (zsh will be mostly the same; it's derivative, but to ensure exactly the same as mine use bash)
- use gitbash on Windows

To change directory

```
cd path/to/go/to
```

To make a directory (folder) for things in this course (or in my case for inclass time)

```
mkdir sysinclass
```

Then we have to `cd` into that new folder:

```
cd sysinclass/
```

To view where we are, we **p**rint **w**orking **d**irectory

```
pwd
```

View files

```
ls
```

It's empty for now, but we will change that soon.

3.4. Using Git and GitHub locally

3.4.1. 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. GitHub CLI: enter the following and follow the prompts.

   ```
   gh auth login
   ```

2. 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)
3. ssh keys

3.4.2. Cloning a repository

Cloning a repository makes a local copy of a remote git repository.

We can clone in two different ways, with git only or with the GitHub CLI tools.

> **Warning**
>
> My repository, like yours, is private so copying these lines directly will not work. You will have to replace my GitHub username with your own.

with the GitHub CLI:

```
gh repo clone introcompsys/github-in-class-brownsarahm
```

with git only:

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

> **Important**
>
> the git only version can be used with git repositories that are hosted anywhere, for example on BitBucket or GitLab

Either way we will see something like this:

```
Cloning into 'github-in-class-brownsarahm'...
remote: Enumerating objects: 15, done.
remote: Counting objects: 100% (15/15), done.
remote: Compressing objects: 100% (9/9), done.
remote: Total 15 (delta 2), reused 5 (delta 1), pack-reused 0
Receiving objects: 100% (15/15), done.
Resolving deltas: 100% (2/2), done.
```

Now we can check what happened using `ls`

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

When we clone a repository, it creates a new directory and downloads all of the contents and the repository information, including where it came from so that we can send our new changes back there.

3.4.3. Adding new files to a Repository Locally

We first go into that folder.

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

We can see what is there.

```
ls
```

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

```
ls
```

and then we see the list of files

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

```
git status
```

which gives us the following output

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

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)
```

```
ls -a
.               .git            README.md
..              .github         about.md
```

```
git add .
```

again, we can check what git knows about

```
git status
```

and take note of the key differences from before.

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

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

git commit -m 'create empty about'
[main b81cf15] create empty about
 1 file changed, 0 insertions(+), 0 deletions(-)
 create mode 100644 about.md
```

3.4.4. Text editing on the terminal

```
nano about.md
```

then we can edit the file, adding some content and then write out (to save) and then exit nano

3.4.5. Commiting Changes to a file

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

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

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

```
git add about.md
```

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

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

```
git commit -m 'complete about closes #2
> '

[main 17320fc] complete about closes #2
 1 file changed, 4 insertions(+)
```

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

nothing to commit, working tree clean
```

3.4.6. Sending Changes to GitHub

```
git push
Enumerating objects: 7, done.
Counting objects: 100% (7/7), done.
Delta compression using up to 8 threads
Compressing objects: 100% (4/4), done.
Writing objects: 100% (6/6), 535 bytes | 535.00 KiB/s, done.
Total 6 (delta 1), reused 1 (delta 0), pack-reused 0
remote: Resolving deltas: 100% (1/1), done.
To https://github.com/introcompsys/github-in-class-brownsarahm.git
   f707186..17320fc  main -> main
```

Notice on GitHub that the issue is now closed. and the commit is referenced and shows the changes.

3.5. A third way to close an issue

See the `classmate` issue:

```
owner:
- [ ] give a class mate access to the repo
- [ ] assign this issue to them

classmate:
- [ ] add `classmate.md` with your name and expected graduation on  a bracn `classmate`
- [ ] open a PR that will close this issue
```

1. Do the `owner` list in your repo (the on that ends with your user name)
2. Do the classmate actions in *another person's repo*

3. In *your own* repo, on the PR made by your class mate, tag `@sp21instructors` in a comment and then merge the PR.

3.5.1. Controlling Access

When you are a repository owner or organization level admin, you can change who has access to a repository on the settings tab.

Since the course repositories are in an organization, you get to choose the role for each collaborator or team. The GitHub Docs define the roles and permissions, so you can always refer t that to choose the right one.

> **Note**
>
> I seeded these notes by using the `Export text` option from the mac terminal app. Other terminals have similar options and you can always get *only* the list of commands you have run with `history`

3.6. Prepare for next class

> **Important**
>
> If you do the last two after Thursday that's okay but the rest are important practice to reinforce what you just learned and to prepare for what we will do in class on Thursday.
>
> > **Tip**
> >
> > The "prepare for class" section is the minimum, the "more practice" is more like what will be reqiured for a B, but there's ramp up time at the beginning.
> > "Prepare for class" will generally be due at the time of the next class because we will use that stuff in the next class session. "More practice" you can come back to as fits in your schedule, (though I don't recommend too big of a lag).

1. Complete the classmate issue in your in-class repository.
2. read the notes PR, add or comment on a tip, resource, a bit of history in a sidebar or additional end of class question
3. try using git in your IDE of choice, log any challenges you have on the practice repo (`github-in-class-username`), and tag @sp22instructors on GitHub. You can use either repo we have made in class, or one for an assignment in another course.
4. using your terminal, download your KWL repo and update your 'learned' column on a new branch
5. answer the questions below in a new markdown file, `gitoffline.md` in your KWL on your new branch and push the changes to GitHub
6. Create a PR from your new branch to main **do not merge this until instructed**
7. add your programming challenge(s) you have had as issues to our private repo or to the course website repo if you like. Put one 'challenge/question' per issue so that we can close them as addressed. See last class notes for prompt.
8. Create or comment on a discussion thread in the private repo about the part of CS/ type of programming you like best/what you want to do post graduation.

> **Tip**
>
> remember you can copy text from here directly into your file

Questions:

```
## Reflection
1. Describe the staging area (what happens after git add) in your own words. Can you think of an analogy for it? Is there anything similar in a hobby you have?
2. what step is the hardest for you to remember?
3. Compare and contrast using git on the terminal and through your IDE. when would each be better/worse?
4. Describe the commit that closed the `classmate` issue, who does it attribute the fix to?
```

3.7. More Practice

1. Find the "Try it yourself" boxes in these notes, try them and add notes/ responses under a `## More Practice` heading in your `gitoffline.md` file of your KWL repo.
2. Download the course site repo via terminal.
3. Explore the difference between `git add` and `git commit` try committing and pushing without adding, then add and push without committing. Describe what happens in each case in your `gitoffline.md`

3.8. Questions After class

3.8.1. Can we push the file before commit? And if so, what will happen?

●●●

3.8.2. what is the difference between add and commit? They seem to do the same thing to me.

●●●

3.8.3. Is there a point to doing multiple commits before a push.

●●●

3.8.4. Can I just use GitHub Desktop if I have it already?

●●●

3.9. Resources:

What is Git?

Interactive Git Cheat Sheet

4. Why Do I Need to Use a terminal?

We will go back to the same repository we worked with on Tuesday, for me that was

```
cd Documents/teaching/sysinclass/github-in-class-brownsarahm/
```

We can use `touch` that we saw Tuesday to create many files at once:

```
touch abstract_base_class.py helper_functions.py important_classes.py alternative_classes.py README.md LICENSE.md CONTRIBUTING.md
setup.py tests_abc.py test_help.py test_imp.py test_alt.py overview.md API.md _config.yml
_toc.yml philosophy.md example.md Untitled.ipynb Untitled01.ipynb Untitled02.ipynb
```

we got an error from this:

```
-bash: _toc.yml: command not found
```

we can intermet this, bash thought `_toc.yml` was a command. That means there was a hard to see accidental line break in the text above.

If we didn't know what that meant, we could also investigate further using `ls` to list.

```
ls
```

we see we have most of the files actually created,

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

we can use the up arrow key to get back the last line.

```
_toc.yml philosophy.md example.md Untitled.ipynb Untitled01.ipynb Untitled02.ipynb
```

and add `touch` at the start of it to create those last few files.

```
touch _toc.yml philosophy.md example.md Untitled.ipynb Untitled01.ipynb Untitled02.ipynb
```

and confirm

```
ls
```

4.1. Scenario

> **Note**
>
> a few of you asked about learning 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

Now we have all of these files, named in abstract ways to signal hypothecial contents and suggest how to organize them.

```
API.md                  _toc.yml                philosophy.md
CONTRIBUTING.md         about.md                setup.py
LICENSE.md              abstract_base_class.py  test_alt.py
README.md               alternative_classes.py  test_help.py
Untitled.ipynb          example.md              test_imp.py
Untitled01.ipynb        helper_functions.py     tests_abc.py
Untitled02.ipynb        important_classes.py
_config.yml             overview.md
```

First we're goign to paste some contents (shared from prismia, view below) in to the readme with nano

```
nano README.md
```

We can view the contents of the file using `cat` to print the contents to the terminal output.

```
cat README.md
```

and we see:

```
# GitHub Practice

Name: sarah
|file | contents |
| ------| ------- |
| abstract_base_class.py | core abstract classes for the project |
| helper_functions.py | utitly funtions that are called by many classes |
| important_classes.py | classes that inherit from the abc |
| alternative_classes.py | classes that inherit from the abc |
| LICENSE.md | the info on how the code can be reused|
| CONTRIBUTING.md | instructions for how people can contribute to the project|
| setup.py | file with function with instructions for pip |
| tests_abc.py | tests for constructors and methods in abstract_base_class.py|
| tests_helpers.py | tests for constructors and methods in helper_functions.py|
| tests_imp.py | tests for constructors and methods in important_classes.py|
| tests_alt.py | tests for constructors and methods in alternative_classes.py|
| API.md | jupyterbook file to generate api documentation |
| _config.yml | jupyterbook config for documentation |
| _toc.yml | jupyter book toc file for documentation |
| philosophy.md | overview of how the code is organized for docs |
| example.md | myst notebook example of using the code |
| Untitled*.ipynb | jupyter notebook from dev, not important to keep |
```

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.

4.2. Making Directories

First we will make directories. We saw `mkdir` on Tuesday

```
mkdir docs/
```

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

```
ls
```

```
API.md                  _toc.yml                overview.md
CONTRIBUTING.md         about.md                philosophy.md
LICENSE.md              abstract_base_class.py  setup.py
README.md               alternative_classes.py  test_alt.py
Untitled.ipynb          docs                    test_help.py
Untitled01.ipynb        example.md              test_imp.py
Untitled02.ipynb        helper_functions.py     tests_abc.py
_config.yml             important_classes.py
```

We might not want to make them all one at a time. Like with `touch` we can pass multiple names to mkdir with spaces between to make multiple at once.

```
mkdir tests mymodule
```

and again use `ls` to see the output

```
API.md                  about.md                philosophy.md
CONTRIBUTING.md         abstract_base_class.py  setup.py
LICENSE.md              alternative_classes.py  test_alt.py
README.md               docs                    test_help.py
Untitled.ipynb          example.md              test_imp.py
Untitled01.ipynb        helper_functions.py     tests
Untitled02.ipynb        important_classes.py    tests_abc.py
_config.yml             mymodule
_toc.yml                overview.md
```

4.3. Moving files

we can move files with `mv`. We'll first move the philosophy.md file into docs and check that it worked.

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

```
API.md                  _toc.yml                mymodule
CONTRIBUTING.md         about.md                overview.md
LICENSE.md              abstract_base_class.py  setup.py
README.md               alternative_classes.py  test_alt.py
Untitled.ipynb          docs                    test_help.py
Untitled01.ipynb        example.md              test_imp.py
Untitled02.ipynb        helper_functions.py     tests
_config.yml             important_classes.py    tests_abc.py
```

4.3.1. Getting help in bash

To learn more about the mv command, we can use the man(ual) file.

```
man mv
```

use enter/return or arrows to scroll and `q` to quit

If we type something wrong, the error message also provides some help

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

We can use man on any bash command to see the options so we do not need to remember them all, or go to the internet every time we need help. We have high quality help for the details right in the shell, if we remember the basics.

```
man ls
```

```
ls -hl
```

```
total 16
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 API.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 CONTRIBUTING.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 LICENSE.md
-rw-r--r--  1 brownsarahm  staff   1.2K Feb  3 12:56 README.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:52 Untitled.ipynb
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:52 Untitled01.ipynb
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:52 Untitled02.ipynb
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 _config.yml
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:52 _toc.yml
-rw-r--r--  1 brownsarahm  staff    14B Feb  1 13:23 about.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 abstract_base_class.py
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 alternative_classes.py
drwxr-xr-x  3 brownsarahm  staff    96B Feb  3 13:04 docs
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:52 example.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 helper_functions.py
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 important_classes.py
drwxr-xr-x  2 brownsarahm  staff    64B Feb  3 13:01 mymodule
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 overview.md
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 setup.py
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 test_alt.py
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 test_help.py
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 test_imp.py
drwxr-xr-x  2 brownsarahm  staff    64B Feb  3 13:01 tests
-rw-r--r--  1 brownsarahm  staff     0B Feb  3 12:51 tests_abc.py
```

In some versions of bash we can use:

```
mv --help
```

4.3.2. Moving multiple files with patterns

let's look at the list of files again.

