So, You Want to Use Git?

A Starter's Guide to Git Including:
GitHub
GitLab
GitKraken
The Terminal / Command Line

Document Version: 1.3.1
Author: Mark C. Anderson
Institution: Brigham Young University

So, You Want to Use Git?

A Starter's Guide to Git Including:

GitHub
GitLab
GitKraken
The Terminal / Command Line

Notes:
- The photograph on the front cover comes from https://www.nasa.gov/image-feature/the-international-space-station-as-of-oct-4-2018-2 and appears to be in compliance with the National Aeronautics and Space Administration (NASA) copyright terms as found online at https://www.nasa.gov/multimedia/guidelines/index.html. Use of this photograph does not imply the sponsorship or approval of NASA for this work.
- The official Git logo that was used on the front page was taken directly from https://git-scm.com/downloads/logos, was created by Jason Long, and is licensed under the Creative Commons Attribution 3.0 Unported License.
- This tutorial is in no way an official material of Git, GitHub, GitLab, GitKraken, Brigham Young University (BYU), or any other organization mentioned in the document. It was created by a student at BYU while in a paid research assistant capacity and screenshots include only material that is freely available to Git users. This tutorial is intended to be distributed freely, with version updates available online at https://github.com/AerospaceMark/So-You-Want-To-Use-Git. For problems or concerns, please contact the author directly at anderson.mark.az@gmail.com. If this runs into BYU intellectual property concerns, I (Mark Anderson), am happy to discuss them. I have never received extra money (and never intend to) beyond my wages as a student researcher for this work and am happy to share it freely.

Anderson, M. C.

- The unofficial BYU Acoustics logo shown on the previous page was created by Logan T. Mathews, a fellow student researcher at BYU.

Why should I read this tutorial?

Git is a useful tool designed to help you write better code, work with files, and collaborate with others. It does this by keeping track of changes that you make to your files and allowing you to easily access old versions. Git also enables you to work with others on a common project by letting everyone access the files, make improvements, and then merge all the improvements back into the original project. Git is used by individuals, large corporations, and everyone in between.

However, Git has a notoriously steep learning curve. If you've tried learning Git in the past, you may have been faced with a mess of command line text, never quite understanding what was happening. It's hard to keep trudging through that, and that's where this tutorial comes in handy. This tutorial aims to teach the reader how to use Git by first explaining what Git is and how it will benefit them, and then demonstrating click by click how to interface with the common online Git providers GitHub and GitLab, all without opening the terminal.

After the reader is comfortable using an online Git provider, the subsequent sections then demonstrate how to use Git on a Desktop through GitKraken and then finally the terminal. These are elaborately described with screenshots showing every step. The section that uses the terminal shows every line that must be used and explains the inputs and outputs in plain terms.

You may have noticed how long this tutorial appears to be. However, this tutorial is designed to be cherry-picked to meet your personal Git goals. If you just want a place to store files and track how they change over time, feel free to read the few pages that describe either GitHub or GitLab and then put the tutorial down. If you want an in-depth experience on how to work in teams and do it all from the terminal, feel free to read just the sections about the terminal, and then go to the Further Learning section listed at the end to expand your knowledge. If you want something in between, or are just exploring, feel free to read any sections you want. For example, if you're only interested in GitHub, then you'll only need to read a few sections of this tutorial, of which a large amount is devoted to screenshots helping you along the way.

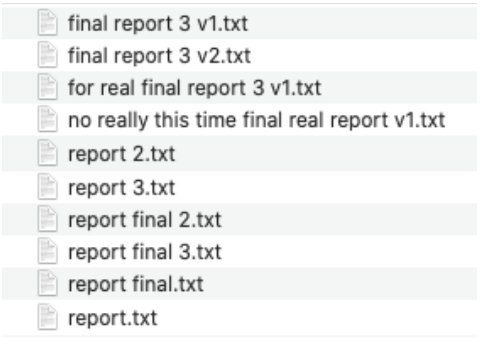
For questions regarding this tutorial or if you have any recommendations for how to make it even better, feel free to email Mark Anderson at anderson.mark.az@gmail.com. Success stories are also greatly appreciated and encouraged. If updates occur in this tutorial, they will be available to everyone for free at https://github.com/AerospaceMark/So-You-Want-To-Use-Git.

Date of last revision: March 26, 2022

1 What is Git?

1.1 Purpose

You know that feeling you get when you open a folder and see something like this?



Or when you try to change something in a file, decide it was a bad idea, but then can't get it back to the way it was before you made those changes? Git is the solution to helping you manage all of the versions of your files. In fact, Git is called a "Version Control System", or VCS for short. There are other similar tools, but Git is the most widely adopted VCS all around the world for both small and large groups to handle their files.

Git enables you to collaborate with others on your files. Imagine that one of your teammates creates a file and gives everyone access to it. You notice an improvement that you can make and so you go ahead and make it. Rather than mass-emailing out a new version of the file, Git lets you merge your new version with the original version, and everyone can automatically access your improvements. These practices can be used as a simple VCS for personal use or can be scaled up to large companies with potentially thousands of employees.[1]

1.2 Git Workflow

Before we get into examples, let's discuss some of the terminology and workflow that we'll be using throughout this tutorial.

1.2.1 Repositories

A group of files stored on Git is called a **repository**. A repository can be stored on your computer, online, or both. Git stores changes that are made to each file over time and lets you see that file's history. A single repository can be shared between many people, with each person adding to it.

1.2.2 Staging

Sometimes you are ready to save new versions of some of your files to Git, but not all of them. The **staging area** is a place that you can store files that you are ready to save (or commit) to Git.

1.2.3 Committing

Think of the phrase "commit to memory", which means that you memorize something and keep it in your brain for future use. That is exactly what a Git **commit** does. A Git commit takes all of the files that are in the staging area and "commits" them to Git, where they will forever be stored

[1] Google, Facebook, Microsoft, LinkedIn, and Linux to name a few examples (https://git-scm.com)

and can be accessed at any time. Each commit is a snapshot of the current state of your files, and Git lets you see the differences between your files over time from commit to commit.

1.2.4 Branching

You always want to have a current functioning version of your files. If your files contain computer code, then this means that you have a version of your files that you can use at any time and have it work properly. This is difficult when you're trying to make improvements to your files because your changes may have unintended consequences. This difficulty is amplified when working in teams. To overcome this challenge, you create a **branch**, which makes a copy of the repository that you can work in, improve, mess up, and perfect until you're ready to bring those improvements to the functioning version of your files.

1.2.5 Merging

When you create a branch, improve your files, and then decide to bring those improvements back to the main branch (typically called the **master[2] branch**), you do a **merge** of your branch with the master branch. You may experience some conflicting changes between your improvements and someone else's improvements, and these can be settled by using a **merge tool** to help you see any conflicts and resolve them. Once you merge your changes to the master branch this becomes the new current functioning version of your files.

1.3 Ways to Access Git

In this tutorial we will cover four ways to access Git. There are many other ways to access Git, so if you find another product or method feel free to explore it. Below are descriptions of the four methods that we will cover.

1.3.1 GitHub

GitHub is an online location to store Git repositories. The web interface lets you use Git without needing to use the terminal and provides a user-friendly and simple Git experience. GitHub is the most popular online Git provider in the world.[3]

1.3.2 GitLab

GitLab is an online location for Git repositories, very similar to GitHub. The interface is different, but all of the same functionality is provided. The decision of whether you use GitHub or GitLab depends on your needs, and both provide more than enough for the purposes of this tutorial.[4,5]

1.3.3 GitKraken

GitKraken is a tool that you can install on your computer to manage both your online and local Git repositories. This provides a nice way to visualize the Git workflow and interact with your files, again without needing to use the terminal.

---

[2] During the year 2020, it became common for the primary branch to be given the name "main", rather than "master". This tutorial was made before that change fully developed. Because the screenshots and associated text would be difficult and time-consuming to adjust, and especially because the author is a full-time university student, please understand that if this tutorial were made today that the primary branch would be called "main" in accordance with what it now common practice. Thank you in advance for understanding.

[3] As of June 2020

[4] https://www.youtube.com/watch?v=s8DCpG1PeaU&feature=youtu.be

[5] The BYU physics department uses GitLab. To access, go to http://git.physics.byu.edu and sign in with your netID and password

1.3.4 The Terminal (or Command Line)

The terminal[6] is arguably the most powerful and efficient way to manage your Git repositories. The terminal provides a very flexible working environment and can be a fast and effective way to navigate files and commit your files to Git. This is where most online tutorials seem to start, and it understandably can look very intimidating. This is a normal feeling and as you learn more about the terminal and how to use Git, you will become more comfortable using it.

1.4 What Makes Git Different from a Shared Online Folder?

This is a good question. A shared online folder to keep track of your files as a group is an excellent choice and should definitely be used in many situations. Let's consider a motivating example that illustrates where Git might come into play: Suppose that you are working on a project that involves writing computer code. After several weeks you decide that you want to reproduce some results that you had several weeks earlier. If your files are stored in an online folder then oftentimes you can go back and restore old file versions, but which version (1-72…)[7] was the one that you used to produce those results? Eventually you will find it, but what if you then remember that your code depended on another file that has also since been modified. Which version (1-34…) of that file do you need to restore to get your original code working again? A potential solution to this is that whenever you produce results, you save copies of all those files into a separate directory. While this is a potentially useful solution, it is risky and it is possible that these files may get accidentally overwritten, deleted, or otherwise changed in the future.

This is where Git steps in. When you use Git, it will take a snapshot of all the files in the repository each time you make a commit. If you want to reproduce those results from a few weeks ago, just open Git and open those files as they appeared when you produced those results. Everything will be exactly as it was.

If you are collaborating with team members on a common set of files, Git enables you to work on them simultaneously and then merge them together later. For example, if you need to make changes to something at the top of the file and your teammate needs to make changes to something in the middle of the file, then you can both work on the file at the same time and merge the differences together when you're done. If there are any conflicting changes Git also lets you decide which version of the changes to keep.

A shared online folder is a great place to store things like data files. Git is a great tool for managing code or other text-based files because it lets you keep an excellent history of your files and enables easier and more efficient collaboration.

1.5 How This Document Works

This tutorial is designed to be cherry-picked based on your individual Git needs and goals. A quick start guide is found in Section 2 and teaches the foundations of Git using all four approaches (GitHub, GitLab, GitKraken, and the terminal/command line). Choose the sections that are most relevant to your goals and read those. After that, Section 3 is a guide for users that want to work in teams and discusses how to use branching and merging for each of the four methods. Then,

[6] The terminal and the command line are synonyms in this tutorial, and the go-to word that we will use is "terminal"

[7] Online folders tend to store file versions as simply "Version 1", "Version 2", etc.

Section 4 goes more in-depth on some advanced topics. If you are an absolute beginner (as we all are to begin with) then I personally recommend that you choose an online version (either GitHub or GitLab) to learn about and play with and then a local version (either GitKraken or the terminal) that you can use for your day-to-day work. As you gain experience and confidence you will find that your productivity increases when you use Git, even though you may feel like your productivity takes a hit for a short time as you are getting up to speed. The time that you spend now learning Git will more than make up for itself in the future.[8]

[8] Note that the screenshots presented in this tutorial reflect the user interfaces as they appeared during the summer months of the 2020. Although it's likely that the user interfaces will evolve over time, this tutorial should provide excellent guidance for many years to come.

2 Quick Start Guide for Personal Use

The purpose of this chapter is to guide you to the point where you can use any of the four methods (GitHub, GitLab, GitKraken, and the terminal) to manage your personal files in Git repositories. As we do this, you will see screenshots and text telling you exactly what to do to get everything working. When you complete this chapter, feel free to explore all of the extra buttons and features in the different tools and you will learn even more. To complete the sections on GitKraken and the terminal you will need either a GitHub or GitLab account because we will be discussing how to use GitKraken and the terminal to manage **remote repositories**.