```
cat README.md
```

```
# GitHub Practice

Name: sarah
|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|
| test_helpers.py | tests for constructors and methods in helper_functions.py|
| test_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 |
```

> **Note**
>
> this is why good file naming is important even if you have not organized the whole project yet, you can use the good conventions to help yourself later.

We see that the ones with similar purposes have similar names.

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

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

4.3.3. Renaming a single file with mv

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

We can use `mv` to change the name as well. This is because “moving” a file and is really about changing its path, not actually copying it from one location to another and the file name is a part of the path.

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

now that it’s fixed

```
API.md                  abstract_base_class.py  setup.py
CONTRIBUTING.md         alternative_classes.py  test_abc.py
LICENSE.md              docs                    test_alt.py
README.md               example.md              test_help.py
Untitled.ipynb          helper_functions.py     test_imp.py
Untitled01.ipynb        important_classes.py    tests
Untitled02.ipynb        mymodule
about.md                overview.md
```

We can use the pattern `test_*` to move them all.

```
mv test_* tests/
ls
```

```
API.md                  Untitled02.ipynb        helper_functions.py
CONTRIBUTING.md         about.md                important_classes.py
LICENSE.md              abstract_base_class.py  mymodule
README.md               alternative_classes.py  overview.md
Untitled.ipynb          docs                    setup.py
Untitled01.ipynb        example.md              tests
```

Now we can move all of the other .py files to the module

```
mv *.py mymodule/
ls
```

```
API.md                  Untitled01.ipynb        mymodule
CONTRIBUTING.md         Untitled02.ipynb        overview.md
LICENSE.md              about.md                tests
README.md               docs
Untitled.ipynb          example.md
```

4.4. Working with relative paths

Let’s review our info again

```
cat README.md
# GitHub Practice

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

We've made a mistake, `setup.py` is actually instructions that need to be at the top level, not inside the module's sub directory.

We can get it back using the relative path to the file and then using `.` to move it to where we "are" sicne we are in the top level directory still.

```
mv mymodule/setup.py .
ls
```

```
API.md                  Untitled01.ipynb        mymodule
CONTRIBUTING.md         Untitled02.ipynb        overview.md
LICENSE.md              about.md                setup.py
README.md               docs                    tests
Untitled.ipynb          example.md
```

Or, if we put it back temporarily

```
mv setup.py mymodule/
```

We can cd to where we put it

```
cd mymodule/
ls
```

```
abstract_base_class.py  helper_functions.py     setup.py
alternative_classes.py  important_classes.py
```

and move it up a level using `..`

```
mv setup.py ..
ls
```

```
abstract_base_class.py  helper_functions.py
alternative_classes.py  important_classes.py
```

then the `..` to go up a level gets us back to where we were.

```
cd ..
ls
```

```
API.md                  Untitled01.ipynb        mymodule
CONTRIBUTING.md         Untitled02.ipynb        overview.md
LICENSE.md              about.md                setup.py
README.md               docs                    tests
Untitled.ipynb          example.md
```

Now we'll move the last few docs files.

```
mv API.md docs/
mv example.md docs/
mv overview.md docs/
ls
```

4.5. Removing files

We still have to deal with the untitled files that we know we don't need any more.

```
CONTRIBUTING.md         Untitled01.ipynb        mymodule
LICENSE.md              Untitled02.ipynb        setup.py
README.md               about.md                tests
Untitled.ipynb          docs
```

we can delete them with `rm` and use `*` to delet them all.

```
rm Untitled*
```

```
ls
```

now we have a nice clean repository.

```
CONTRIBUTING.md README.md       docs            setup.py
LICENSE.md      about.md        mymodule        tests
```

4.6. Copying

The typical contents of the README we would also want in the documentation website. We might add to the file later, but that's a good start. We can do that by copying.

When we copy we designate the file to copy and a path/name for the copy we want to make.

```
cp README.md docs/index.md
cd docs/
ls
```

```

API.md          _toc.yml        index.md        philosophy.md
_config.yml     example.md      overview.md
```

we can check the contents of the file too:

```
cat index.md
```

```
# GitHub Practice

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

it matches .

4.7. More relative paths

We need a `__init__.py` in the mymodule directory but we are in the docs directory currently. No problem!

```
touch ../mymodule/__init__.py
```

```
git status
```

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

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        ../CONTRIBUTING.md
        ../LICENSE.md
        ./
        ../mymodule/
        ../setup.py
        ../tests/

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

note we have both changes and untracked files… but wherre is the docs folder?

we're in there so all of the files are listed relative to there, so it's the `./` line to say that we are currently in an untracked directory. Git status doesn't look inside directories it doens't know if it should track or not.

if we go back to the top level

```
cd ..
```

```
git status
```

> **Tip**
>
> Compare these two outputs carefully. Getting used to noticing these details will help you get yourself unstuck!

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

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        CONTRIBUTING.md
        LICENSE.md
        docs/
        mymodule/
        setup.py
        tests/

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

Then we can add the changes and push

```
git add .
git commit -m 'insclass 2-3'
```

```
[main c15cf43] insclass 2-3
 20 files changed, 41 insertions(+)
 create mode 100644 CONTRIBUTING.md
 create mode 100644 LICENSE.md
 create mode 100644 docs/API.md
 create mode 100644 docs/_config.yml
 create mode 100644 docs/_toc.yml
 create mode 100644 docs/example.md
 create mode 100644 docs/index.md
 create mode 100644 docs/overview.md
 create mode 100644 docs/philosophy.md
 create mode 100644 mymodule/__init__.py
 create mode 100644 mymodule/abstract_base_class.py
 create mode 100644 mymodule/alternative_classes.py
 create mode 100644 mymodule/helper_functions.py
 create mode 100644 mymodule/important_classes.py
 create mode 100644 setup.py
 create mode 100644 tests/test_abc.py
 create mode 100644 tests/test_alt.py
 create mode 100644 tests/test_help.py
 create mode 100644 tests/test_imp.py
```

```
git push
```

```
Enumerating objects: 9, done.
Counting objects: 100% (9/9), done.
Delta compression using up to 8 threads
Compressing objects: 100% (6/6), done.
Writing objects: 100% (7/7), 1.22 KiB | 1.22 MiB/s, done.
Total 7 (delta 0), reused 1 (delta 0), pack-reused 0
To https://github.com/introcompsys/github-in-class-brownsarahm.git
   17320fc..c15cf43  main -> main
```

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

4.8. Git order of operations

above since we didn't make a branch we pushed to main.

```
git status
```

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

nothing to commit, working tree clean
```

We could make the file changes first and then make the branch we want to commit them too as well. it's best to make the branch first so you don't forget, but it is an option

```
touch test_file.md
```

```
git status
```

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

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

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

```
git checkout -b test
```

```
git status
```

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

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

4.9. Recap

Why do I need a terminal

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

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

This week we saw two really important tools. Next week we're going to take a sort of archealogical look at computer systems, first software then hardware to wrap up our overivew and exploration of what all these topics we're going to cover in class are and how they relate.

4.10. Prepare for the next class

1. Review the notes
2. Reorganize a folder on your computer (good candidate may be desktop or downloads folder), using only a terminal to make new directories, move files, check what's inside them, etc. Answer reflection questions (will be in notes) in a new file, `terminal.md` in your kwl repo.
3. Add a glossary to the site to define a term or cheatsheet entry to describe a command that we have used so far.
4. Examine a large project you have done or by finding an open source project on GitHub. Answer the reflection questions in `software.md` in your kwl repo. (will be in notes)

4.10.1. Terminal File moving reflection

Start with a file explorer open, but then try to close it and use only command line tools to explore and make your choices

```
1. Did this get easier toward the end?
1. Use the history to see which commands you used and how many times each, make a table below.
1. Did you have to look up how to do anything we had not done in class?
1. When do you think that using the terminal will be better than using your GUI file explorer?
1. What questions/challenges/ relfections do you have after this?
1. What kinds of things might you want to write a bash script for given what you know in bash so far? come up with 1-2 scenarios
```

4.10.2. Software Reflection

```
1. link to public repo if applicable or title of your project
1. What types of files are there that are not code?
1. What different types of code files are in the project? Do they serve different goals?
1. Is it all in one language or are there multiple?
1. Try to figure out (remember) how the project works. What types of things, without running the code can you look at at a high level?
```

4.11. More Practice

1. Try to do as many things as possible on the terminal for a whole week.
2. Make yourself a bash cheatsheet (and/or contribute to one on the course site)
3. Read through part 1 of the programmer's brain and try the exercises, especially in chapter 4.

4.12. Questions at the end of class

4.12.1. what makes a file be able to stage? In other words, what does “staging” actually mean?

•••

4.12.2. how do you make a branch on the GitHub site?

•••

4.12.3. how do we write a bash script?

•••

4.12.4. Where can I learn more about the GitHub flow?

•••

4.12.5. How do we link a project we already have made to a new git repository

•••

4.12.6. what can happen if I moved a file but I had another file pointing to the old address

•••

4.12.7. how in depth will our bash scripting go in this class? clearly it can be used for a lot of different things

•••

4.12.8. How might you go about re-instantiating a repo? I.e. starting back from whatever the origin is

•••

4.13. Resources

Bash Cheat Sheet

Alternative Shells

5. Review and Abstraction

5.1. Can I reset a Git repository?

1. Find the hash number for the first commit of your in-class repo.
2. On your terminal, navigate to that repo.
3. Check out that commit `git checkout <paste hash here>`
4. Look back at what happened, using `ls`
5. Make a new branch called 'reset' and push that branch to GitHub.
6. Switch back to the current version of the repo
7. In browser, compare the two branches, visually.

5.2. Moving Files Requires Care

A question from last week was what happens if we move a file to an address where there already is one?

```
touch fa
echo "file one" > fa
cat fa
```

```
echo "file two" > fb
cat fb
```

```
mv fa fb
cat fb
```

5.3. Standard In, Out, and Error

We have been using bash to move files around and explore the system so far. In doing so we have also seen `cat` that we saw would display the contents of a file.

What it actually does is a little bit different. Let's try `cat` without putting a file name after it.

```
cat
```

It waits for us to type, if we type and then press enter, what we typed is displayed and it keeps waiting.

Use control/command + d to exit.

`cat` actually looks at standard input, a special file in our computer that gets the input from the keyboard if we don't tell it otherwise.

```
cat fa
```

is a shortcut basically for

```
cat < fa
```

which says explicitly, get ready to the contents of standard in to standard out and then put the contents of `fa` and put it on standard in. The arrow is called a redirect.

We used `echo` to write to a file above in the little experiment.

```
echo "some text" > a_file
cat a_file
```

and we get output as before

```
some text
```

That line has two new parts both `echo` and the `<` syntax. Let's try `echo` by itself.

```
echo "hello world"
```

and we see

```
hello world
```

Echo puts content on standard out, which is a special file that is by default linked to the display of the terminal. It could have been set elsewhere, and that's what the redirect does.

```
echo "some text" > a_file
cat a_file
```

This sends that text to standard out and redirects standard out to the file `a_file`

```
some text
```

if we use two arrows it will append instead of overwriting.

```
echo "some more text" >> a_file
cat a_file
```

```
some text
some more text
```

```
man echo
```

Name	File descriptor	Description	Abbreviation
Standard input	0	The default data stream for input, for example in a command pipeline. In the terminal, this defaults to keyboard input from the user.	stdin
Standard output	1	The default data stream for output, for example when a command prints text. In the terminal, this defaults to the user's screen.	stdout
Standard error	2	The default data stream for output that relates to an error occurring. In the terminal, this defaults to the user's screen.	stderr

Important

GitBash does not support man the reasons athe developer does not want to are also visible. You can use the help option `-help` try the help command.
The help is slightly different from the man pages overall.

Alternatively, you can modify your environment further. Enabling the Windows subsystem for Linux is one option. So is booting into Linux for example ubuntu that is installed on a flash drive. This uses the flas drive as the hard drive for the operating system. This option creates 2 whole "computers" at the software level, that use the same hardware.

5.4. Layers of a Computer System

1. Application
2. Algorithm
3. Programming Language
4. Assembly Language
5. Machine Code
6. Instruction set Architecture
7. Micro Architecture
8. Gates/registers
9. Devices (transistors)
10. Physics

5.5. Prepare for Next Class

1. install h/w simulator
2. Add a glossary, cheatsheet entry, or historical context/facts about the things we have learned to the site.
3. Review past classes prep/more practice and catchup if appropriate
4. Map out how you think about data moving through a small program using the levels of abstraction. Add this to a markdown table in your KWL chart repo called `abstraction.md`. If you prefer a different format than a table, that is okay, but put it in your KWL repo. It is okay if you are not sure, the goal is to think through this.

5.6. More Practice

1. Once your PRs in your KWL are merged so that main and feedback match, pull to updates your local copy. In a new terminal window, navigate there and then move the your KWL chart to a file called `chart.md`. Create a new README files with a list of all the files in your repo. Use `history N` (N is the number of past commands that history will return) and redirects to write the steps you took to `reorg.md`. Review that file to make sure it doesn't have extra steps in it and remove any if needed using nano then commit that file to your repo.
2. find a place where there is a comment in the course notes indicating content to add and submit a PR adding that content. This could be today's notes or a past day's.
3. Add a new file to your KWL repo called `stdinouterr.md` Try the following one at a time in your terminal and describe what happens and explain why or list questions for each in the file. What tips/reminders would you give a new user (or yourself) about using redirects and `echo`?
    - `echo "hello world" > fa > fb`
    - `echo "a test" > fc fd`
    - `> fe echo "hi there"`
    - `echo "hello " > ff world`
    - `<ff echo hello`
    - `fa < echo hello there`
    - `cat`

> **Tip**
>
> pay attention to how many steps you do to know what value of N to use. You should be able to do all of number 1 in your terminal.

> **Hint**
>
> redirects (>) can be used with other commands, not only echo

> **Note**
>
> If you get stuck on any of these create an issue and tag @sp22instructors

5.7. Questions After class

5.7.1. What happens if I don’t meet the requirements for the grade I contract for?

5.7.2. When can we expect approved pull requests?

5.7.3. Can you echo multiple files at the same time?

5.7.4. does this character “<” do something different than the redirect character “>”?

5.7.5. What’s under the hood with >?

5.7.6. What level of understanding of the abstraction stack is typical for a programmer?

5.8. Why you would want to override the name/path of a file?

5.8.1. How does Authentication on GitHub work?

5.9. Resources

Interactive Git Cheat Sheet

6. Survey of Hardware

6.1. Where does assembly come from?

In order to watch what happens in hardware when a program runs, which is our goal today, we will execute a compiled program. It is written in assembly code. Typically, we do not write assembly, but instead it is produced by the compiler.

Technically assembly instructions and the values we operate on are represented in binary on hardware, but these more readable level instructions, though basic, are a useful abstraction. These instructions are also hardware nonspecific.

6.2. Using the simulator

On MacOS:

```
cd path/nand2tetris/tools
bash CPUEmulator.sh
```

On Windows: Double click on the CPUEmulator.bat file

We’re going to use the test cases from the book’s project 5:

1. Load Program
2. Navigate to nand2tetris/projects/05

We’re not going to *do* project 5, which is to build a CPU, but instead to use the test.

For more on how the emulator works see the CPU Emulator Tutorial.

For much more detail about how this all works chapter 4 of the related text book has description of the machine code and assembly language.

6.3. Adding Constants

We'll use add.hack first.

This program adds constants, 2+3.

It is a program, assembly code that we are loading to the simulator's ROM, which is memory that gets read only by the CPU.

Run the simulator and watch what each line of the program does.

Notice the following:

- to compute with a constant, that number only exists in ROM in the instructions
- to write a value to memory the address register first has to be pointed to what where in the memory the value will go, then the value can be sent there

> **Try it yourself**
>
> Write code in a high level language that would compile into this program. Try writing two different variations.

6.4. Using a variable

Next use the max.hack.

This one compares the values at two memory locations. In order to use it, you have to write values to the RAM 0 and RAM 1 manually.

It first takes the value from each location and passes the first value to D, then uses the ALU to assign the difference between the two values to D. Then, if the value is greater than 0, it jumps to line to line 10 in the ROM (of the instructions). Line 10, sets A to 0 and 11 sets D to the value from RAM 0. If, instead, the value is less than 0, A is set 1, then and D is set to that value. Then the program points A to 2 and writes the value from D there.

> **Try it yourself**
>
> Write code in a high level language that would compile into this program. Try writing multiple different versions.
>
> What does this program assume has happened that it doesn't include in its body.

6.5. Using output

The rect.hack program writes to output, by using a specific memory location that is connected to the output.

> **Try it yourself**
>
> Try working through this program using the tools to understand what it does

6.6. Prepare for Next Class

1. Read these notes and practice with the hardware simulator, try to understand its assembly and walk through what other steps happen. Make notes on what you want to remember most or had the most trouble with in `hardwaresurvey.md`
2. Review and update your listing of how data moves through a program in your `abstraction.md`. Answer reflection questions below.
3. Review the commit history and git blame of a repo in browser- what must be in the .git directory for GitHub to render all of that information? (be prepared to discuss this in class)
4. Fill in the Know and Want to know columns for the new KWL chart rows below.
5. Begin your grading contract, bring questions to class Tuesday.

6.6.1. Abstraction reflection

```
1. Did you initially get stuck anywhere?
1. How does what we saw with the hardware simulators differ from how you had thought about it before?
1. Are there cases where what you previously thought was functional for what you were doing but not totally correct? Describe them.
```

6.6.2. New KWL chart rows

```
|file system | _ | _ |_ |
|bash | _ | _ | _ |
|abstraction | _ | _ | _ |
|programming languages | _ | _ | _ |
```

6.7. More Practice

1. Complete the `Try it Yourself` blocks above in your `hardwaresurvey.md`.
2. Expand on your update to `abstraction.md`: Can you reconcile different ways you have seen memory before?

6.8. Questions After Class

6.8.1. Is assembly then converted to binary and if so, why isn't the code translated straight to binary instead of from code to assembly to binary?

6.8.2. What does A mean in CPU Emulator?

6.8.3. When the prepare for next class says things like "organize your thoughts on …" is it expected that we make a .md file somewhere on GitHub to show this work?

7. What actually *is* git?

7.1. Grading Contract Q & A

Grading contract information is added to the syllabus

and the FAQ section

7.2. Git is a File System with a Version Control user interface

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

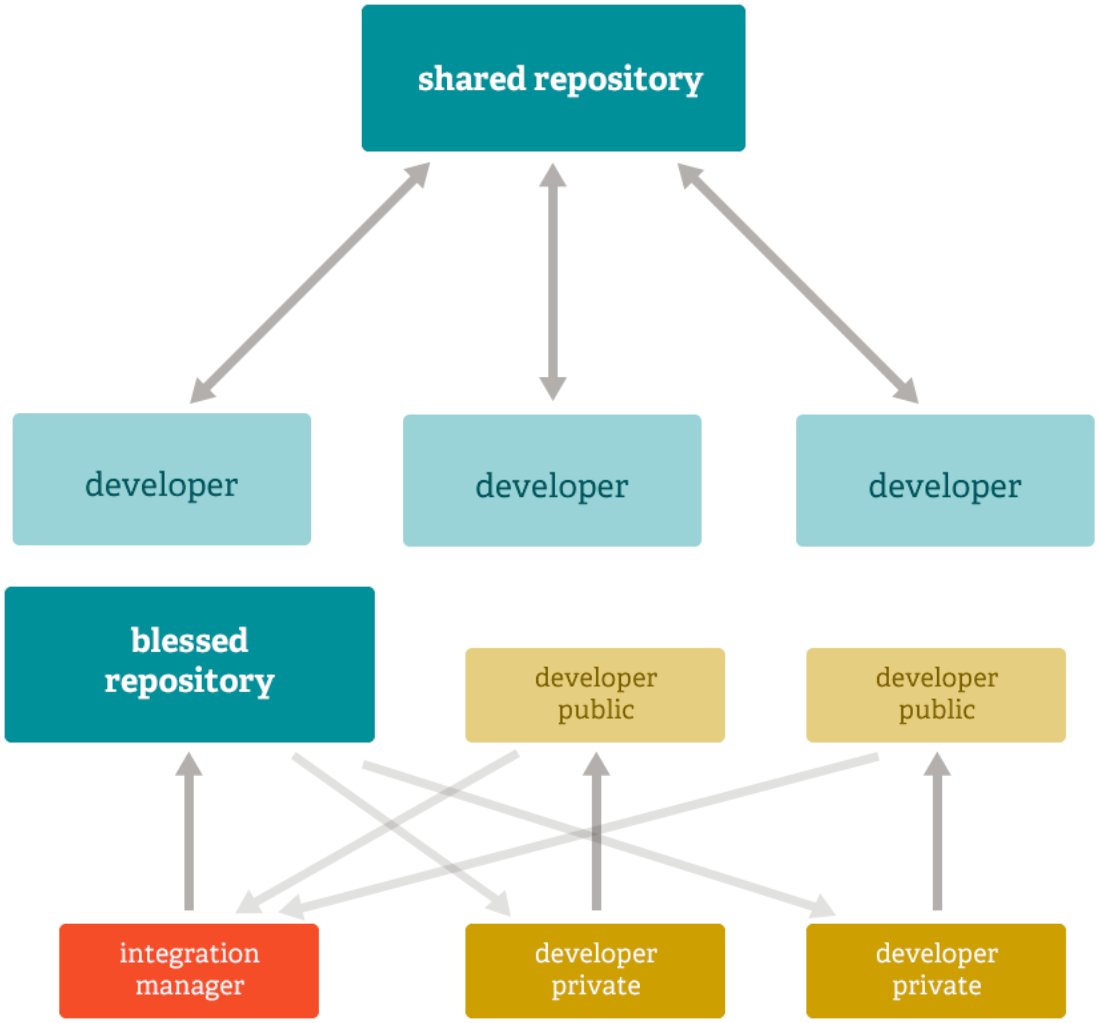
Porcelain: the user friendly VCS

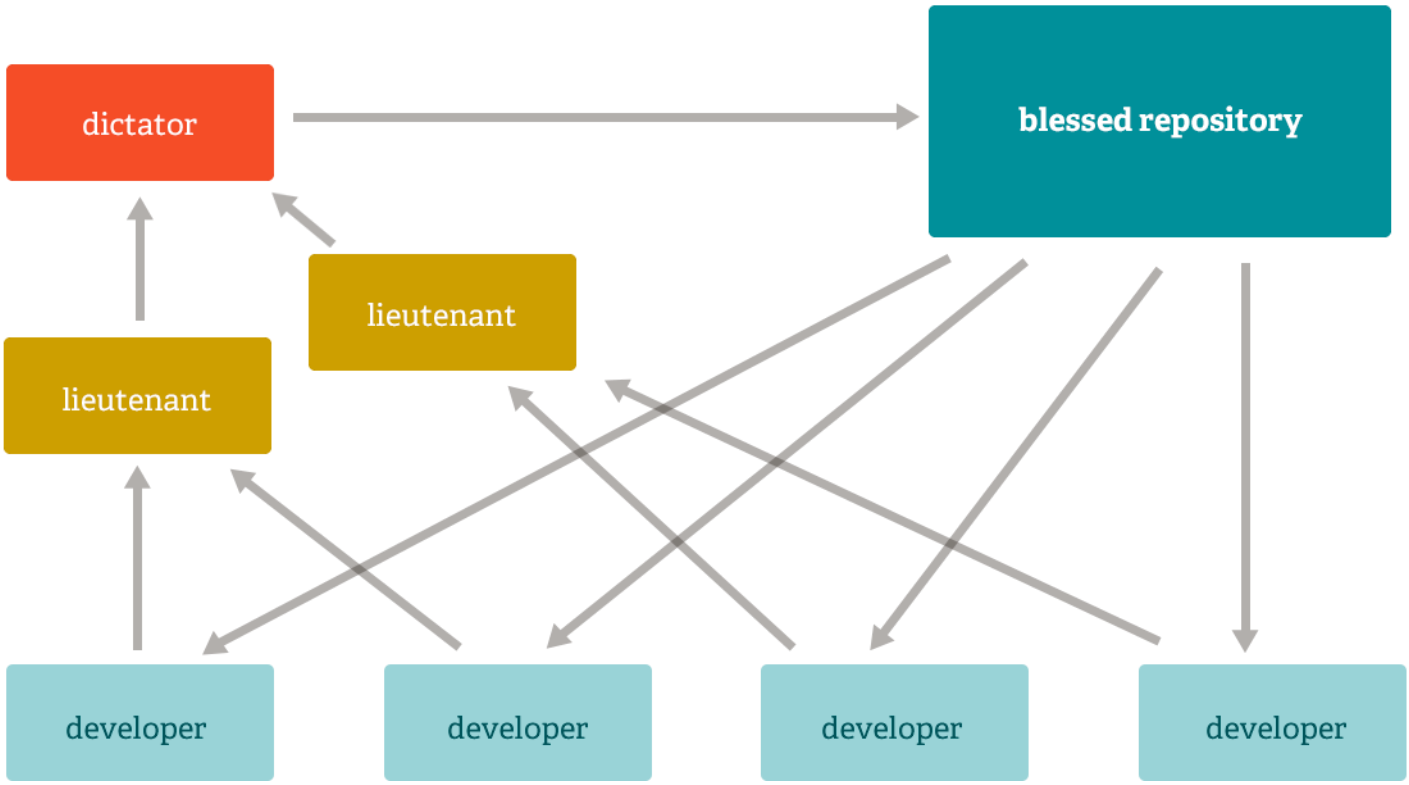
Plumbing: the internal workings- a toolkit for a VCS

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

7.3. Git is distributed

Git can have different workflows





7.4. What are the parts of git?

and we're going to start by examinging our familiar github inclass repo

```
cd path/to/sysinclass/github-in-class-brownsarahm/
ls -a
```

```
.               .github         README.md       b_file          setup.py
..              CONTRIBUTING.md a_file          docs            test_file.md
.git            LICENSE.md      about.md        mymodule        tests
```

We are going to look inside the git folder

```
cd .git
ls
```

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

The most important parts:

name	type	purpose
objects	directory	the content for your database
refs	directory	pointers into commit objects in that data (branches, tags, remotes and more)
HEAD	file	points to the branch you currently have checked out
index	file	stores your staging area information.