For this part of the tutorial we are going to be using two files:

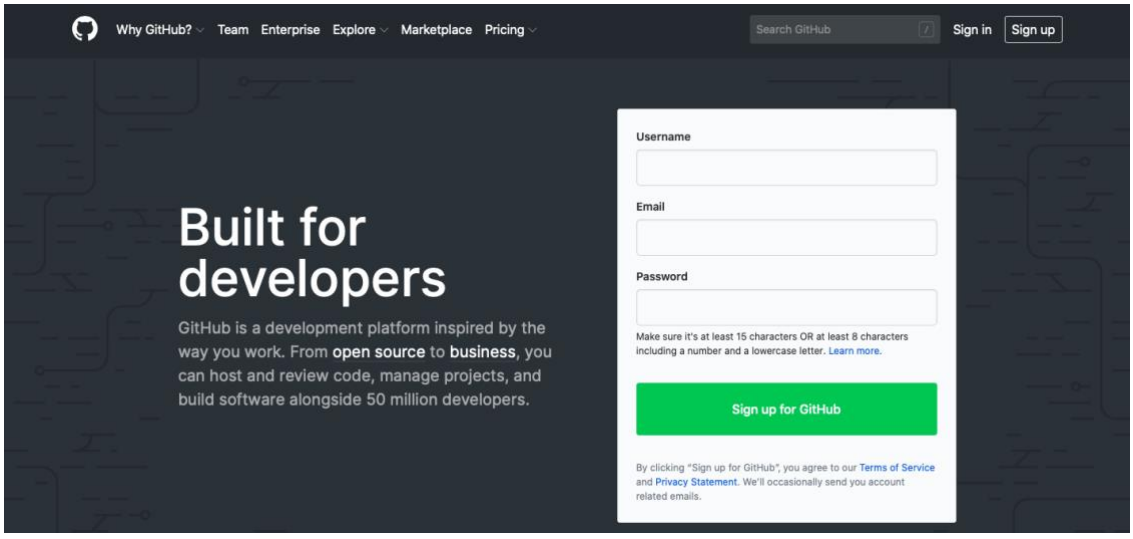
some_math.m
important-notes.txt

The some_math.m file will have some math written in the MATLAB programming language and important-notes.txt will have some plain text written in it. It will be easier to follow along if you create your own "practice" files too. Your files don't have to match the names of the files in this tutorial, and the actual content of the files doesn't matter too much, as long as you can go in and make changes to the files during the tutorial so we can practice using Git to track those changes.

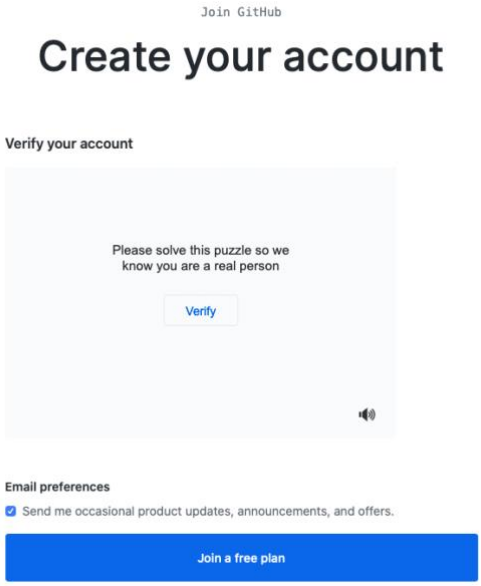
2.1 Online Providers

2.1.1 GitHub

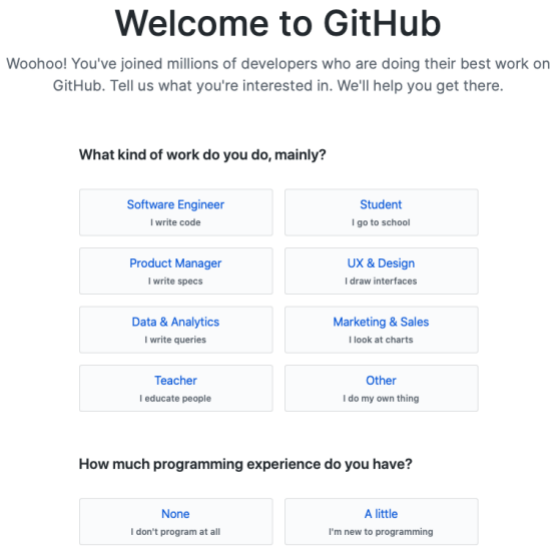
GitHub is currently the most commonly used online Git provider. They offer a simple interface and you can do many of the useful Git tasks completely online, without ever having to type code into your terminal. To get started, go to https://github.com and create an account. Type in your username and email, create a password, and then press the green "Sign up for GitHub" button.



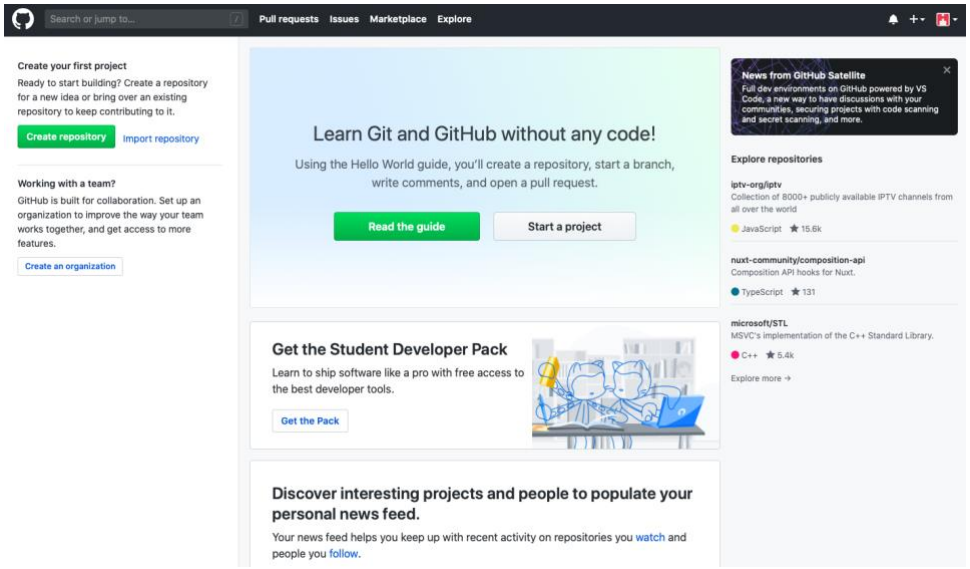
You may be asked to perform some task that verifies you are a human and not a robot, and then click the blue "Join a free plan" button.



You'll then be brought to a page with several questions about who you are and why you want to use GitHub. Go ahead and answer them appropriately and then click on the blue "Complete setup" button at the bottom of the screen.

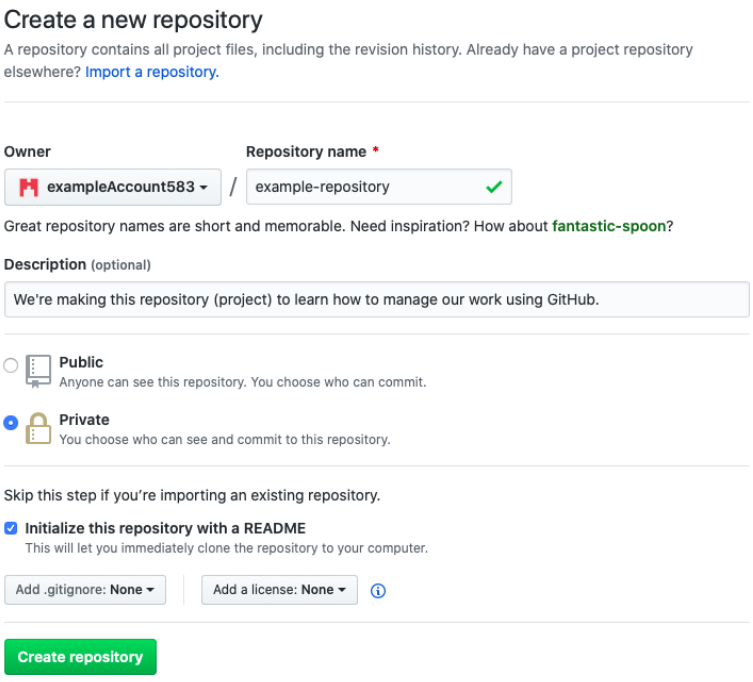


You'll then be greeted with the following window, which will likely be customized to your responses on the previous page, and your account has been successfully created!



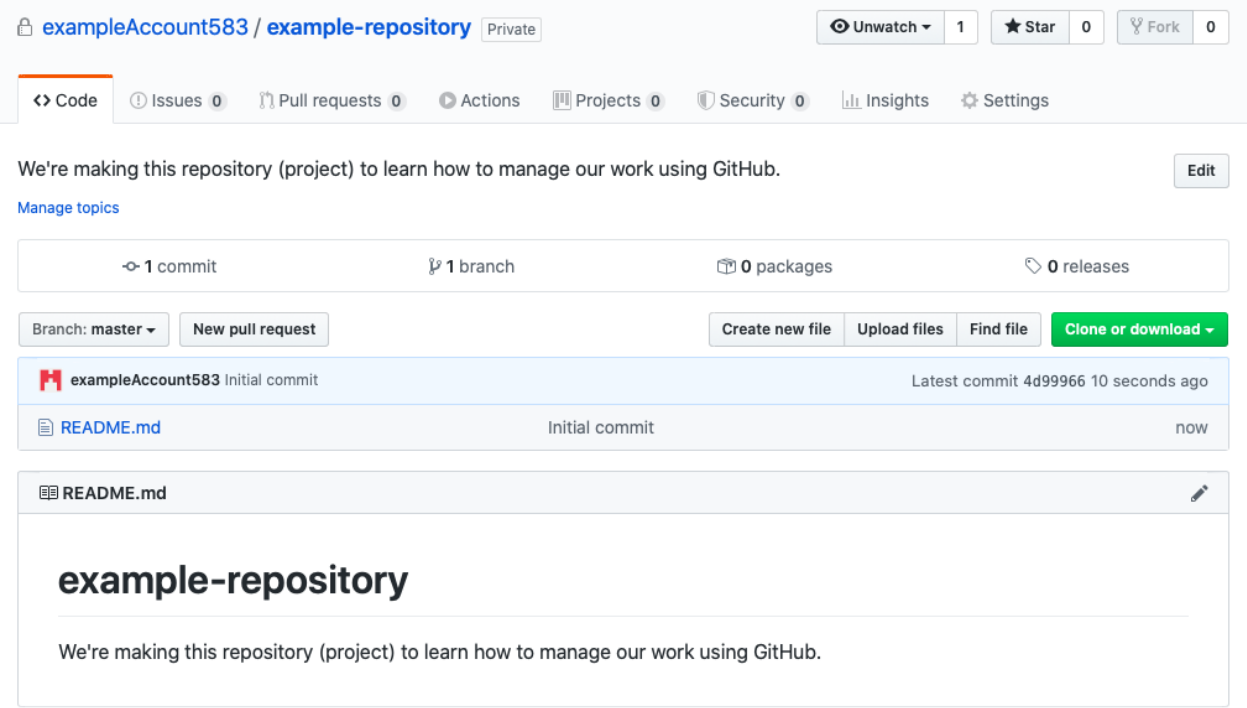
2.1.1.1 Creating a Repository

To create a repository, click on the green "Create repository" button on the left-hand side of the screen. If you're using an account that already has repositories, the button to make a new repository will look different but will still be located in the same general area of the web page next to a list of your repositories.