7.4.1. Git HEAD

We can look at the head file

```
cat HEAD
```

we see

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

This tells us where in the git history the current status of the repository is.
This is what git uses when we call `git status`. It does more after reading that file, but that is the first thing it does.

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

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

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

Note that the current branch is main and the HEAD file has a path to a file named main in the `refs` directory.

7.4.2. Git Refs

We can loook at that

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

we see the most recent commit hash.

```
cea6a93d576ecd042823fca24553a58a6cd6565b
```

We can verify this with `git log`

when we run this it opens the log interactively, so we see something like:

```
commit cea6a93d576ecd042823fca24553a58a6cd6565b (HEAD -> main, origin/main, origin/HEAD)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 16:42:12 2022 -0500

    try to prevernt repeated running
```

In parenthesis that is what branches are pointed to that commit. Use enter/return to scroll and press `q` to exit.

> **Try it Yourself**
>
> use git log to draw a map that shows where the different branches are relative to one another

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

```
ls
heads   remotes tags
```

```
cd heads/
ls
main    reset   test
```

this has one directory for each branch. we can confirm this with `git branch` at the top level.

```
cd ..
ls remotes/
```

```
origin
```

```
cd remotes/origin/
ls
```

```
HEAD    main    reset
```

```
(base) brownsarahm@origin $ cat HEAD
ref: refs/remotes/origin/main
(base) brownsarahm@origin $ cd main
-bash: cd: main: Not a directory
(base) brownsarahm@origin $ cat main
c15cf43b6807e172aaba7cf3b57adc7214b91082
(base) brownsarahm@origin $ cd ..
(base) brownsarahm@remotes $ cd ..
cd ..
cat config
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
[remote "origin"]
        url = https://github.com/introcompsys/github-in-class-brownsarahm.git
        fetch = +refs/heads/*:refs/remotes/origin/*
[branch "main"]
        remote = origin
        merge = refs/heads/main
[branch "reset"]
        remote = origin
        merge = refs/heads/reset
ls
COMMIT_EDITMSG  config          info            refs
FETCH_HEAD      description     logs
HEAD            hooks           objects
ORIG_HEAD       index           packed-refs
cat ORIG_HEAD
c15cf43b6807e172aaba7cf3b57adc7214b91082
pwd
/Users/brownsarahm/Documents/sysinclass/github-in-class-brownsarahm/.git
cd ../../
pw
```

7.4.3. Git Config and branch naming

```
cd .git
cat config
```

and we see

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

This file tracks the different relationships between your local copy and remots that it knows. This repository only knows one remote, named origin, with a url on GitHub. A git repo can have multiple remotes, each with its own name and url.

it also maps each local branch to its corresponding origin and the local place you would merge to when you pull from that remote branch.

> **Warning**
>
> I remoeved looking at the index here, we're going to come back to it with more time to inspect it more carefully on Thursday

7.4.4. Git Objects

```
cd objects/
ls
```

```
0c      35      55      87      b6      c1      pack
17      45      79      b5      b8      info
```

7.5. Starting a Git Repository from Scratch

We'll create clear repo at the top levle of our inclass directory so that it is not inside another repo

```
cd ..
pwd
```

```
/path/to/Documents/sysinclass
```

then ceate the repo with

```
git init test
```

> **Note**
>
> official statement on git branch naming

GitHub moved a little faster and used a repo to share info about their process
this change was complex and spurred a lot of discussion

```
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/sysinclass/test/.git/
```

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

```
cd test
git branch -m main
```

and we can confirm it works with

```
git status
```

to see that it is now on branch main.

```
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
ls
```

7.6. Prepare for next Class

1. review the notes and ensure that you have a new, empty repository named test with its branch renamed to main from master.
2. Add the following to your kwl:

```
|git workflows | _ | _ | _ |
| git branches | _ | _ | _ |
| bash redirects | _ | _ | _ |
```

3. Practice with git log and redirects to write the commit history for your kwl chart to a file gitlog.txt and commit that file to your repo.

7.7. More Practice

1. ¸ Read about different workflows in git and add responses to the below in a `workflows.md` in your kwl repo. Git Book atlassian Docs
2. Contribute either a glossary term, cheatsheet item, additional resource/reference, or history sidebar to the course website.

```
## Workflow Reflection

1. What advantages might it provide that git can be used with different workflows?
1. Which workflow do you think you would like to work with best and why?
1. Describe a scenario that might make it better for the whole team to use a workflow
other than the one you prefer.
```

7.8. Questions after class

7.8.1. when should the grading contract be turned in?

7.8.2. Can you create multiple remotes that have the same name?

7.8.3. what does it mean when a branch is both x commits ahead of main, but also y commits behind?

7.8.4. Should we be working with Git entirely from the terminal for classwork or is it our preference?

7.8.5. Will we be using the github cli to publish our repo to github?

7.8.6. Questions we will answer in the next couple of classes

1. What is the purpose of the index file?
2. How do you add your local repo to github?
3. What would happen if we would change this hexadecimal code which is in the files in .git?
4. Are any more efficient ways to navigate through repositories and git files

7.9. Resources

- git docs
- git book this includes other spoken languages as well if that is helpful for you.

8. How does git work?

8.1. Review

How can you write the history of a repo to a file?

We'll do this inside our in-class repo.

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

Recall, the `git log` command

```
git log
```

displays it in a text editor:

```
commit cea6a93d576ecd042823fca24553a58a6cd6565b (HEAD -> main, origin/main, origin/HEAD)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 16:42:12 2022 -0500

    try to prevernt repeated running

commit c15cf43b6807e172aaba7cf3b57adc7214b91082 (test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 13:38:42 2022 -0500

    insclass 2-3

commit 17320fc6f26806eb7d1ffc62c23b3bf1361b58b2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 13:32:46 2022 -0500

    complete about closes #2

commit b81cf1525e96782e868e96a20eacf6eb26e882b7
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 13:20:05 2022 -0500
```

use the `q` key to exit

We can use a redirect `>` to send that to a file instead of to where it normally goes.

```
git log > gitlog.txt
```

We can confirm this is the same as we saw before with `cat`

```
cat gitlog.txt
```

```
commit cea6a93d576ecd042823fca24553a58a6cd6565b
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 16:42:12 2022 -0500

    try to prevernt repeated running

commit c15cf43b6807e172aaba7cf3b57adc7214b91082
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 13:38:42 2022 -0500

    insclass 2-3

commit 17320fc6f26806eb7d1ffc62c23b3bf1361b58b2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 13:32:46 2022 -0500

    complete about closes #2

commit b81cf1525e96782e868e96a20eacf6eb26e882b7
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 13:20:05 2022 -0500

    create empty about

commit f707186cd978072d2888b0d2112023b5618b6414
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 12:52:28 2022 -0500

    create readme closes #3

commit 611180dd89ffd8977dd9e5c502bcd4147c4980be
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Feb 1 17:45:49 2022 +0000

    Setting up GitHub Classroom Feedback

commit 3fbd9ee19ff64241b102fa61f279e9c4683d0e69
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Feb 1 17:45:49 2022 +0000

    GitHub Classroom Feedback

commit 270a43daebe1ba7bb719f141b327ca35e8e187ad
Author: github-classroom[bot] <66690702+github-classroom[bot]@users.noreply.github.com>
Date:   Tue Feb 1 17:45:47 2022 +0000

    Initial commit
```

8.2. Review: Anatomy of git

Recall that the `HEAD` file contains a pointer to the place in the git database that matches the current status of the directory

```
cat .git/HEAD
```

in this case it points to the head ref called main

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

This is what `git status` uses, we can see that using our first git “plumbing” command

```
git cat-file -p refs/heads/main
```

> **Note**
>
> note that it’s sort of like the bash command cat, but it *only* works on git *objects* the HEAD file is a plain text file that we can view with `cat`

Before we look at the output of this, let’s examine the command:

- git cat-file: displays git content
- -p (pretty print) option figure out the type of content and display it appropriately

```
tree 4cbe76bea90531a07dbc8cc413de26f39994478b
parent c15cf43b6807e172aaba7cf3b57adc7214b91082
author Sarah Brown <brownsarahm@uri.edu> 1643924532 -0500
committer GitHub <noreply@github.com> 1643924532 -0500
gpgsig -----BEGIN PGP SIGNATURE-----

 wsBcBAABCAAQBQJh/Ew0CRBK7hj4Ov3rIwAATFIIAAeEB38i/5hNr1OUfYMJ/1PA
 RKdcQYspkvi70ISXpkLTEvTMwkFfmJO6Y12QqKdy7WJHIeD0qRsigmOv9oUCv2LJ
 YIdd1d6m9gfe6lwrxh5QG3itURfXyhVIff0j8YHYzq0HN7gAjw+s6UHGGjJiR7u4
 Jq01ekrQJ/OrXWmQnr6UWJNqwBx/7rSgeh0yWpx+5f8z+dFp5N57nk7MGZE04YBn
 huJG6lDPRxTHifYwDx1Ib5C0BvrLKQO0j9TvI/86LhcFSyoDBG3/diHokrQO5bdU
 okm+JsNZAx4prZebZ8eKcMEu5YiAX0wllQxrAEBKEUCc3EU2W4MMRFeiCOY3UNI=
 =oLlf
 -----END PGP SIGNATURE-----


try to prevernt repeated running
```

This object has the hash for the tree that this commit is in, it also has the hash of the parent commit to this commit, then author information and the last commit message

8.3. Git Plumbing in a Fresh repo

Recall we used `git init test` to create a new repo on Tuesday, let's got back to that.

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

We can use `git status` to confirm that it is completely blank

```
git status
On branch main

No commits yet

nothing to commit (create/copy files and use "git add" to track)
```

The directory is also empty

```
ls -a
```

Except for the `.git` database

```
.       ..      .git
```

And recall the different files.

```
ls .git/
HEAD            description     info            refs
config          hooks           objects
```

> **Try it yourself**
>
> Try to remember what each of those files/directories does, make a table and try to write each one's role. After, check your answers using Tuesday's notes

8.4. Git Objects

we noted above that the `git cat-file` only works to git objects. Lets see what objects there are. Today we will see three types today:

- 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

First, lets examine the `objects` directory. This time, instead of using `ls` we'll use the `bash` command `find`. This pattern matches and looks for files or directories and lists the results

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

We see that it finds, the `object` directory itself and two subdirectories.

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

Let's look at another option of this command, the `-type` option filters the results and only returns the ones of the desired type. We'll look at only files using `f`

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

In this empty repository, there are no files.

8.4.1. Hashing

Let's create an object. We have mentioned breifely before that git stores data by hashing it, so the plumbing command `git hash-object` does exactly that.

```
echo 'text content' | git hash-object -w --stdin
```

Let's examine this command:

- `git hash-object` by default hashes the return the unique key to that particular content
- the `-w` option then tells git to alsow write that object to the database
- the `--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, with the pipe it passes that to stdin for the `git hash-object` to use

> **Note**
>
> the key is *unique* in that there is a one to one mapping from any particular content to a single key. However, as we saw in class, if we all do the same content, we all get the key same key back

and we see it returns the unique key to us:

```
c182a93374d6b18a87e1f1e8a9a18812639c58c8
```

We can then view the file that was added to the database with `git cat-file` again. Recall the `-p` option, uses information about the file type to display it in an appropriate manner.

```
git cat-file -p c182a93374d6b18a87e1f1e8a9a18812639c58c8
text content
```

So far, we added "text content" to the git database, but nothing in the directory:

```
ls
```

shows us that its an empty directory.

Lets create a file and add that to our git database. We will create the file with echo and a redirect:

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

then we can hash it and write to the database from the file this time.

```
git hash-object -w test.txt
83baae61804e65cc73a7201a7252750c76066a30
```

we can list the directory so and note that more directories have been made:

```
ls .git/objects/
83      c1      info    pack
```

Even better, we can use the `find` command filtered by type that we saw earlier.

```
find .git/objects/ -type f
.git/objects//c1/82a93374d6b18a87e1f1e8a9a18812639c58c8
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
```

We see that this created an additional file for each of the two has objects that we have created.