You'll be brought to the above page, where you enter some details about the repository you would like to make. In the "Repository name" field, enter a simple name that reflects the project you are making. Although this can be changed later down the road, try to choose a good name

now because changing the name can cause issues. In the “Description” field put a short description of what the project is about. Depending on your account permissions, you may not be able to choose between “Public” and “Private”, but if you can choose then you’ll most likely want to choose “Private” so that only you and people you allow can access the project. You will want to check the box that says, “Initialize this repository with a README”, which will take the description you wrote and display it with your repository so that anyone accessing your repository will immediately see a description of your project. For now, leave the “Add .gitignore” and “Add a license” boxes as they are by default. When finished, click the green “Create repository” button at the bottom of the page. You’ll be brought to the following new page:



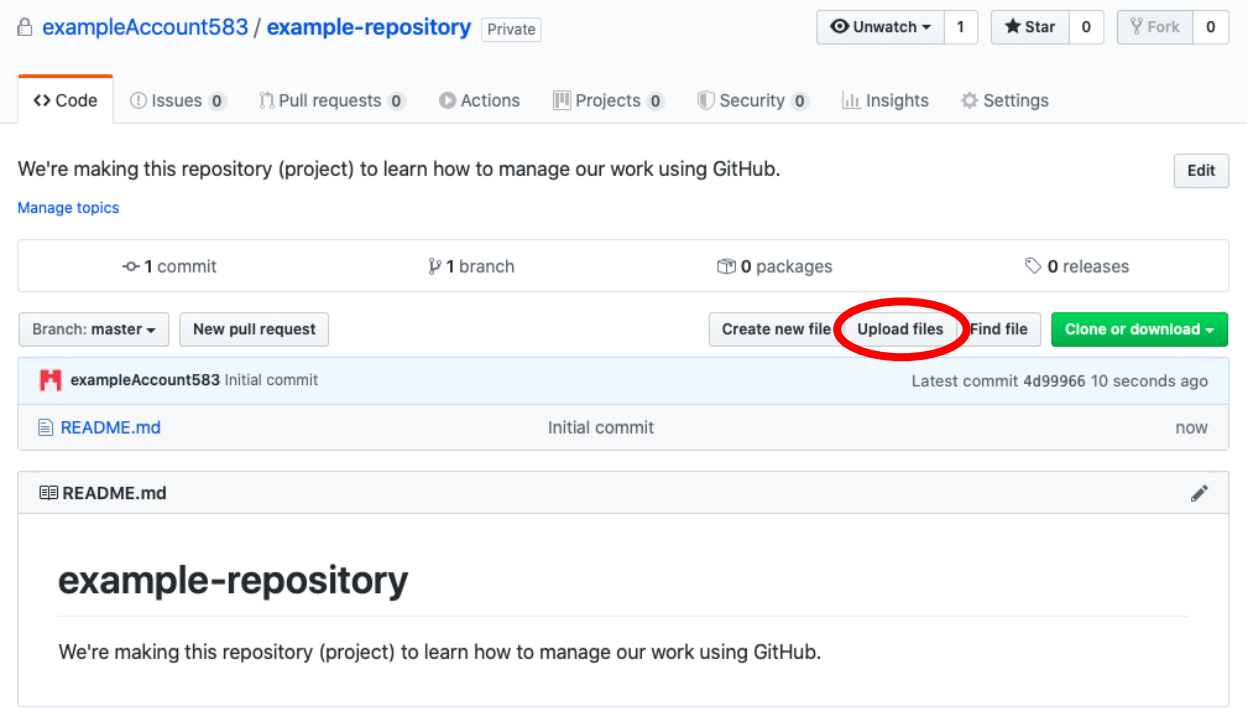
You have successfully created a Git repository using GitHub. You can upload files, see how your files have changed over time, and keep all of your work in a safe spot online. You may have some additional popups offering to help you learn about all the different options available to you on this page, explore them if you’d like or go ahead and dismiss them.

2.1.1.2 Uploading Files

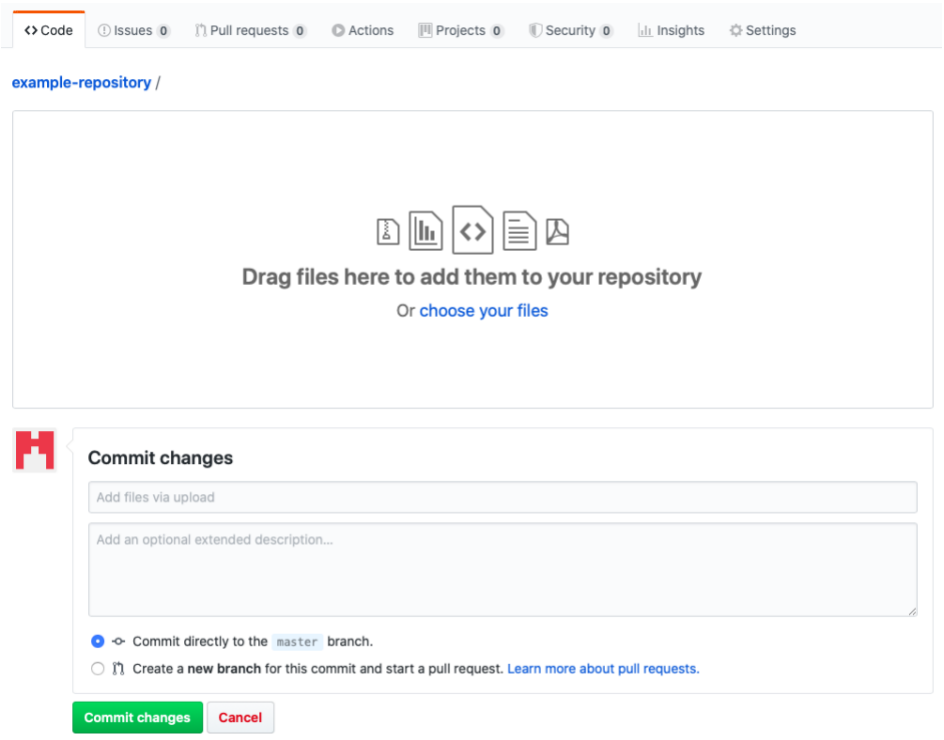
Let’s say that you have two files in your project that are saved on your computer:

some_math.m
important-notes.txt

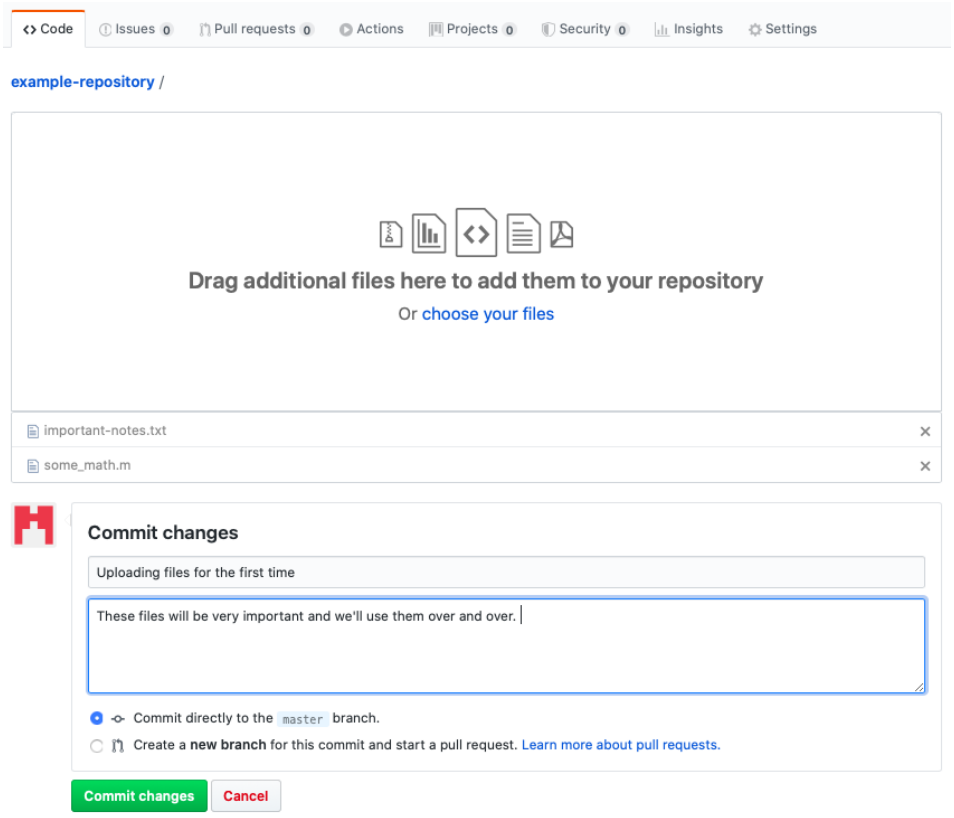
You would like to upload them to GitHub so that you can track all of the changes that you make to them over time and maybe someday collaborate on the files with another user. To upload these files, click on the “Upload files” button.



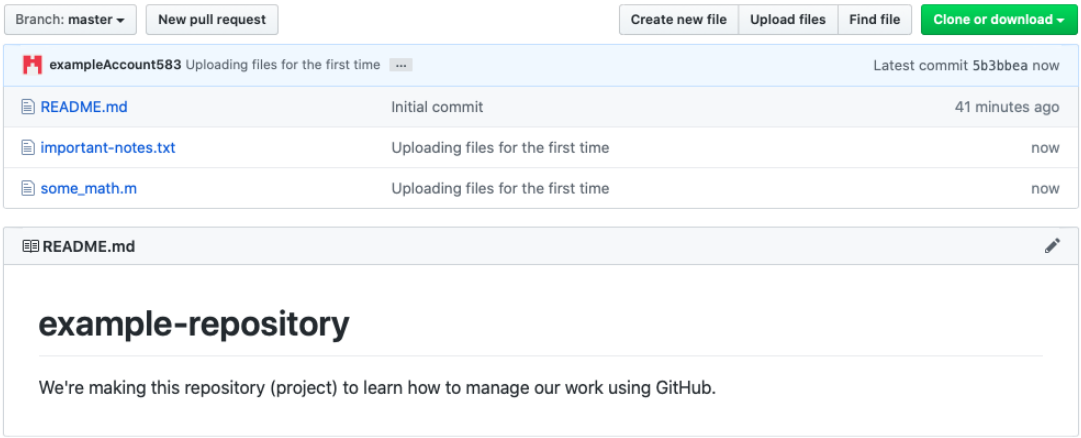
You will be brought to this page, which understandably may look rather frightening:



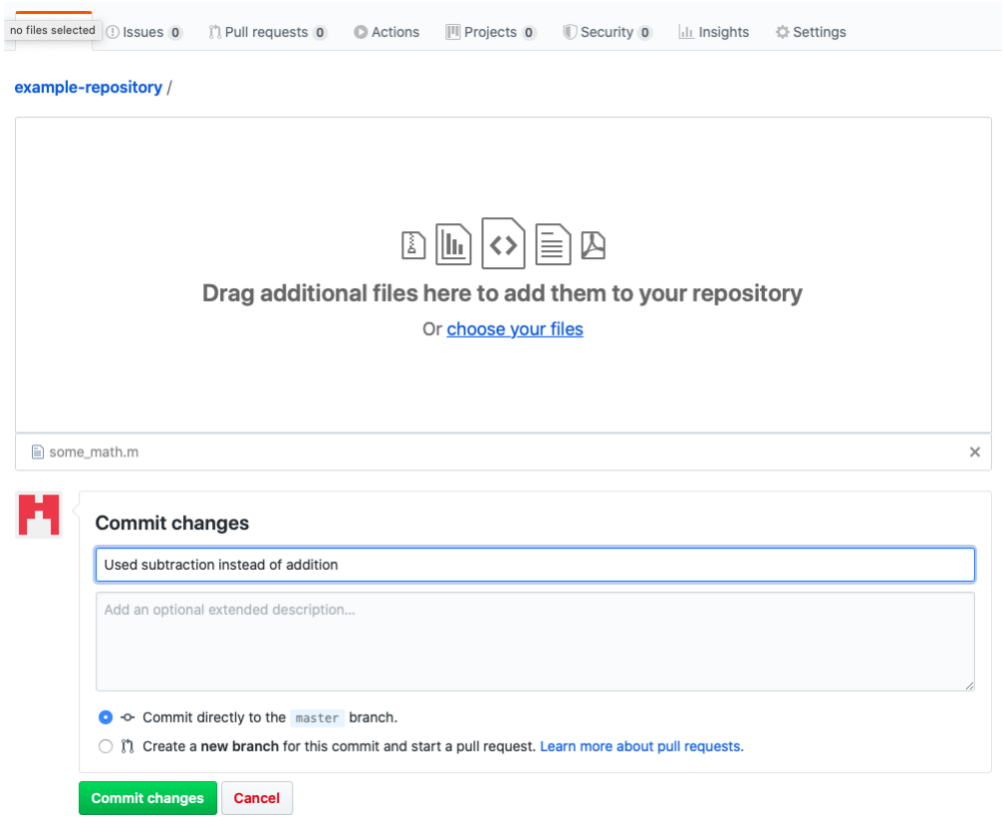
Select the blue “choose your files” text to add files. Then you will need to add a commit message describing the changes that you are making to the project. If your message is too long you will be prompted to put additional details in the larger text box so that as needed other people can see your message in full detail. Go ahead and leave other options as they are.



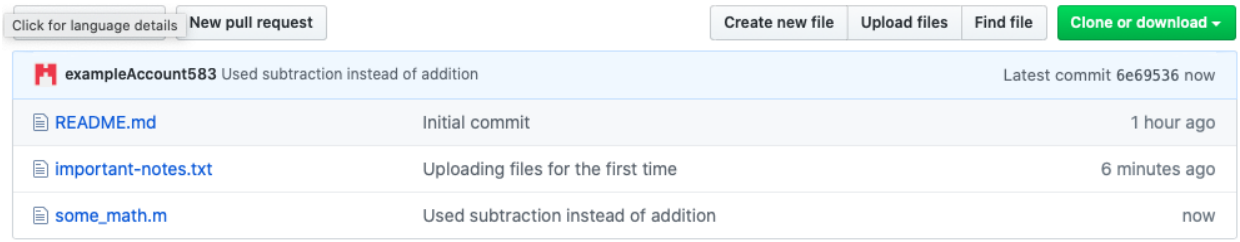
When you’re ready, click the green “Commit changes” button to save these to your repository. After this, your repository will look like this:



Now, let's say that you have made changes to some_math.m on your computer, and you would like to save those changes on GitHub too. Again, click on "Upload files" and upload the new version of the file from your computer, like so:



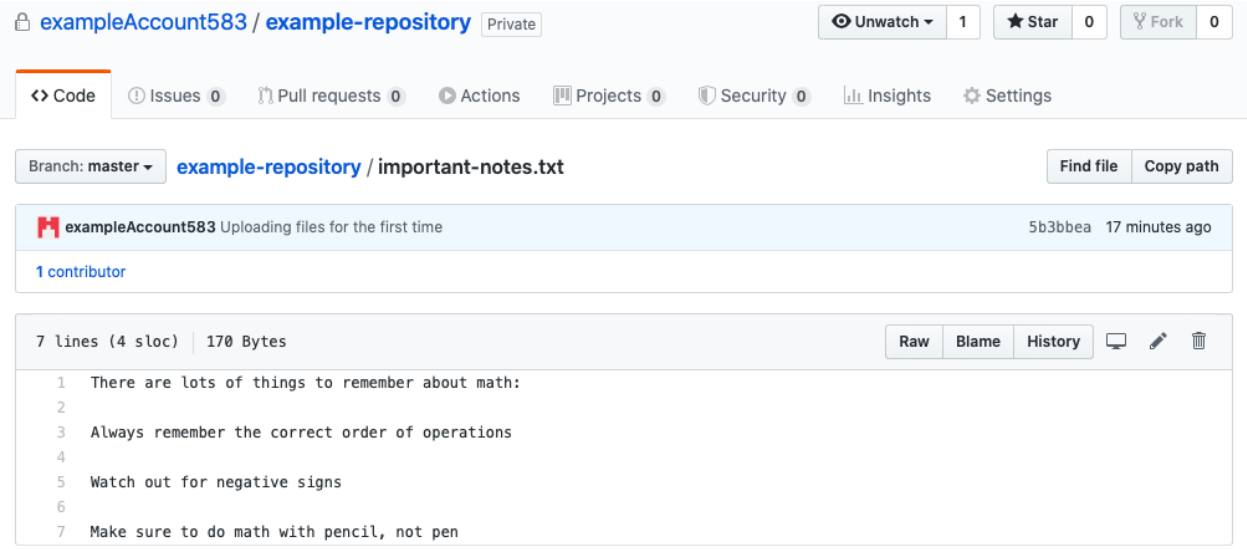
Go ahead and commit those changes using the green button and when you look at your repository you will see that the new file was uploaded. Notice that the most recent commit message for each file is shown along with how long ago it was committed.



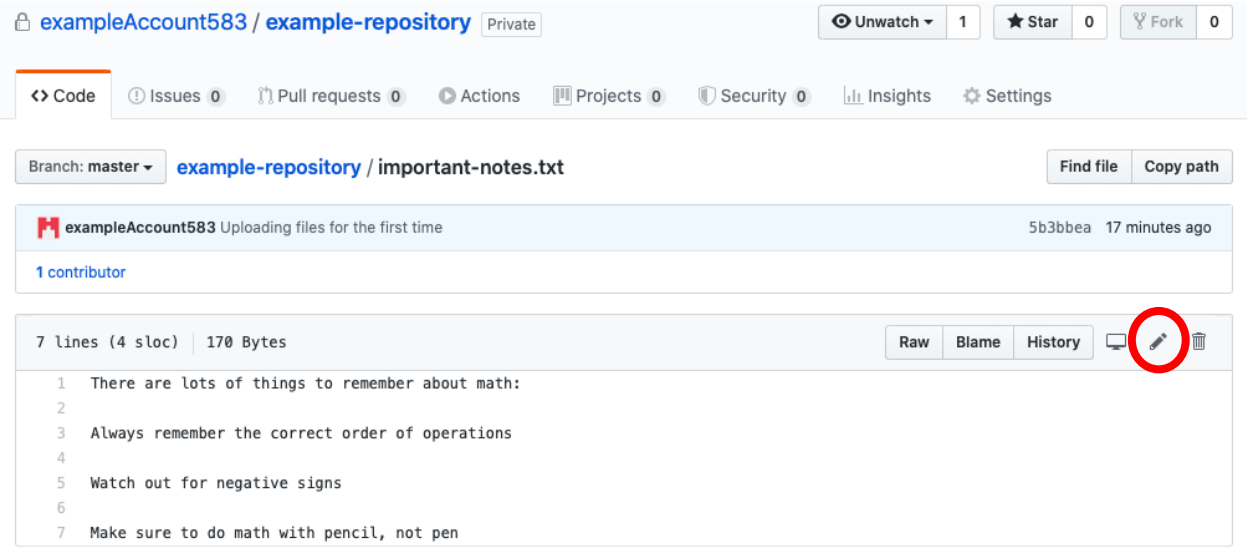
2.1.1.3 Editing Files Online

Although it is generally bad practice, you can actually edit files on GitHub's website. This is risky because it means that the files online are changed in ways that the files on your computer aren't. Better practice would be to make changes on your computer and then upload the new files, as shown in the previous section. Nevertheless, should you ever find yourself in an appropriate situation to edit files online, here's how to do it.

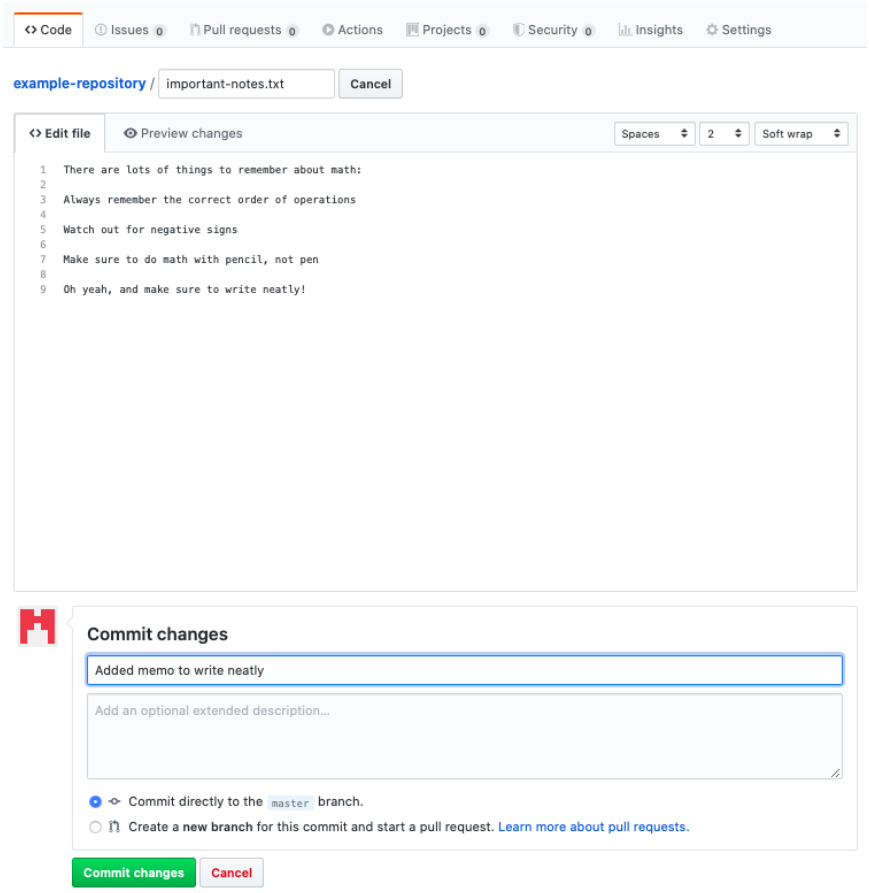
Click on one of your files to open it. For example, after clicking on the “important-notes.txt” file you’ve that uploaded you’ll see the contents of the file:



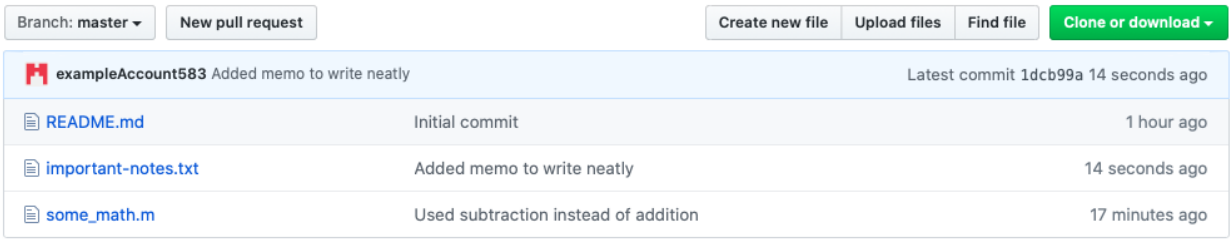
To edit the file, click on the pencil icon on the right-hand side of the page:



Make any desired changes, add a commit message, and then press the green “Commit changes” button.