Let's append more text to the file, recall two `>>` redirects to append instead of overwrite.

```
echo 'version 2' >> test.txt
```

We can confirm this worked as we expected:

```
cat test.txt
version 1
version 2
```

And then has the file

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

and view the hashed content using the key that was returned.

```
git cat-file -p 0c1e7391ca4e59584f8b773ecdbbb9467eba1547
version 1
version 2
```

So far, we have been adding a new hash of the whole file each time. We will see later that git *can* be more efficient than this because this would begin to take a lot of space very quickly.

8.4.2. Tree Objects

The next type of object to inspect is the tree, since the tree is a git object, we can use `git cat-file` again.

```
git cat-file -p main^{tree}
```

but this test repo that we are working with does not have a tree yet, so it

```
fatal: Not a valid object name main^{tree}
```

We can confirm this using git status

```
git status
```

```
On branch main

No commits yet

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

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

> **Important**
>
> This syntax works in `bash` but has common problems in other shells:
> - CMD on Windows: the `^` character is used for escaping, so you have to double it:`git cat-file -p main^^{tree}`.
> - PowerShell: parameters using `{}` characters have to be quoted to parse correctly, so use: `git cat-file -p 'master^{tree}'`.
> - ZSH: the ^ character is used for globbing, so enclose the whole expression in quotes: `git cat-file -p "master^{tree}"`.

In a more complete repo, we can view the tree:

```
git cat-file -p main^{tree}
```

```
040000 tree c91892d93170077fea81f1d8009c4ccc91af1c29    .github
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    CONTRIBUTING.md
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    LICENSE.md
100644 blob 7987a001e70d28376129bfe0538f98f9aa281a55    README.md
100644 blob b5f4fcb8f875fe33781256ebf6fdde7547188d6f    about.md
040000 tree 55651ac0763db741988f02a45842aa020a77c14d    docs
040000 tree 3581dc38ca3929efcbbc6f7ce510ded2c7dd7309    mymodule
100644 blob e69de29bb2d1d6434b8b29ae775ad8c2e48c5391    setup.py
040000 tree 45fcb1dd311e5e45af759cb3627dca5f47f58f04    tests
```

We see a blob for each file and a tree for each directory. The blob objects refer to the last hashed version of the file added to this branch (main)

```
cat HEAD
ref: refs/heads/main
```

8.5. Creating a Commit manually

> **Warning**
>
> this is revised relative to in class

In class we tried to add directly to the commit tree:

```
echo 'first commit' | git commit-tree c182a9
```

but this failed:

```
fatal: c182a93374d6b18a87e1f1e8a9a18812639c58c8 is not a valid 'tree' object
```

because we had skipped a step. We need to create the tree first.

You use this command to artificially add the earlier version of the test.txt file to a new staging area.

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

the `\` allows us to continue typing on a second visual line in one long command. this command puts the hashed object 83baae with file named `test.txt`

- 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 your staging area (you don't even have a staging area set up yet)
- `--cacheinfo` because the file you're adding isn't in your directory but is in your database. Then, you specify the mode, SHA-1, and filename

n this case, you're specifying a mode of 100644, which means it's a normal file.

```
git status
On branch main

No commits yet

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

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

```
git add test.txt
```

```
git status
On branch main

No commits yet

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

> **Note**
>
> We tried viewing the index but it did not work. the documentation shows the specification for the index, it is not readable

```
git commit -m 'first commit'
[main (root-commit) 2948028] first commit
 1 file changed, 2 insertions(+)
 create mode 100644 test.txt
```

```
find .git/objects/ -type f
.git/objects//0c/1e7391ca4e59584f8b773ecdbbb9467eba1547
.git/objects//c1/82a93374d6b18a87e1f1e8a9a18812639c58c8
.git/objects//29/480288d80e3850d6bf7ae170bd3bea5b3164c7
.git/objects//83/baae61804e65cc73a7201a7252750c76066a30
.git/objects//25/8231c1cee8048eef3a8057cfbdab76261277c6
```

```
git cat-file -p main^{tree}
100644 blob 0c1e7391ca4e59584f8b773ecdbbb9467eba1547    test.txt
```

```
git cat-file -p 2948028
tree 258231c1cee8048eef3a8057cfbdab76261277c6
author Sarah M Brown <brownsarahm@uri.edu> 1645121714 -0500
committer Sarah M Brown <brownsarahm@uri.edu> 1645121714 -0500

first commit
```

```
git cat-file -p 258231
100644 blob 0c1e7391ca4e59584f8b773ecdbbb9467eba1547    test.txt
```

8.6. How is git efficient?

```
git gc
Enumerating objects: 3, done.
Counting objects: 100% (3/3), done.
Writing objects: 100% (3/3), done.
Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
```

```
ls
test.txt
```

```
find .git/objects -type f
.git/objects/c1/82a93374d6b18a87e1f1e8a9a18812639c58c8
.git/objects/pack/pack-07b0c08cc0268d55d02ea62ea880c61f810956d8.idx
.git/objects/pack/pack-07b0c08cc0268d55d02ea62ea880c61f810956d8.pack
.git/objects/info/commit-graph
.git/objects/info/packs
.git/objects/83/baae61804e65cc73a7201a7252750c76066a30
```

```
git verify-pack
usage: git verify-pack [-v | --verbose] [-s | --stat-only] <pack>...

    -v, --verbose         verbose
    -s, --stat-only       show statistics only
    --object-format <hash>
                          specify the hash algorithm to use
```

```
git verify-pack -s
usage: git verify-pack [-v | --verbose] [-s | --stat-only] <pack>...

    -v, --verbose         verbose
    -s, --stat-only       show statistics only
    --object-format <hash>
                          specify the hash algorithm to use
```

8.7. Why didn’t git cat-file work on HEAD?

We tried:

```
git cat-file -p HEAD
```

and saw an error

```
fatal: Not a valid object name HEAD
```

To confirm what it was doing wrong, we checked another repository:

```
git init test2
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/sysinclass/test2/.git/
```

```
git cat-file -p HEAD
```

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

```
cd test2
```

```
git cat-file -p HEAD
```

```
fatal: Not a valid object name HEAD
```

We confirmed that this does not work.

This was because I made a mistake, the HEAD file does not need `git cat-file` it is a plain text file, not a git blob (type) object.

8.7.1. Could the lack of pushing be why?

```
cat .git/config
```

lets compare the results of this across two repos first for the small test repo:

```
[core]
        repositoryformatversion = 0
        filemode = true
        bare = false
        logallrefupdates = true
        ignorecase = true
        precomposeunicode = true
```

and for github-inclass:

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

This is all of the difference that is caused by the lack or configuring a remote.

We can view the heads by listing that directory

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

```
main    reset   test
```

8.8. Prepare for next Class

1. Review the notes
2. For the core "Porcelain" git commands we have used (add, commit), make a table of which git plumbing commands they use in `gitplumbing.md` in your KWL repo
3. In a `gitunderstanding.md` list 3-5 items from the following categories (1) things you have had trouble with in git in the past and how they relate to your new understandign (b) things that your understanding has changed based on today's class (c) things about git you still have questions about
4. Make notes on *how* you use IDEs for the next week or so using the template `idethoughts.md` file in the course notes.

8.9. More Practice

1. Add to your `gitplumbing.md` file explanations of the main git operations we have seen (add, commit, push) in your own words in a way that will either help you remember or behow you would explain it to someone else at a high level. This might be analogies or explanations using other progrmaming concepts or concepts from a hobby. Add this under a subheading `##` with a descriptive title (for example "Git In terms of ")
2. For one thing your understanding changed or an open question you, look up or experiment to find the answer

> **Further Reading**
>
> The goal of this exercise is to take an ethnographic approach to understanding the IDE(s) you use most often. We will combine this with a more formal study of them soon. Approaching a topic through multiple lenses can help you understand it better and presenting you, as a group, with multiple ways is a strategy of mine to help make sure that every one of you finds *at least* one way that works for you.
> More on ethnography in CS

```
# IDE Thoughts

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


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

8.10. Questions After Class

8.10.1. when does the grade-free zone end?

8.10.2. Are there other uses for the hashes besides identifying git files in our repositories

8.10.3. what does print pretty / -p mean?

8.10.4. will we ever need to use the plumbing tools in a bind?

8.10.5. Questions addressed above integrated into the narrative

• I am still confused about the idea of the objects and what they do exactly/their purpose

• so all of the plumbing commands we used today are all the parts that are wrapped up in the processes like adding/committing/pushing things to GitHub?

8.10.6. Questions to practice with

- What if we would change the file again (added 3rd line) and committed again. How many hash objects would be created? What connection would there be between the commits?

8.10.7. Questions we will come back to next

- how are hash number formed?
- From init, first commit, to first push, what is the exact sequence of events that happens under the hood of git?

9. How do hashes work in git?

In this class we will cover:

- what type of hashes are used in git
- what a hash is
- number systems

9.1. Git Plumbing Q&A

Plumbing commands work with the .git directory as a file system directly. Porcelain commands provide higher level interactions with git as a version control system.

9.2. What *is* a hash?

- a hash is a fixed size value that can be used to represent data of arbitrary sizes
- the *output* of a hashing function
- often fixed to a hash table

A hasing function could be really simple, to read off a hash table, or it can be more complex.

For example:

Hash	content
0	Success
1	Failure

If we want to represent the status of a program running it has two possible outcomes: success or failure. We can use the following hash table and a function that takes in the content and returns the corresponding hash. Then we could pass around the 0 and 1 as a single bit of information that corresponds to the outcomes.

This lookup table hash works here.

In a more complex scenario, imagine trying to hash all of the new terms you learn in class. A table would be hard for this, because until you have seen them all, you do not know how many there will be. A more effective way to hash this, is to derive a *hashing function* that is a general strategy.

9.3. When does hashing occur 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.

In git, 40 characters that uniquely represent either the content or a commit.

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

> **Note**
>
> You can view commits shorter with options to `git log`
>
> ```
> git log --abbrev-commit --pretty=oneline
> ```

> **Important**
>
> git commits only need to be unique is **per repository** the git program is not searching all git repositories when we run commands that use commits, it is looking only in the local `.git` directory. You could even have two different projects on your computer with an identical hash and that would not be a conflict because git only looks within the current directory.

9.4. What hashing function does git use?

Git uses SHA-1. See a generator

This is a Secure Hashing Algorithm that is derived from cryptography. Because it is secure, no set of mathematical options can directly decrypt an SHA-1 hash. It is designed so that any possible content that we put in it returns a unique key. It uses a combination of bit level operations on the content to produce the unique values.

The SHA-1 Algorithm hashes content into a fixed length of 160 bits. This means it can produce \(2^160\) different hashes. Which makes the probability of a collision very low.

> The number of randomly hashed objects needed to ensure a 50% probability of a single collision is about \(2^{80}\) (the formula for determining collision probability is p = (n(n-1)/2) * (1/2^160)). \(2^{80}\) is 1.2 x 1024 or 1 million billion billion. That's 1,200 times the number of grains of sand on the earth.
>
> —A SHORT NOTE ABOUT SHA-1 in the Git Documentation

This output, 160 bits (a bit is a unit of information in base 2; a 0 or 1) can be interpreted as a number and represented in diffrent ways. Since 160 characters is really long, git represents it as 40 characters in hexadecimal.

However, SHA-1 is subjec to collision attacks.

> We have broken SHA-1 in practice.
>
> …
>
> It is now practically possible to craft two colliding PDF files and obtain a SHA-1 digital signature on the first PDF file which can also be abused as a valid signature on the second PDF file.
>
> …
>
> GIT strongly relies on SHA-1 for the identification and integrity checking of all file objects and commits. It is essentially possible to create two GIT repositories with the same head commit hash and different contents, say a benign source code and a backdoored one. An attacker could potentially selectively serve either repository to targeted users. This will require attackers to compute their own collision. — shattered it

Git switched to hardended SHA-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

and they will change again soon

9.5. What is a number?

a mathematical object used to count, meaure and label

a mathematical object used to count, meaure and label

What types of numbers are you familiar with?

9.6. Number Representation

Numbers are a cultural artifact: We use a base 10 system most commonly becuase we count with our hands and have 10 fingers. Computers use base 2 because they are digital:on & off. The current representation system we use (0,1, 2, 3, 4,5,6,7,8,9) is called hindu-arabic.

9.6.1. 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.

9.6.2. Binary

uses base 2, but the same characters. So the place values are the ones(\(2^0\)), the twos(\(2^1\)), the fours(\(2^2\)), the eights(\(2^3\)), etc.
\[10 => 2*1 + 1*0 = 2\]
so this 10 in binary is 2 in decimal
\[1001 => 8*1 + 4*0 + 2*0 + 1*1 = 9\]
Binary numbers have been discovered in ancient egyptian, chinese and Indian texts.