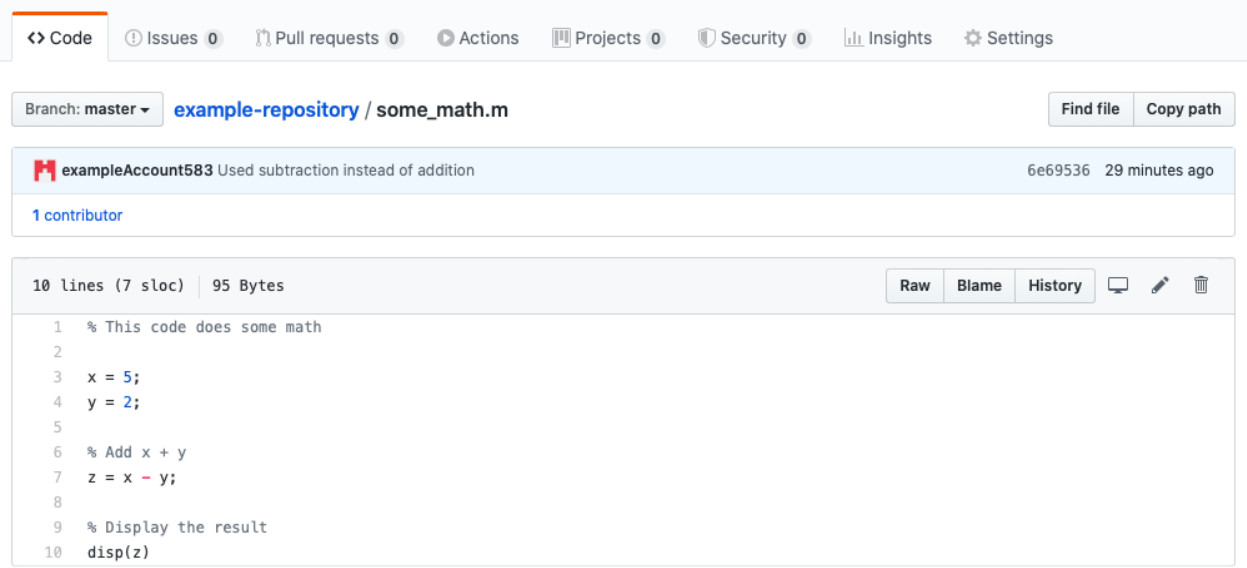


You will again see the changes you've made when looking at your repository:

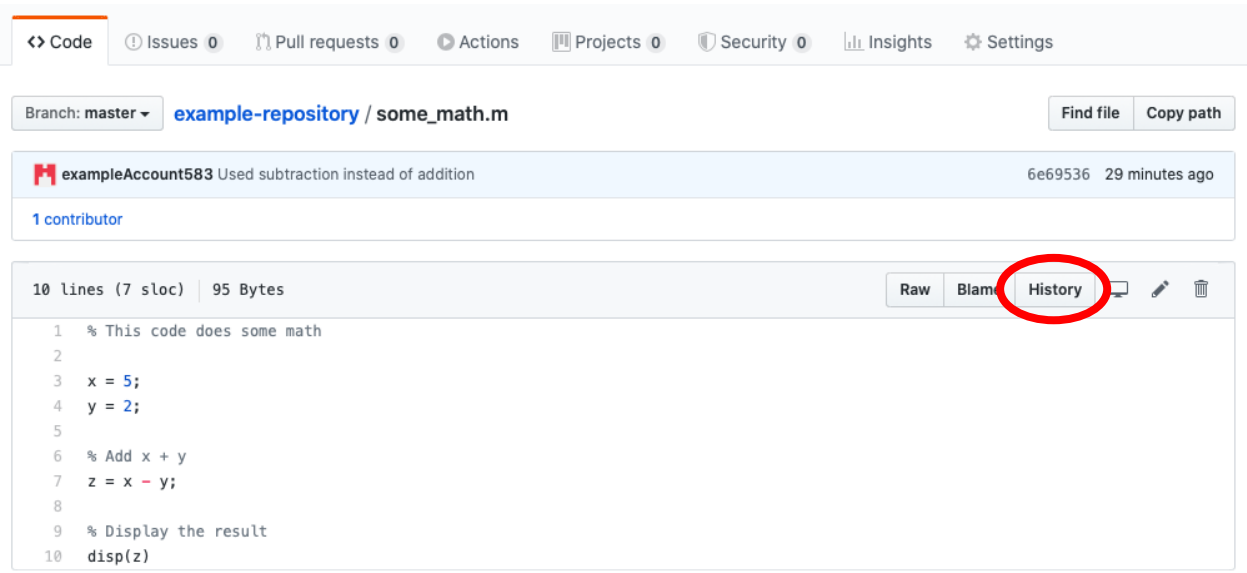


2.1.1.4 File History and Comparing File Versions

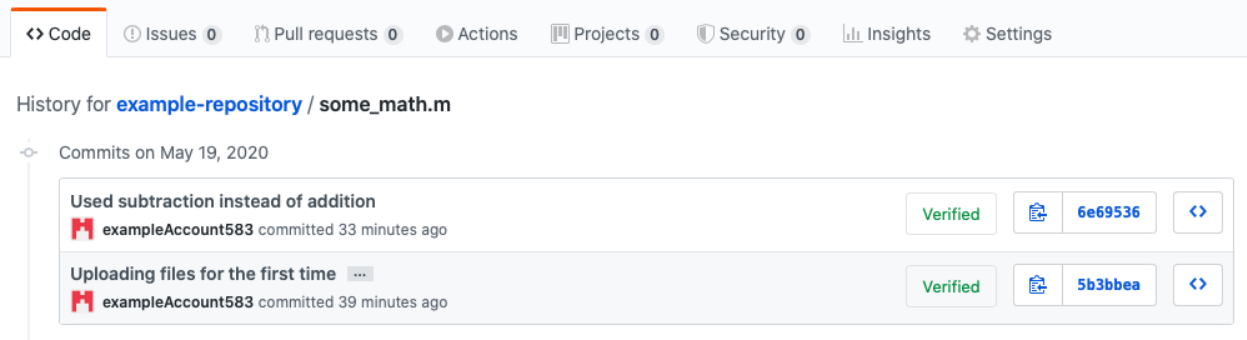
One of the most important Git benefits is the ability to look at and even compare old versions of your files. If you want to look at the history of a file, click on it in your repository. Doing this for "some_math.m" leads to the following page:



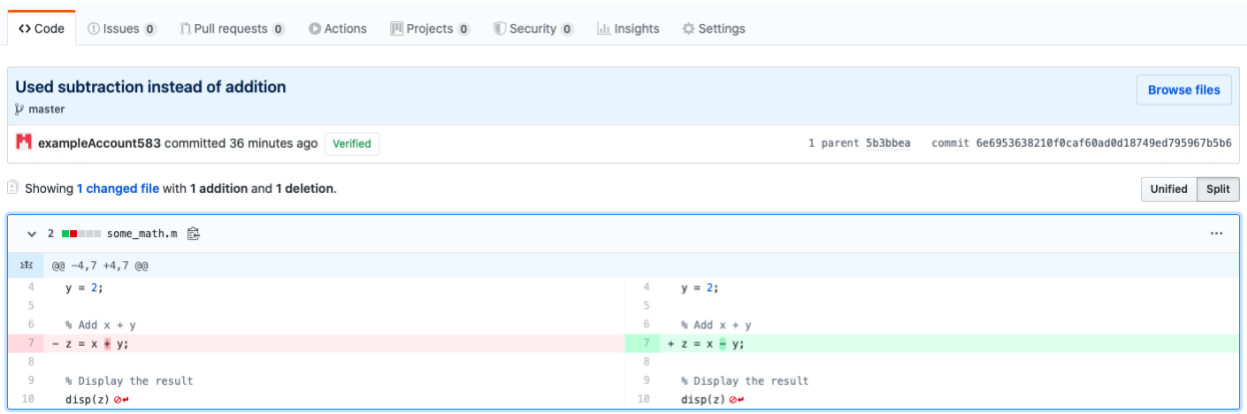
If you want to see the file history, click on the “History” button:



This will bring you to a page where you can see the history of this particular file. We see here that this file has changed twice (being created technically counts as a change).

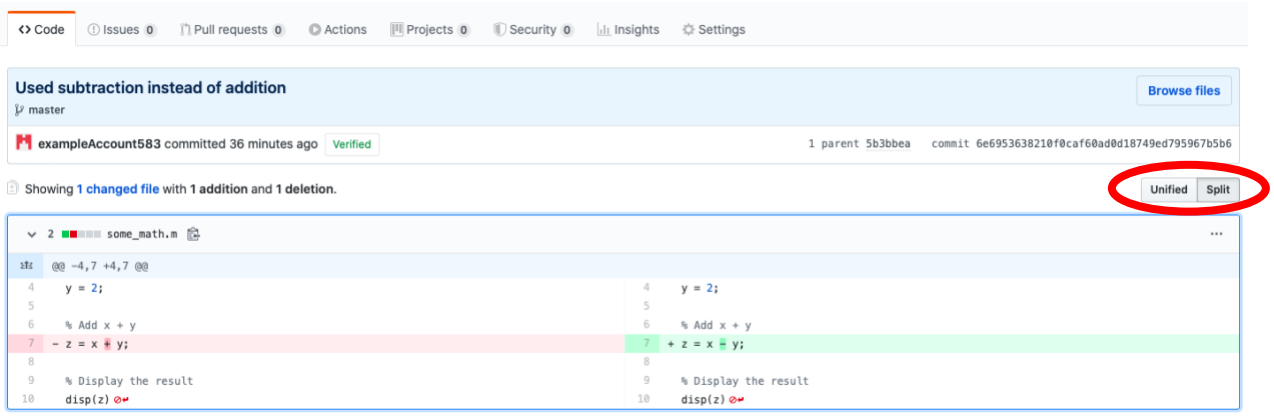


Click on one of the file versions to compare it with the version that came before it. This will bring you to a page that looks something like this:



The version of the file on the left-hand side of the page is the older version and the version on the right-hand side of the page is the newer version. You can see here that in this example the only change between the two files was turning the addition symbol into a subtraction symbol, as described in the commit message.

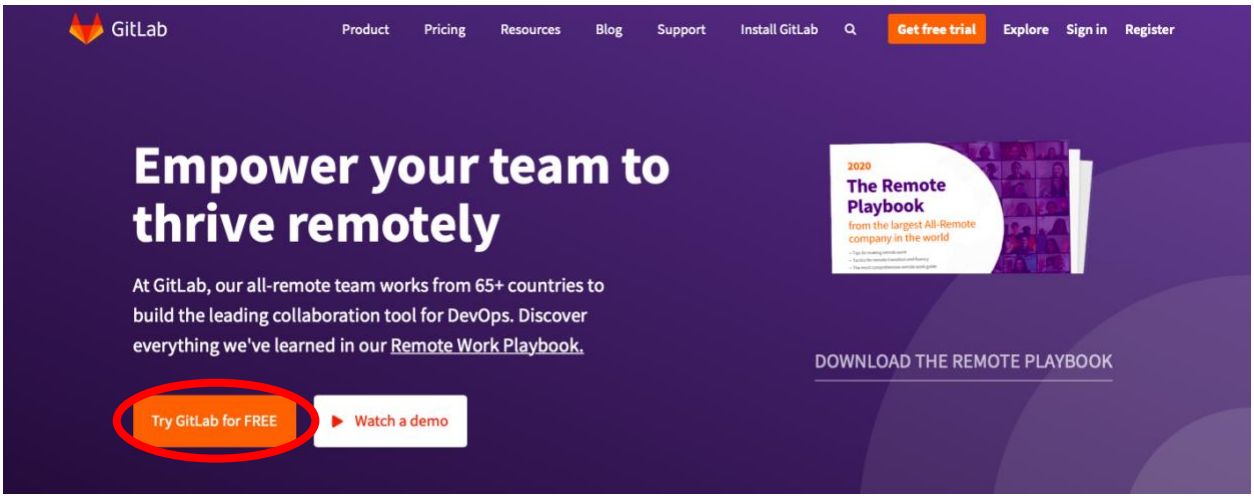
The files can either be viewed in “Unified” or “Split” mode. To switch between these views, click on the buttons on the right-hand side of the page:



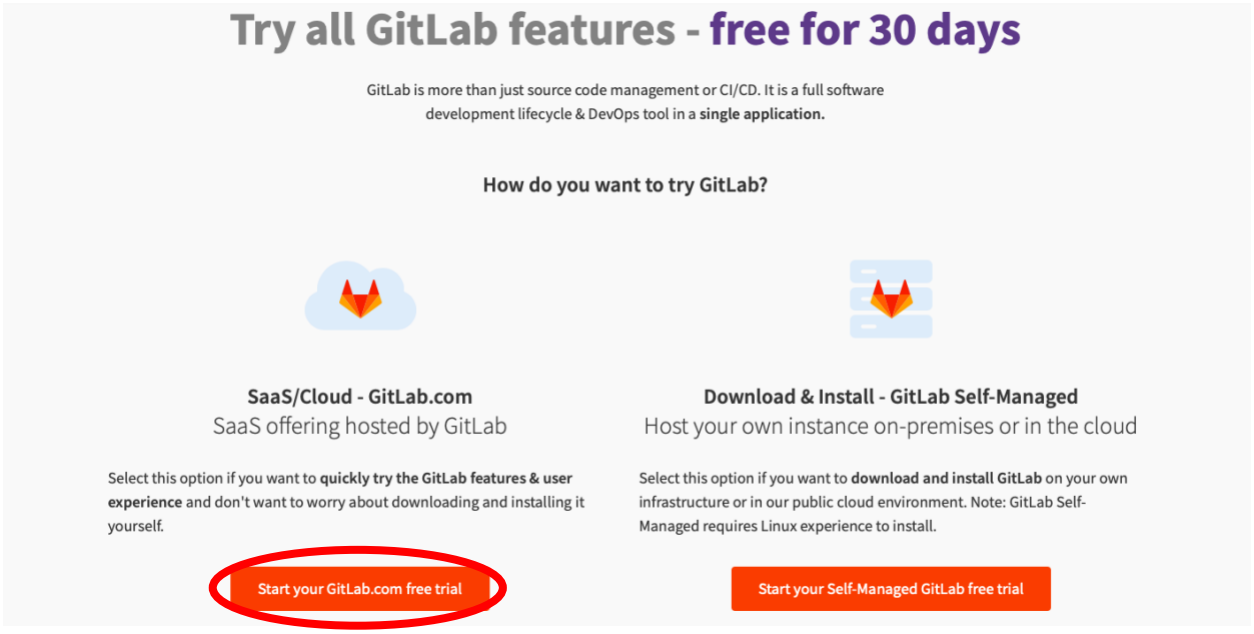
There you go! You can now begin to use GitHub to store your files and help keep your work clean. There are so many other things you can do with GitHub besides what is covered here. Refer to Section 5.2.2 in the Appendix for more helpful GitHub resources.

2.1.2 GitLab

GitLab is an online Git provider similar to GitHub. You can access many of the key Git features without ever typing code into your terminal. To create an account on GitLab for yourself[9], go to https://about.gitlab.com, and click on the "Try GitLab for FREE" icon.



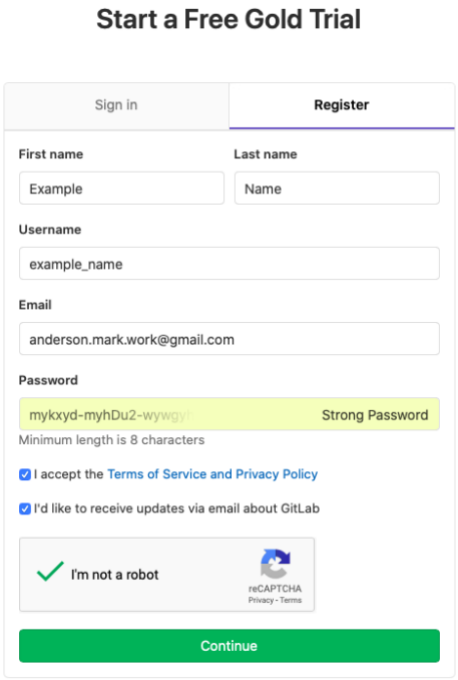
This will bring you to a page where you select which version you want to install. You can choose to download GitLab onto your computer, but for now we will choose to keep it online.



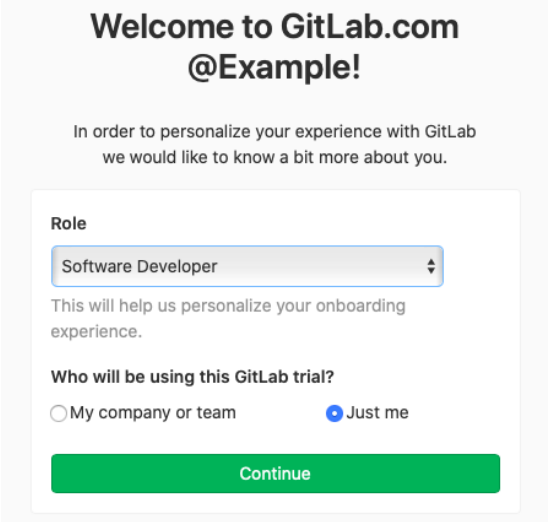
You can ignore the notices about losing functionality after 30 days because we will choose the very functional free version. After clicking on the "Start your GitLab.com free trial" icon indicated

[9] BYU physics students already have a GitLab account on http://git.physics.byu.edu and can sign in with their netID and password and can do this tutorial using that account if desired.

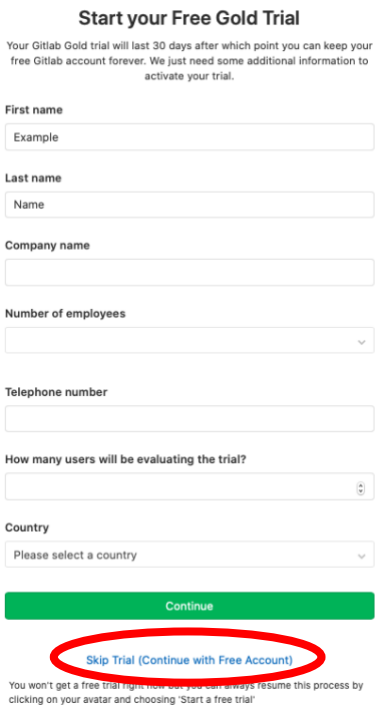
above, you will be led to a page to either sign in or register. If you do not have an account yet, then click on the “Register” tab. Fill in the information appropriately and then click on “Continue”.



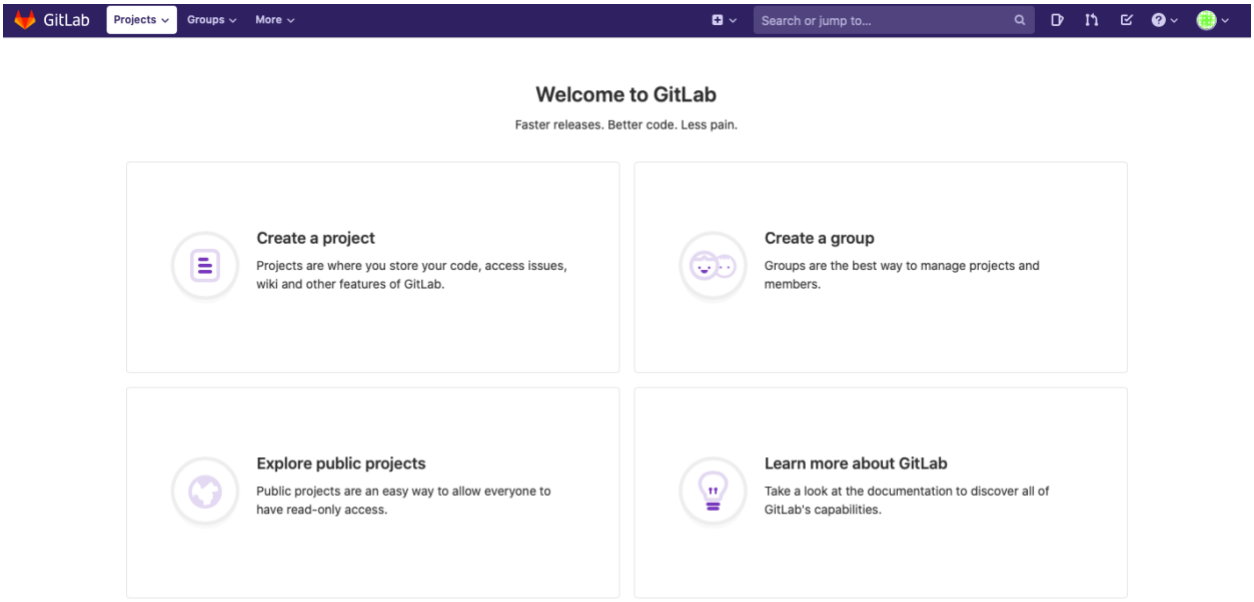
You will be directed to a page where they ask some information about you and why you’re using GitLab. Fill this out appropriately and then click “Continue”.



On the next page they will ask you to fill out some more information about yourself as part of the “Gold” level free trial. You may fill this out or skip the “Gold” version free trial and just use a free account by clicking the blue text at the bottom of the page. This tutorial uses the free account, so everything shown here is available for free to any user.