9.6.3. Octal

uses base 8, but the same characters. So the place values are the ones(\(8^0\)), the eights(\(8^1\)), the 64s (\(8^2\)), etc.
\[10 = > 8*1 + 1*0 = 8\]
so 10 in octal is 8 in decimal

\[401 => 64*4 + 8*0 + 1*1 = 257\]
This numbering system was popular in 6 bit and 12 bit computers, but is has origins before that. Native Americans using the Yuki Language (based in what is now California)used an octal system because they count using the spaces between fingers and speakers of the Pamean languages in Mexico count on knuckles in a closed fist. Europeans debated using decimal vs octal in the 1600-1800s for various reasons because 8 is better for math mostly. It is also found in Chinese texts dating to 1000BC.

9.6.4. Hexadecimal

uses base 16, but the same characters plus letters A-F. The letters fill in for the numbers aft 9 so A is 10, B is 11, etc. So the place values are the ones(\(16^0\)), the sixteens(\(16^1\)), the two hundred fifty sixes(\(16^2\)),etc.
\[10 = > 16*1 + 1*0 = 16\]
so 10 in hex is 16 in decimal
\[E => 1*14 = 14\]
\(CD => 16*12 + 1*13 = 205\)$

There was debate a number of different proposals for how the characters beyond 9 should be represented.

9.6.5. Roman numerals

Use different representation completely. It doesn't have places the same way. But it is still a way to represent numbers.

I = 1, V = 5, X = 10, L = 50, C = 100, D = 500, M =1000

This repesentation concatenates symbols and adds them if they are in desceding order and if a smaller digit before a larger then it is subtracted.

- III = 1 + 1 + 1= 3
- IX = 10 -1 = 9
- XL = 50 - 10 = 40

> **Note**
>
> On a lighter note: passWordle

> **Learn more**
>
> Learning more about other number systems or the history of these number systems is a good topic for a deeper exploration.

9.7. Prepare for next Class

1. review the past two classes of notes
2. find 3 more examples of using other number systems, list them in `numbers.md`
3. Read about hexpeak from Wikipedia for an overview and one additional source. In your kwl repo in `hexspeak.md` summarize it in your own words, one interesting fact from your additional source and link to the source you found. Come up with a word or two on your own.
4. (priority) Bring to class a scenario where you think git could help, but we haven't covered yet how to do it. (be prepared to post it to Prismia at the start of class)

9.8. More Practice

1. Read more about git's change from SHA-1 to SHA-256 and reflect on what you learned (questions provided)
2. (priority) In a language of your choice or pseudocode, write a short program to convert, without using libraries, between all pairs of (binary, decimal, hexidecimal) in `numbers.md`. Test your code, but include in the markdown file enclosed in three backticks so that it is a "code block" write the name of the language after the ticks like:

````
```python
# python code
```
````

9.8.1. transition questions

1. Why is the switch important?
2. Summarize one vulnerability of SHA-1.
3. What impact will the swith have on how git works?
4. If you have scripts that operate on git repos, what might you do to future proof them so that the switch won't break your code.

9.9. Questions After Class

9.9.1. Why did git use SHA-1 isntead of SHA 256 from the start?

9.9.2. If there is ever a class cancellation or any kind of issue with the notes again, will that be emailed to us or only sent through Slack?

9.9.3. Are both git and github switchig from sha1

9.9.4. Are octal numbers ever used in computing?

9.9.5. Questions left as an exercise

> **Warning**
>
> these are for you to look up/hypothesize about and then we will discuss with you in your KWL repo

- What are the possible dangers of git switching to a better encryption scheme immediately?

10. How can git help me?

So far we have seen git add a bunch of steps to a workflow and worked to understand what it is doing. I've *told* you tracks versions and can help you out because keeping track of versions is good.

Today I will *show* you how it can help.

we are going to work in out github-inclass repo today. Let's reveiw its status.

```
git status
```

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

Untracked files:
  (use "git add <file>..." to include in what will be committed)
        a_file
        b_file
        gitlog.txt
        test_file.md

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

In mine, I have these extra files I don't need from testing things. How can I get rid of them since I do not want to keep them?

We see that they are in the files along with the other files that we planned to be here. Remember the "scenario" of this repo is that it has a number of mostly empty files to represent a python project.

```
ls
CONTRIBUTING.md README.md       about.md        docs            mymodule        test_file.md
LICENSE.md      a_file          b_file          gitlog.txt      setup.py        tests
```

Could it be git pull?

```
git pull
```

```
Already up to date.
```

No git pull, gets, from the origin, the most up to date version of the git database (`.git` directory) from the origin (in this case github). It does not, however do anything to untracked files.

We can remove them:

```
rm *_file
rm gitlog.txt
rm test_file.md
```

Then we can confirm the git status of the repo.

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

nothing to commit, working tree clean
```

10.1. What if I make a file that I don't want to commit?

Imagine we made a configuration file that contained some text (for example a personal key of some sort) that we need in the directory to make our code work locally, but that we do not want to push to GitHub?

```
echo "secrets" >> config.txt
```

The way to do that is to not put it into git at all, so that when we push it will not go.

As usual, before we do anything with git, we check the status

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

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

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

In this case, we see that git has the file here, but it is untracked.

Recall, git status has the git program check the `HEAD` file to determine what in the databaes to look at and then compares the current directory status to what the corresponding last hash contained.

Git provides a special file that it will check when we add, and then not add any files that match the files in.

```
nano .gitignore
```

We will add the following to the file:

```
config.*
```

Then write and exit from nano.

Now our status has changed:

```
git status
```

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

We see the config file is not in the untracked anymore (that means it will not be added if we do git add) but the gitignore file is.

We want to commit this file becuase we would want collaborators to share the same ignored patterns.

```
git add .
```

```
git commit -m 'ignore configs'
[main 3c46e01] ignore configs
 1 file changed, 1 insertion(+)
 create mode 100644 .gitignore
```

Now, what is the advantage of using the pattern?

If we make another file that starts with “config”

```
echo "more secrets" >> config.yml
```

and then check the status

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

nothing to commit, working tree clean
```

git does not see it at all.

we can see though that the files are boht there in our directory on disk.

```
ls
```

```
CONTRIBUTING.md README.md       config.txt      docs            setup.py
LICENSE.md      about.md        config.yml      mymodule        tests
```

What happens if we try to explicitly add it?

```
git add config.txt
```

Let's see:

```
The following paths are ignored by one of your .gitignore files:
config.txt
hint: Use -f if you really want to add them.
hint: Turn this message off by running
hint: "git config advice.addIgnoredFile false"
```

git tries to warn us.

> **Important**
>
> In general, if something you are trying to do requires `-f` or `-hard` in git or bash (or others) do not do it unless you are **very** confident that it is the right thing to do.
>
> These options are generally only required for things are either hard or impossible to undo.

10.2. What if I break my code and commit the bad code?

Let's imagine we wrote some code that does not actually work

```
echo "bad code" >> mymodule/important_classes.py
```

but we forgot to test it, so we add

```
git add .
```

and commit it:

```
git commit -m 'best feature ever'
```

now it is in our repository

```
[main 2c1c3e2] best feature ever
 1 file changed, 1 insertion(+)
```

Now we test the code, find out that it does not work. but we remember that it worked before this code we just added. We can view the commit history to get the hash of the last commit that worked.

```
git log
commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:00:50 2022 -0500

    best feature ever

commit 3c46e01c5b29ffa2ed9a76ff4e8c8f334c0c1bf2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 12:54:45 2022 -0500

    ignore configs

commit cea6a93d576ecd042823fca24553a58a6cd6565b (origin/main, origin/HEAD)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 16:42:12 2022 -0500

    try to prevernt repeated running

commit c15cf43b6807e172aaba7cf3b57adc7214b91082 (test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 13:38:42 2022 -0500

    insclass 2-3

commit 17320fc6f26806eb7d1ffc62c23b3bf1361b58b2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Tue Feb 1 13:32:46 2022 -0500
```

and then we can look at the difference between that version of `important_classes.py` and the current version:

```
git diff 3c46e0 important_classes.py
```

Git diff is what GitHub uses on the PR files changed tab. it takes the commit reference first and then the file to look at just the one files changes.

```
diff --git a/mymodule/important_classes.py b/mymodule/important_classes.py
index e69de29..90baf46 100644
--- a/mymodule/important_classes.py
+++ b/mymodule/important_classes.py
@@ -0,0 +1 @@
+bad code
```

From this, we can confirm that the only change in the most recent commit is the bad code, there is not any other good code we want to keep.

We can look back our git log above to get the hash for that last commit and we can use git revert to “undo” it.

```
git revert 2c1c3e
```

Then git asks us to confirm a commit message, and we get output like this, which looks like after a commit.

```
[main 8bb2df9] Revert "best feature ever"
 1 file changed, 1 deletion(-)
```

To see how git revert works lets look at our commit history again.

```
git log
```

```
commit 8bb2df91d371ab3f9245d49aee5bd0e5f95a9294 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:09:56 2022 -0500

    Revert "best feature ever"

    This reverts commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4.

commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:00:50 2022 -0500

    best feature ever

commit 3c46e01c5b29ffa2ed9a76ff4e8c8f334c0c1bf2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 12:54:45 2022 -0500

    ignore configs

commit cea6a93d576ecd042823fca24553a58a6cd6565b (origin/main, origin/HEAD)
Author: Sarah Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 16:42:12 2022 -0500

    try to prevernt repeated running

commit c15cf43b6807e172aaba7cf3b57adc7214b91082 (test)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 3 13:38:42 2022 -0500

    insclass 2-3

...
```

git did not go back in history in this case, it looked at what changes were applied in the commit we were reverting, applied the inverse of them to the current place the head points and added a new commit for that.

To illustrate this more clearly, lets make two commits. First we'll commit some code we will later realize is not good ot helper functions and then something good to alternative classes.

```
echo "heleper code" >> helper_functions.py
git add .
git commit -m 'bad code'
```

we see the message for that

```
[main e9c132f] bad code
 1 file changed, 1 insertion(+)
```

and the second change

```
echo "something good" >> alternative_classes.py
git add .
git commit -m 'some code'
```

```
[main a224c42] some code
 1 file changed, 1 insertion(+)
```

Now we view the commit history again and the code we want to change is one commit back.

```
git log
commit a224c42a9fd5ba19b55d1e5e4f34c411c8d519f3 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:47 2022 -0500

    some code

commit e9c132f6922a4bfae5b21793a9982fe837db6643
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:17 2022 -0500

    bad code

commit 8bb2df91d371ab3f9245d49aee5bd0e5f95a9294
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:09:56 2022 -0500

    Revert "best feature ever"

    This reverts commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4.

commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:00:50 2022 -0500

    best feature ever

commit 3c46e01c5b29ffa2ed9a76ff4e8c8f334c0c1bf2
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 12:54:45 2022 -0500

    ignore configs
```

We can use revert again

```
git revert e9c132
```

it behaves just as before, having us write a commit message and then commiting the “new” changes

```
[main b6437b4] Revert "bad code"
 1 file changed, 1 deletion(-)
```

Now when we view the commit history we see the bad code, the commit we wanted to keep after it and hten the reverted bad code.

```
git log
commit b6437b4174f69d79e769e296c5e0db5f38c78a55 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:14:58 2022 -0500

    Revert "bad code"

    This reverts commit e9c132f6922a4bfae5b21793a9982fe837db6643.

commit a224c42a9fd5ba19b55d1e5e4f34c411c8d519f3
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:47 2022 -0500

    some code

commit e9c132f6922a4bfae5b21793a9982fe837db6643
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:17 2022 -0500

    bad code

commit 8bb2df91d371ab3f9245d49aee5bd0e5f95a9294
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:09:56 2022 -0500

    Revert "best feature ever"

    This reverts commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4.

commit 2c1c3e2d7bebdbd238e517e07142151cdf5256f4
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:00:50 2022 -0500
```

Important

What does this functionality tell us about how we should commit?

10.3. What if I try a bunch of stuff and it doesn't work, but I notice before I commit?

Imagine, you tried out a new idea:

```
echo "test idea" >> mymodule/important_classes.py
```

and then you test and it does not work. You want to put your working directory back to match the last commit in order to start from scratch with a new idea.

```
git status
```

We can see that our new file is not staged for commit and the git status tells us how to put it back.

```
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

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

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

We can use `git restore` with the file name (path) of what to restore

```
git restore mymodule/important_classes.py
```

Now if we try something new again

```
echo "test idea again" >> mymodule/important_classes.py
```

10.4. How can git help me remember what I'm working on?

Imagine you worked some then walked away and forget what you were working on.
A diff can help you see what changes you have made since your las commit.

```
git diff mymodule/important_classes.py
diff --git a/mymodule/important_classes.py b/mymodule/important_classes.py
index e69de29..a8343bc 100644
--- a/mymodule/important_classes.py
+++ b/mymodule/important_classes.py
@@ -0,0 +1 @@
+test idea again
```

Now if we add the file, to commit.

```
git add .
```

If you are about to commit and forget what work you are about to commit or have trouble thinking of what to write as your commit message, you can use `git diff --staged` to see exactly what changes you are about to add.

```
git diff --staged
```

this outputs a diff between the index and the last commit.

```
diff --git a/mymodule/important_classes.py b/mymodule/important_classes.py
index e69de29..a8343bc 100644
--- a/mymodule/important_classes.py
+++ b/mymodule/important_classes.py
@@ -0,0 +1 @@
+test idea again
```

Note that it shows you what you are comparing `e69de29` to `a8343b` and lists the files and both a summary `@@ -0,0 +1 @@`

and the actual line by line changes.

This can be more useful after we ahve made more changes after we staged some.

```
echo "doc new feature" >> README.md
```

and check the diff again

```
git diff --staged
```

we get the same output

```
diff --git a/mymodule/important_classes.py b/mymodule/important_classes.py
index e69de29..a8343bc 100644
--- a/mymodule/important_classes.py
+++ b/mymodule/important_classes.py
@@ -0,0 +1 @@
+test idea again
```

Thisis the same becuase it comares the *staging* area to the last commmit, not the current directory to the last commit.

```
git status
```

This shows that only the important_classes file is staged, not the readme chagnes.

```
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   mymodule/important_classes.py

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
```

10.5. How can update my feature branch when main gets updated?

Imagine you are working on a new feature for a the project, so you create a new branch

```
git checkout -b new_feature
Switched to a new branch 'new_feature'
```

and add your work.

```
echo "new feature" >> mymodule/important_classes.py
```

```
git status
On branch new_feature
Changes to be committed:
  (use "git restore --staged <file>..." to unstage)
        modified:   mymodule/important_classes.py

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
        modified:   mymodule/important_classes.py
```

and we will commit the code here.

```
git add mymodule/important_classes.py
git commit -m 'new feature'
```

```
[new_feature e7a92d2] new feature
 1 file changed, 2 insertions(+)
```

Then you remember you had done work on the README that was supposed to be on main already, so we go back there and commit that file. Imagine this work is a bug fix that is unrelated to new feature and you want users and your co-workers to have before you finish the new feature.

```
git checkout main
M       README.md
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)
```

```
git status
On branch main
Your branch is ahead of 'origin/main' by 6 commits.
  (use "git push" to publish your local commits)

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

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

and commit the readme here.

```
git add .
```

```
git commit -m 'update readme'
[main 07dbc9e] update readme
 1 file changed, 1 insertion(+)
```

We can view commit history to see that these two branches have similar, but not the same history.

```
git log
commit 07dbc9e484b92cdbc7b055883fb5e1bfaee645f1 (HEAD -> main)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:27:33 2022 -0500

    update readme

commit b6437b4174f69d79e769e296c5e0db5f38c78a55
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:14:58 2022 -0500

    Revert "good code"

    This reverts commit e9c132f6922a4bfae5b21793a9982fe837db6643.

commit a224c42a9fd5ba19b55d1e5e4f34c411c8d519f3
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:47 2022 -0500

    some code

commit e9c132f6922a4bfae5b21793a9982fe837db6643
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:17 2022 -0500

    good code

commit 8bb2df91d371ab3f9245d49aee5bd0e5f95a9294
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:09:56 2022 -0500

    Revert "best feature ever"
```

then to the other branch

```
git checkout new_feature
```

```
Switched to branch 'new_feature'
```

and commit history again

```
git log
```

```
commit e7a92d24de2d70c6122e28b55cd9066b504ae8c8 (HEAD -> new_feature)
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:26:40 2022 -0500

    new feature

commit b6437b4174f69d79e769e296c5e0db5f38c78a55
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:14:58 2022 -0500

    Revert "good code"

    This reverts commit e9c132f6922a4bfae5b21793a9982fe837db6643.

commit a224c42a9fd5ba19b55d1e5e4f34c411c8d519f3
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:47 2022 -0500

    some code

commit e9c132f6922a4bfae5b21793a9982fe837db6643
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:12:17 2022 -0500

    good code

commit 8bb2df91d371ab3f9245d49aee5bd0e5f95a9294
Author: Sarah M Brown <brownsarahm@uri.edu>
Date:   Thu Feb 24 13:09:56 2022 -0500

    Revert "best feature ever"
```

Now we want to get the changes from main into the new feature branch because those fixes impact our ability to test the new feature to see if it works right.

We can update the new_feature branch with rebase.

```
git rebase main
Successfully rebased and updated refs/heads/new_feature.
```

Rebase gets the current status of the main branch (which has a mostly shared commit history with `new_feature` and then applies all of the commits that are on `new_feature` but not `main` on top of the most recent version of main.

We can see than now the history of new_feature is everything on main +1 one commit, where before they had shared except the last commit, where each one had a different most recent commit.

Now, if we're done with the new feature and it works, we can merge it into main.

```
git checkout main
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 7 commits.
  (use "git push" to publish your local commits)
```

On main we have no current thigns to commit:

```
git status
On branch main
Your branch is ahead of 'origin/main' by 7 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

Git `merge` allows us to apply the commits from new_feature to main.

```
git merge new_feature
Updating 07dbc9e..19d30d1
Fast-forward
 mymodule/important_classes.py | 2 ++
 1 file changed, 2 insertions(+)
```

This applies the commits to new_feature that are not on main to the end of main.

```
git status
On branch main
Your branch is ahead of 'origin/main' by 8 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
```

10.6. What if the content diverges

Let's make a new branch

```
git checkout -b hotfix
Switched to a new branch 'hotfix'
```

Then switch back to main

```
git checkout main
Switched to branch 'main'
```

and add more content to our main branch

```
echo "bold commit to main" >> mymodule/important_classes.py
echo "slow thoughtful fix" >> mymodule/important_classes.py
```

We can look at the version of the file we have here

```
cat mymodule/important_classes.py
```

we have the old content and our new two lines

```
test idea again
new feature
bold commit to main
slower more thoughtful fix
```

and commit.

```
git add .
git commit -m "working project"
[main b832b2a] working project
 1 file changed, 2 insertions(+)
```

Now we'll go back to the feature branch

```
git checkout -b hotfix
Switched to branch 'hotfix'
```

and look at what's there.

```
cat mymodule/important_classes.py
```

we don't have those two new lines.

```
test idea again
new feature
```

Now we fix the old bug that's not the new things. (maybe this was a different person working on this branch)

```
echo "fix older bug" >> mymodule/important_classes.py
```

and commit that

```
git add .
```

```
git commit -m 'fix other bug"
> '
[hotfix 3664aa0] fix other bug"
 1 file changed, 1 insertion(+)
```

Now we can compare the two versions.

```
cat mymodule/important_classes.py
```

we have 3 lines on the hotfix branch

```
test idea again
new feature
fix older bug
```

```
git checkout main
```

```
Switched to branch 'main'
Your branch is ahead of 'origin/main' by 9 commits.
  (use "git push" to publish your local commits)
```

```
cat mymodule/important_classes.py
```

and 4 on the main branch and the 3rd line is different on each.

```
test idea again
new feature
bold commit to main
slower more thoughtful fix
```

Now, we want to have all of those so we try to merge.

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

but we see an error

```
nano mymodule/important_classes.py
```

in nano now we see the file looks different

```
test idea again
new feature
>>>>> HEAD
bold commit to main
slower more thoughtful fix
=======
fix older bug
<<<<<<< hotfix
```

Git did not know how to merge the two files together, so it makred the part it did not know how to handle. There's the part from the HEAD (the branch we have checked out, in this case main)

to resolve the merge conflict, we edit the file to be what we want. In this casewe want both sets of changes. SO we edit to look as follows:

```
test idea again
new feature
bold commit to main
slower more thoughtful fix
fix older bug
```

Now git tells us that we need to fix the conflicts and commit ecause we have unmerged paths.

```
git status
On branch main
Your branch is ahead of 'origin/main' by 9 commits.
  (use "git push" to publish your local commits)

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

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

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

```
git commit -m "keep all changes"
U       mymodule/important_classes.py
error: Committing is not possible because you have unmerged files.
hint: Fix them up in the work tree, and then use 'git add/rm <file>'
hint: as appropriate to mark resolution and make a commit.
fatal: Exiting because of an unresolved conflict.
```

We have to add as usual

```
git add .
```

then we can commit.

```
git commit -m "keep all changes"
```

```
[main 62b943f] keep all changes
```

10.7. Prepare for next Class

1. Check that jupyter book is installed on your computer and the windows advice
2. Fix any open PRs you have that need to be rebased.
3. In your github in class repo, create a series of commits that tell as story of how you might have made a mistake and fixed it. Use git log and redirects to write that log to a file, `gitstory.md` in your KWL repo and then annotate your story to add in any narrative that didn't fit in the commit messages. This could be that you made a mistake in your code and used git to recover or that you got your git database to an undesirable state and got it back on track.

10.8. More Practice

1. Find an open source repository and look at the .gitignore file. In `donotcommit.md` reflect on what types of content typically get ignored and why you think they are ignored. If you can't figure out why, try to look it up.
2. Create a second git story for the other type form your first (code mistake or git mistake) in `gitstory2.md`
3. add something to the glossary, cheatsheet or a history sidebar to the notes.

KWL Chart

Working with this repo

When you are working on things that are not ready for feedback make a new branch to work on them.

If you are working on things that you want feedback on right away you can work directly on main.

When work is ready for feedback merge it into main and create a pull request from main into feedback.

Minimum Rows

> **Important**
>
> This is not currently complete but will contain a listing with links to recommendations

Added Files

file	content
README	the chart, (or toc)
workflows.md	[workflow]

Prepare for the next class

> **Warning**
>
> these are listed byt the date they were *posted* (eg the content here under Feb 1, was posted Feb 1, and shoudl be done before the Feb 3 class)
>
> *below* refers to followingin the notes

2022-01-25

- More practice with GitHub terminology. Accept this assignment, read through it, and follow the instructions at the end.
- Review these notes, bring any questions you have to class
- Read the syllabus, explore this whole website. Bring questions about the course. Be prepared for a scavenger hunt that asks you not to recall every fact about the course, but to know where to find informatio.
- Think about one thing you've learned really well (computing or not) and how do you know that you know it? (bring your example)

2022-01-27

1. Review these notes, both rendered as html and the raw markdown in the repository.
2. find 2-3 examples of things in programming you have got working, but did not really understand. this could be errors you fixed, or something you just know you're supposed to do, but not why
3. map out your computing knowledge and add it to your kwl chart repo. this can be an image that you upload or a text-based outline.
4. Make sure you have a working environment for next week. Use slack to ask for help.
    - check that you have Python installed with Jupyter, ideally with Anaconda
    - install jupyter book
    - install GitBash on windows (optional for others)
    - make sure you have Xcode on MacOS
    - install the GitHub CLI on all OSs

2022-02-01

1. Complete the classmate issue in your in-class repository.
2. read the notes PR, add or comment on a tip, resource, a bit of history in a sidebar or additional end of class question
3. try using git in your IDE of choice, log any challenges you have on the practice repo (`github-in-class-username`), and tag @sp22instructors on GitHub. You can use either repo we have made in class, or one for an assignment in another course.
4. using your terminal, download your KWL repo and update your 'learned' column on a new branch
5. answer the questions below in a new markdown file, `gitoffline.md` in your KWL on your new branch and push the changes to GitHub
6. Create a PR from your new branch to main **do not merge this until instructed**
7. add your programming challenge(s) you have had as issues to our private repo or to the course website repo if you like. Put one 'challenge/question' per issue so that we can close them as addressed. See last class notes for prompt.
8. Create or comment on a discussion thread in the private repo about the part of CS/ type of programming you like best/what you want to do post graduation.

2022-02-03

1. Review the notes
2. Reorganize a folder on your computer (good candidate may be desktop or downloads folder), using only a terminal to make new directories, move files, check what's inside them, etc. Answer reflection questions (will be in notes) in a new file, `terminal.md` in your kwl repo.
3. Add a glossary to the site to define a term or cheatsheet entry to describe a command that we have used so far.
4. Examine a large project you have done or by finding an open source project on GitHub. Answer the reflection questions in `software.md` in your kwl repo. (will be in notes)

2022-02-08

1. install h/w simulator
2. Add a glossary, cheatsheet entry, or historical context/facts about the things we have learned to the site.

3. Review past classes prep/more practice and catchup if appropriate
4. Map out how you think about data moving through a small program using the levels of abstraction. Add this to a markdown table in your KWL chart repo called `abstraction.md`. If you prefer a different format than a table, that is okay, but put it in your KWL repo. It is okay if you are not sure, the goal is to think through this.