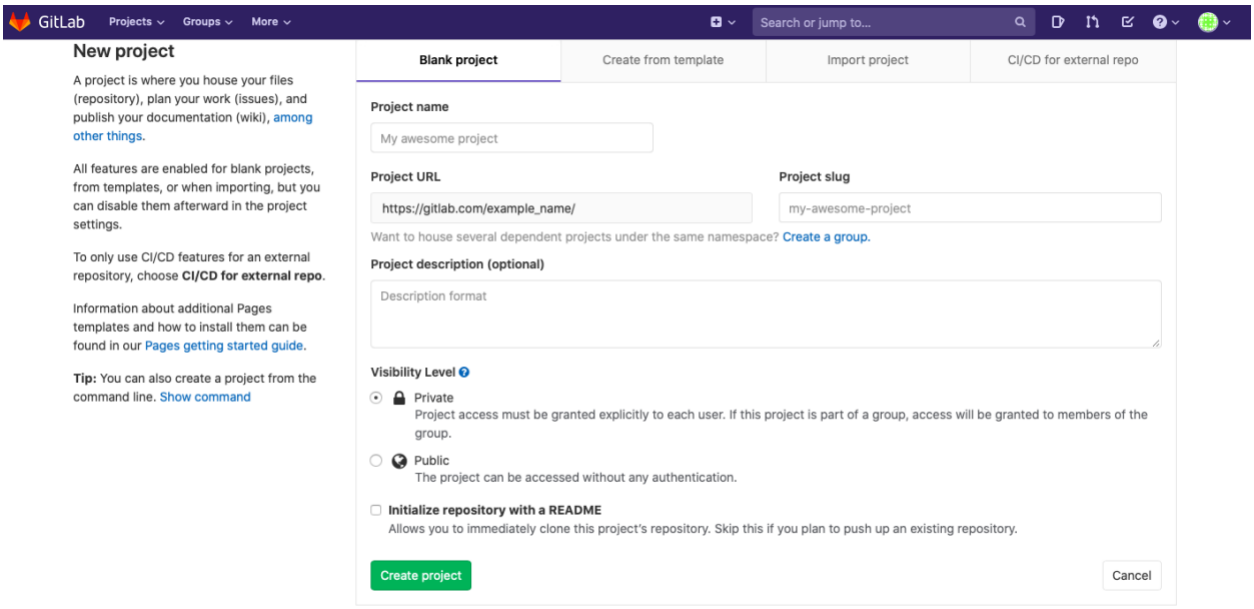


You will be redirected to the following page, and your account has been created.



2.1.2.1 Creating a Repository

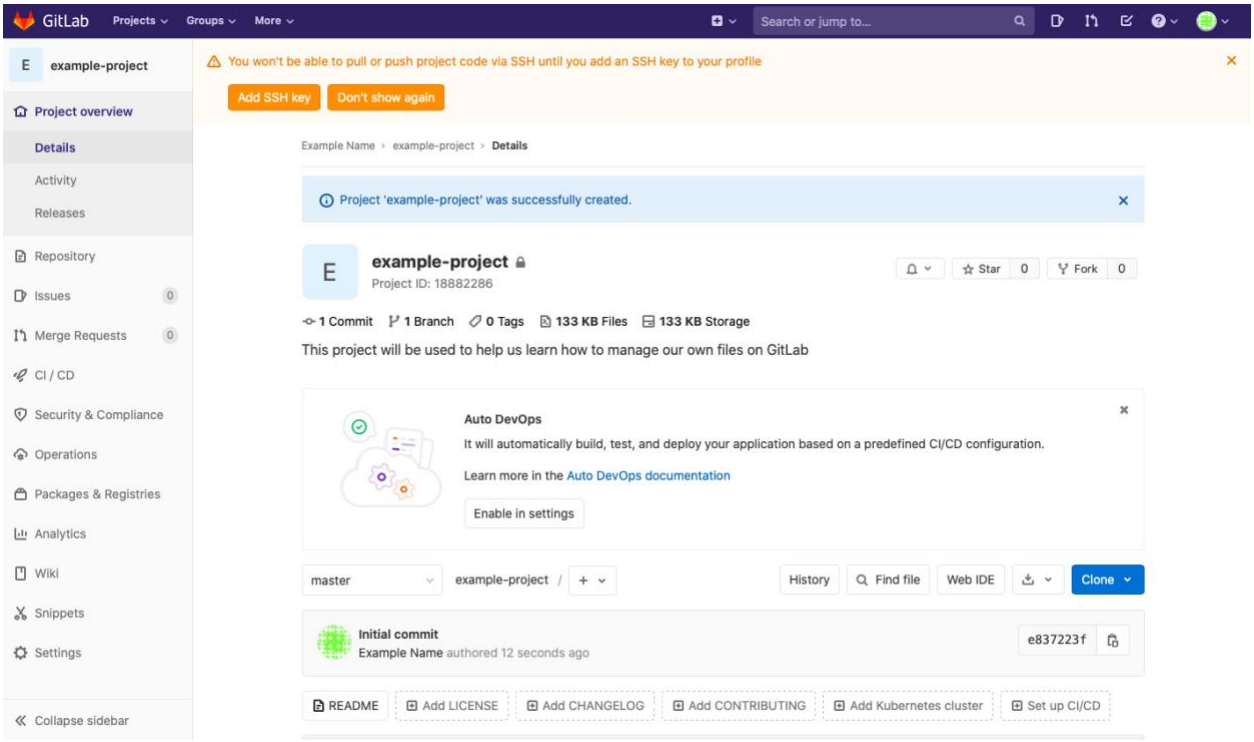
In GitLab, repositories are called “projects”, and we will use the word “projects” in this tutorial. Click on the “Create a project” button[10] to create a new project. You will be directed to the following page:



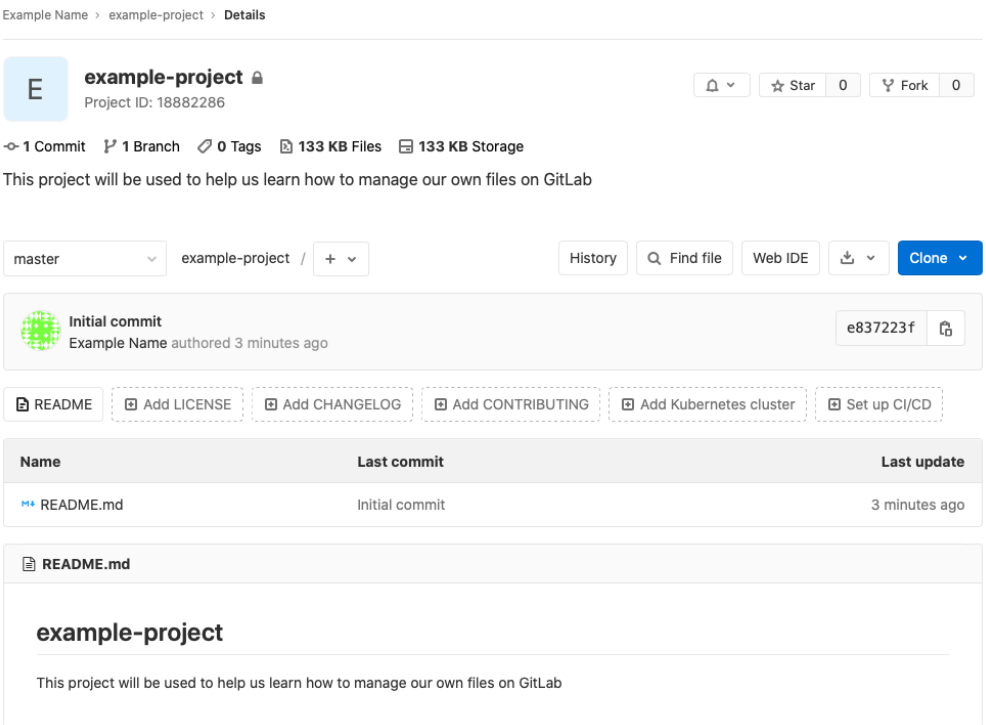
Fill out the boxes. The “Project name” box is where you officially name the project. Make sure to choose a name that is short and sweet, but descriptive enough that nobody will question what it is. The “Project slug” box will be filled out automatically according to the “Project name” box but will replace any spaces with dashes. For simplicity, it is often desirable to give your project a name that does not have any spaces so it will automatically be the same as the project slug. The “Project description” box is a place to include more detail about the project and should be filled out each time you make a new project. GitLab defaults to making the project private, so that only you and people you give permission to can see the project. Lastly, you want to check the “Initialize repository with a README” box. This will provide people viewing your project with a description of the project when they view it. When you’re ready, click on the green “Create project” box.

You will see a page similar to the following:

[10] If you’re not creating a new account, but rather signing into an already-existing account, there will instead be a green button that says, “New project”. Click on that to create a new project.



The dialogue box at the top that talks about SSH keys can be ignored for now. Click on “Don’t show again” (this feature can be enabled later but not needed for this tutorial). After exiting from the blue dialogue box and the “Auto DevOps” dialogue box, you will see the following:



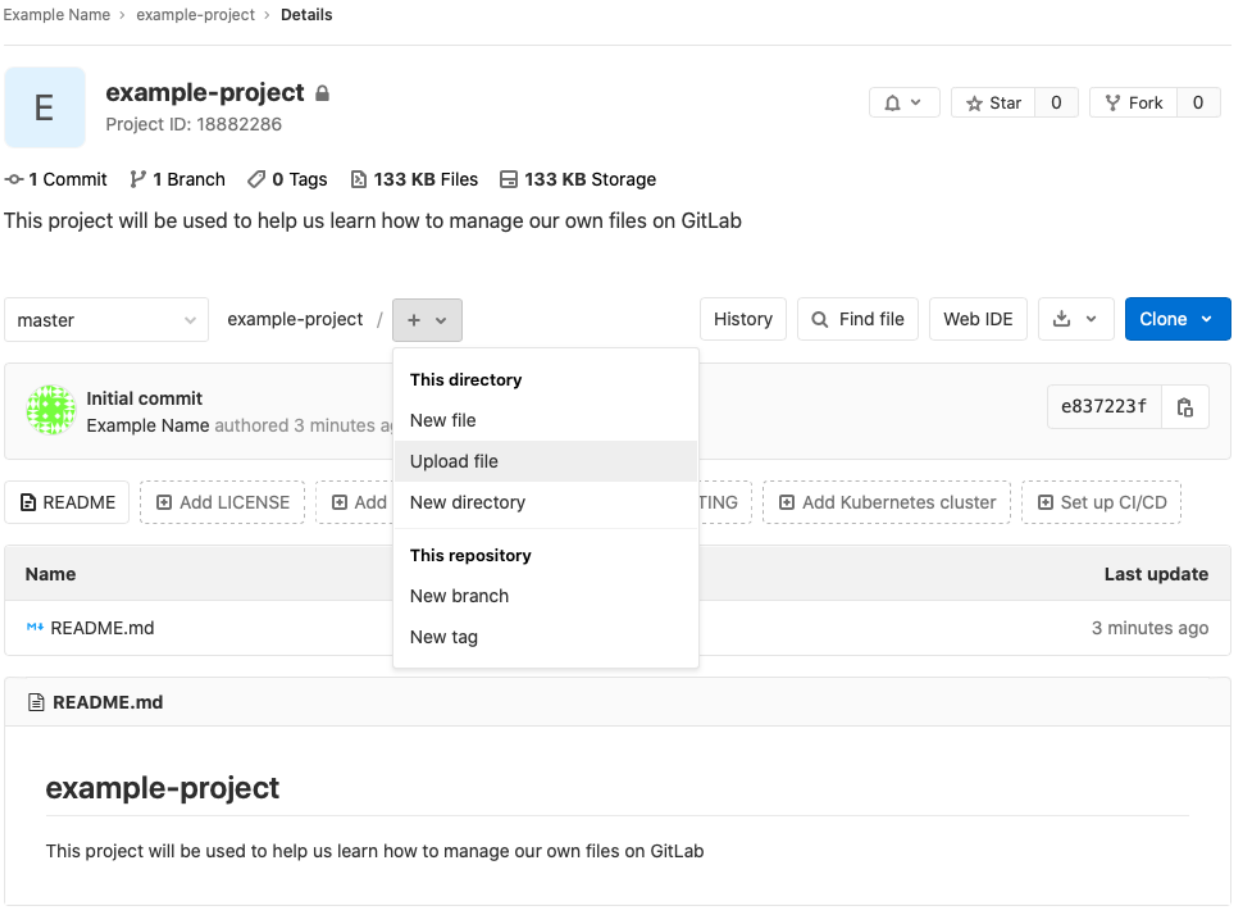
You have now successfully created a project on GitLab. This will be a place to store your files and where you can see different versions of your file over time. Eventually, you can also add others to the project so you can work in a team.