2022-02-10

1. Read these notes and practice with the hardware simulator, try to understand its assembly and walk through what other steps happen. Make notes on what you want to remember most or had the most trouble with in `hardwaresurvey.md`
2. Review and update your listing of how data moves through a program in your `abstraction.md`. Answer reflection questions below.
3. Review the commit history and git blame of a repo in browser- what must be in the .git directory for GitHub to render all of that information? (be prepared to discuss this in class)
4. Fill in the Know and Want to know columns for the new KWL chart rows below.
5. Begin your grading contract, bring questions to class Tuesday.

2022-02-15

1. review the notes and ensure that you have a new, empty repository named test with its branch renamed to main from master.
2. Add the following to your kwl:

```
|git workflows | _ | _ | _ |
| git branches | _ | _ | _ |
| bash redirects | _ | _ | _ |
```

3. Practice with git log and redirects to write the commit history for your kwl chart to a file gitlog.txt and commit that file to your repo.

2022-02-17

1. Review the notes
2. For the core "Porcelain" git commands we have used (add, commit), make a table of which git plumbing commands they use in `gitplumbing.md` in your KWL repo
3. In a `gitunderstanding.md` list 3-5 items from the following categories (1) things you have had trouble with in git in the past and how they relate to your new understandign (b) things that your understanding has changed based on today's class (c) things about git you still have questions about
4. Make notes on *how* you use IDEs for the next week or so using the template `idethoughts.md` file in the course notes.

2022-02-22

1. review the past two classes of notes
2. find 3 more examples of using other number systems, list them in `numbers.md`
3. Read about hexpeak from Wikipedia for an overview and one additional source. In your kwl repo in `hexspeak.md` summarize it in your own words, one interesting fact from your additional source and link to the source you found. Come up with a word or two on your own.
4. (priority) Bring to class a scenario where you think git could help, but we haven't covered yet how to do it. (be prepared to post it to Prismia at the start of class)

2022-02-24

1. Check that jupyter book is installed on your computer and the windows advice
2. Fix any open PRs you have that need to be rebased.
3. In your github in class repo, create a series of commits that tell as story of how you might have made a mistake and fixed it. Use git log and redirects to write that log to a file, `gitstory.md` in your KWL repo and then annotate your story to add in any narrative that didn't fit in the commit

messages. This could be that you made a mistake in your code and used git to recover or that you got your git database to an undesirable state and got it back on track.

More Practice

> **Note**
>
> these are listed byt the date they were *posted*

2022-01-27

- (optional) try mapping out using mermaid syntax, we'll be using other tools that will faciltate rendering later, or try getting it to render on your own.
- (optional) read chapter 1 the programmmer's brain. Some of the ideas we talked about today are mentioned there, and it relates to where you're suppoed to be looking for things that you have done, but didn't really understand.
- try adding something to this page or the glossary of the course site or link a glossary term to an occurence of it on the site.

2022-02-01

1. Find the "Try it yourself" boxes in these notes, try them and add notes/ responses under a `## More Practice` heading in your `gitoffline.md` file of your KWL repo.
2. Download the course site repo via terminal.
3. Explore the difference between `git add` and `git commit` try committing and pushing without adding, then add and push without committing. Describe what happens in each case in your `gitoffline.md`

2022-02-03

1. Try to do as many things as possible on the terminal for a whole week.
2. Make yourself a bash cheatsheet (and/or contribute to one on the course site)
3. Read through part 1 of the programmer's brain and try the exercises, especially in chapter 4.

2022-02-08

1. Once your PRs in your KWL are merged so that main and feedback match, pull to updates your local copy. In a new terminal window, navigate there and then move the your KWL chart to a file called `chart.md`. Create a new README files with a list of all the files in your repo. Use `history N` (N is the number of past commands that history will return) and redirects to write the steps you took to `reorg.md`. Review that file to make sure it doesn't have extra steps in it and remove any if needed using nano then commit that file to your repo.
2. find a place where there is a comment in the course notes indicating content to add and submit a PR adding that content. This could be today's notes or a past day's.
3. Add a new file to your KWL repo called `stdinouterr.md` Try the following one at a time in your terminal and describe what happens and explain why or list questions for each in the file. What tips/reminders would you give a new user (or yourself) about using redirects and `echo`?
    - `echo "hello world" > fa > fb`
    - `echo "a test" > fc fd`
    - `> fe echo "hi there"`
    - `echo "hello " > ff world`
    - `<ff echo hello`
    - `fa < echo hello there`
    - `cat`

2022-02-10

1. Complete the `Try it Yourself` blocks above in your `hardwaresurvey.md`.
2. Expand on your update to `abstraction.md`: Can you reconcile different ways you have seen memory before?

2022-02-15

1. ¸ Read about different workflows in git and add responses to the below in a `workflows.md` in your kwl repo. Git Book atlassian Docs
2. Contribute either a glossary term, cheatsheet item, additional resource/reference, or history sidebar to the course website.

```
## Workflow Reflection

1. What advantages might it provide that git can be used with different workflows?
1. Which workflow do you think you would like to work with best and why?
1. Describe a scenario that might make it better for the whole team to use a workflow
other than the one you prefer.
```

2022-02-17

1. Add to your `gitplumbing.md` file explanations of the main git operations we have seen (add, commit, push) in your own words in a way that will either help you remember or behow you would explain it to someone else at a high level. This might be analogies or explanations using other progrmaming concepts or concepts from a hobby. Add this under a subheading `##` with a descriptive title (for example "Git In terms of ")
2. For one thing your understanding changed or an open question you, look up or experiment to find the answer

2022-02-22

1. Read more about git's change from SHA-1 to SHA-256 and reflect on what you learned (questions provided)
2. (priority) In a language of your choice or pseudocode, write a short program to convert, without using libraries, between all pairs of (binary, decimal, hexidecimal) in `numbers.md`. Test your code, but include in the markdown file enclosed in three backticks so that it is a "code block" write the name of the language after the ticks like:

```
```python
# python code
```
```

2022-02-24

1. Find an open source repository and look at the .gitignore file. In `donotcommit.md` reflect on what types of content typically get ignored and why you think they are ignored. If you can't figure out why, try to look it up.
2. Create a second git story for the other type form your first (code mistake or git mistake) in `gitstory2.md`
3. add something to the glossary, cheatsheet or a history sidebar to the notes.

Syllabus and Grading FAQ

How much does activity x weigh in my grade?

Can I submit this assignment late if …?

I don’t understand the feedback on this assignment

What should a Deeper exploration 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 Feedback Pull Request before my assignment was graded

For an Assignment, should we make a new branch for every assignment or do everything in one branch?

```
Doing each new assignment in its own branch is best practice. In a typical software development flow once the codebase is stable a new branch would be created for each new feature or patch. This analogy should help you build intuition for this GitHub flow and using branches.  Also, pull requests are the best way for us to give you feedback. Also, if you create a branch when you do not need it, you can easily merge them after you are done, but it is hard to isolate things onto a branch if it's on main already.
```

Glossary

> **Tip**
>
> We will build a glossary as the semester goes on. When you encounter a term you do not know, create an issue to ask for help, or contribute a PR after you find the answer.

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

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.

pull (changes from a repository)
: download changes from a remote repository and update the local repository with these changes.

repository
: a project folder with tracking information in it in the form of a .git file

shell
: a command line interface; allows for access to an operating system

terminal
: a program that makes shell visible for us and allows for interactions with it

Language Specific References

Python

- Python

Cheatsheet

Patterns and examples of how to accomplish frequent tasks. We will build up this section together over the course of the semester.

Basic Bash file operations

Move one folder up:

```
cd ..
```

Move at the top of your directory:

```
cd
```

Move to the specified directory:

```
cd directory
```

Create a file:

```
touch file_name.ext
```

Text editor:

```
nano file_name.ext
```

Display content:

```
cat file_name.ext
```

Create a new folder:

```
mkdir new_folder_name
```

Move a file:

```
mv file_name.ext folder_name_file_is_going_moved_to
```

Move multiple files:

```
mv file_name1.ext file_name2.ext file_name3.ext folder_name_files_are_going_moved_to
```

List files in the current directory:

```
ls
ls -hl  //displays rw-r-r
ls -G - //folders are colorized
ls -hlG //displays rw-r-r and folders are colorized
```

Show your currect directory:

```
pwd
```

Remove a file:

```
rm file_name.ext
```

Copy a file:

```
cp file_name.ext copied_file_name.ext
```

> **Note**
>
> 1. In every command you can add your directory/ location (ex. docs/file_name.ext).
> 2. "." dot symbolizes our current location and "…" two dots, one level up in the directory tree.
> 3. "*" represents any number of unknown characters; creates a pattern (ex. rm pyt*.py removes all files that start on 'pyt' and end with extension py).

Delete an empty directory:

```
rmdir directory
```

Since you can't delete a directory with files in it you need to recursively delete the folder and its contents. The `-R` is a recursive declaration which tells the terminal to delete the folder, the files within the folder, subfolders, files in the subfolder etc. Source

Delete everything in a directory without confirmation:

```
rm -R directory
```

The `-i` is a flag that prompts you if you want to remove each separate file in the directory.

Delete everything in a directory with confirmation:

```
rm -iR directory
```

List the contents of the directory:

```
ls
```

Wildcard / Kleene star

The wildcard character in bash `*` works by expanding in place to separate arguments that match whatever pattern you're writing.

Example, given a directory `stuff/`:

```
stuff/
  a.py
  b.py
  c.py
  other.txt
  another.md
  nested_folder/
```

If we were to run `ls *.py` while in the `stuff/` directory, the command actually run by the computer is `ls a.py b.py c.py`. This also works for commands like `mv`. I.e. `mv *.py nested_folder/` runs `mv a.py b.py c.py nested_folder/`

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.

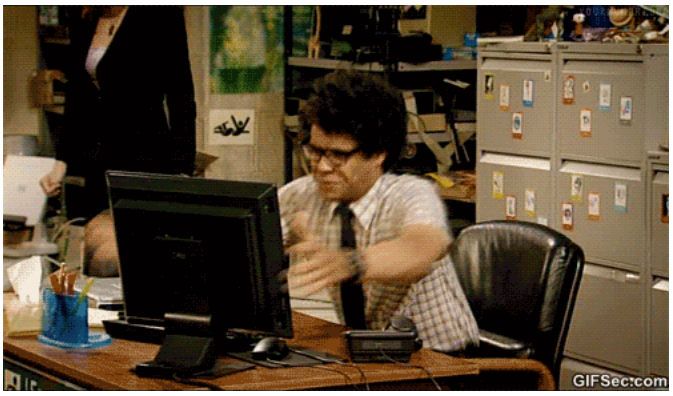
on email

- 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.

I hope that with this advice, you never feel like this while working on assignments for this class.



Why this way?

A new book that might be of interest if you find programming classes hard is the Programmers Brain As of 2021-09-07, it is available for free by clicking on chapters at that linked table of contents section.

Learning requires iterative practice. It does not require memorizing all of the specific commands, but instead learning the basic patterns.
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. 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.

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. We will collaboratively annotate notes for this course. Dr. Brown will post a basic outline of what was covered in class and we will all fill in explanations, tips, and challenge questions. Responsibility for the main annotation will rotate.

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.

Getting Help with Programming

This class will help you get better at reading errors and understanding what they might be trying to tell you. In addition here are some more general resources.

Asking Questions

One of my favorite resources that describes how to ask good questions is this blog post by Julia Evans, a developer who writes comics about the things she learns in the course of her work and publisher of wizard zines.

Describing what you have so far

> **Note**
>
> A fun version of this is rubber duck debugging

Stackoverflow is a common place for programmers to post and answer questions.
As such, they have written a good guide on creating a minimal, reproducible example.

Creating a minimal reproducible example may even help you debug your own code, but if it does not, it will definitely make it easier for another person to understand what you have, what your goal is, and what's working.

Getting Organized for class

The only **required** things are in the Tools section of the syllabus, but this organizational structure will help keep you on top of what is going on.

Your username will be appended to the end of of the repository name for each of your assignments in class.

File structure

I recommend the following organization structure for the course:

```
CSC392
  |- notes
  |- kwl-char-username
  |- spring2022
  |- ...
```

This is one top level folder will all materials in it. A folder inside that for in class notes, and one folder per repository that you work on.

Finding repositories on github

Each assignment repository will be created on GitHub with the `introcompsys` organization as the owner, not your personal acount. Since your account is not the owner, they do not show on your profile.

Your assignment repositories are all private during the semester. At the end, you may take ownership of your portfolio[^pttrans] if you would like.

If you go to the main page of the organization you can search by your username (or the first few characters of it) and see only your repositories.

> **Warning**
>
> Don't try to work on a repository that does not end in your username; those are the template repositories for the course and you don't have edit permission on them.

By Professor Sarah M Brown