2.1.2.2 Uploading Files

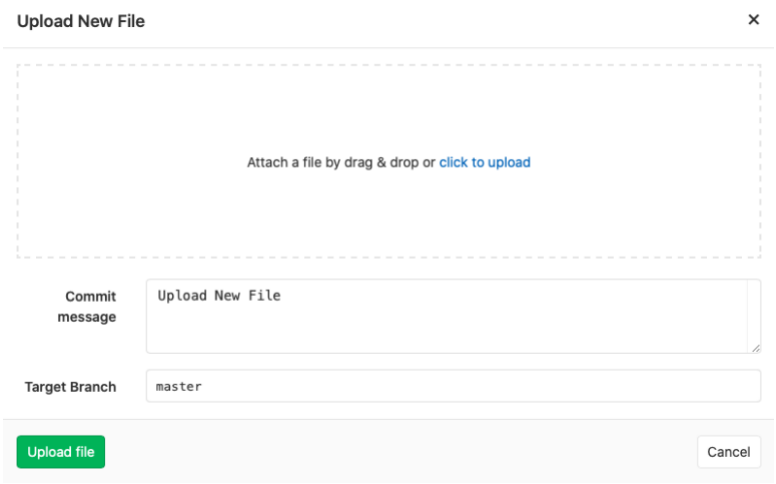
Let's say that you have two files in your project that are saved on your computer:

some_math.m
important-notes.txt

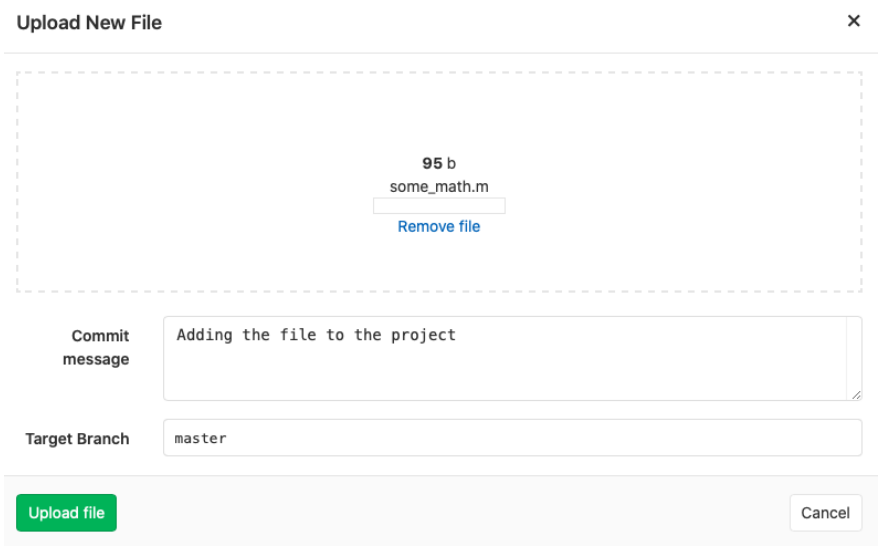
You would like to upload them to GitLab so that you can track all of the changes that you make to them over time and maybe someday collaborate with others on making these files better. To upload these files, click on the plus sign icon and select "Upload file":



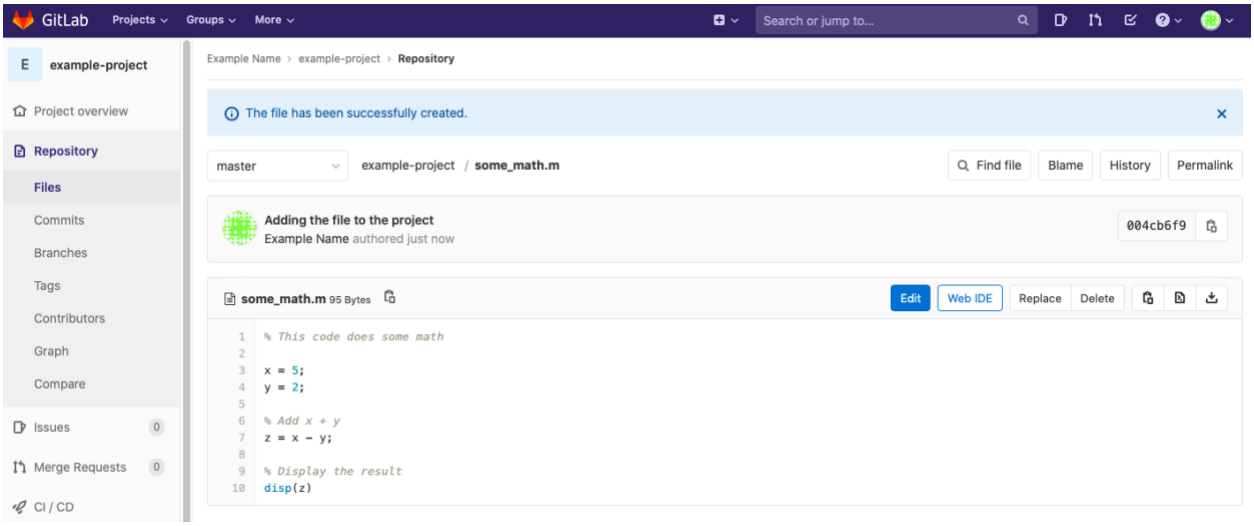
The following box will appear:



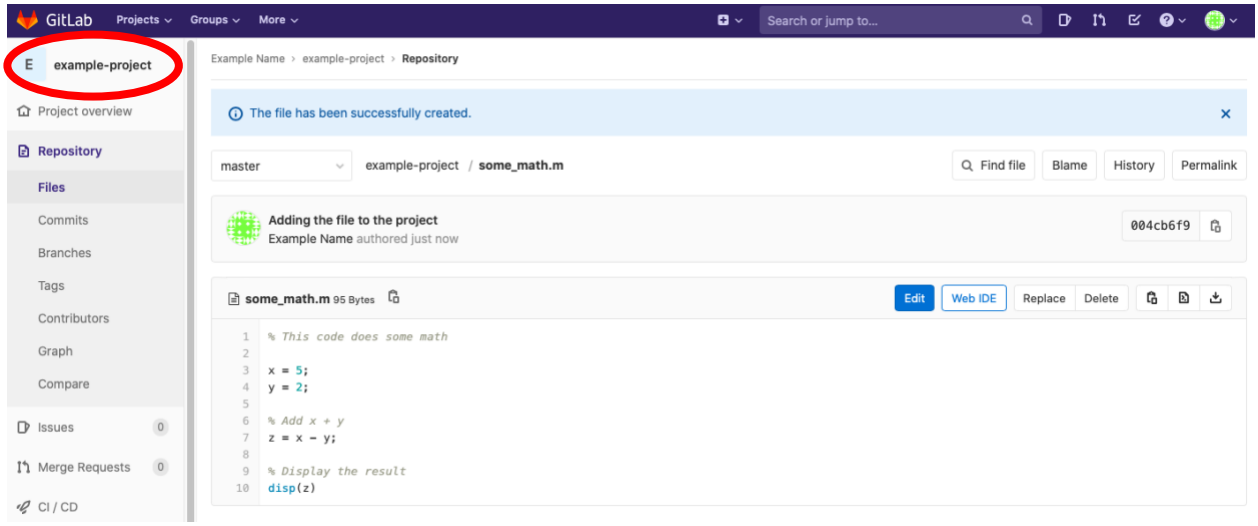
Click on the blue “click to upload” text and select a file you would like to upload (file uploads must occur one at a time, but you can add an entire folder at once if you want). Then type a message in the “Commit message” box. This message will appear next to these files in the project so make sure that it’s a short yet descriptive message. Click on the green “Upload file” icon to complete the upload. Do this for all files that you want to add to the project.



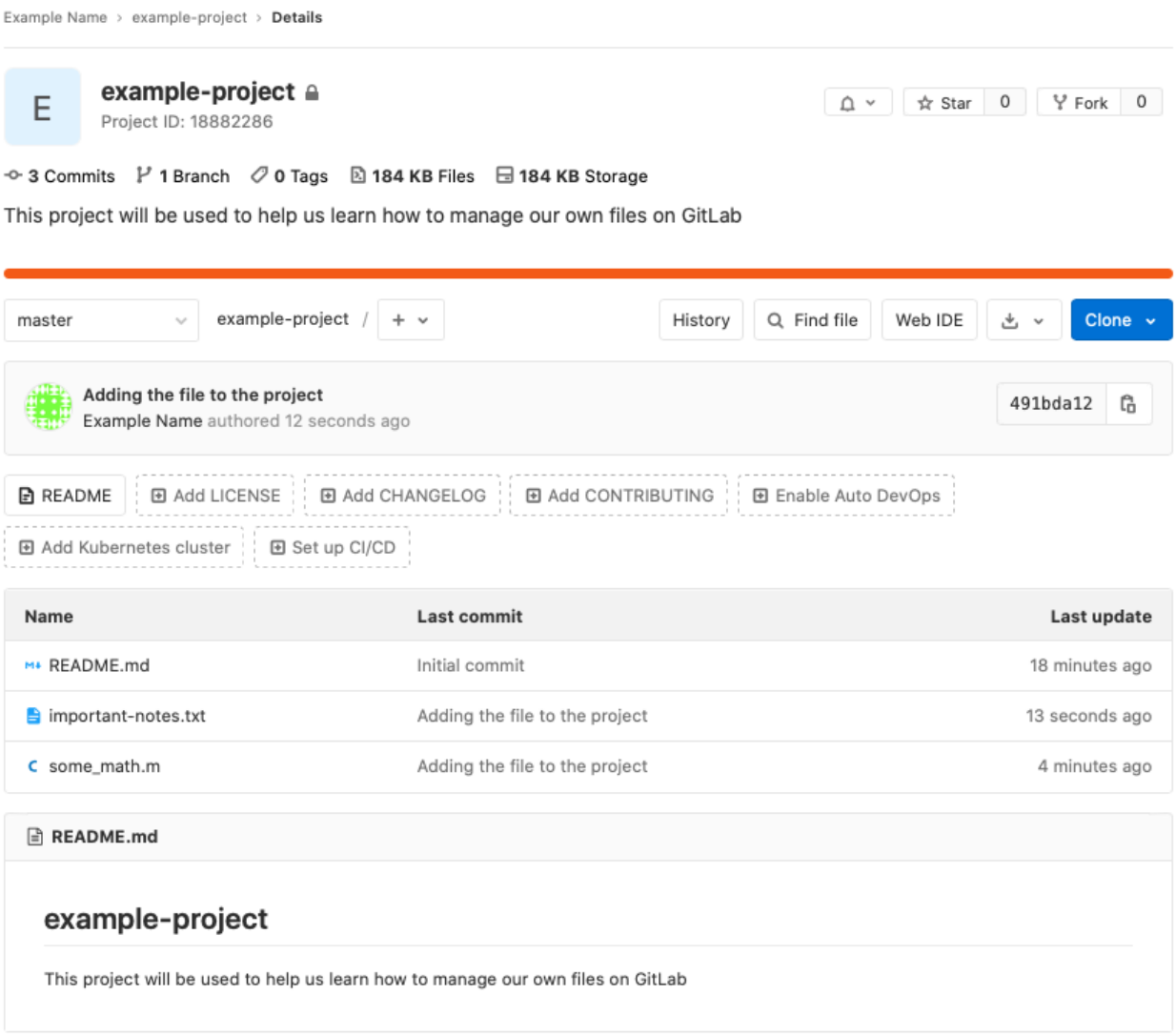
GitLab might open the file immediately and you may see a screen that shows the contents of the file, like this:



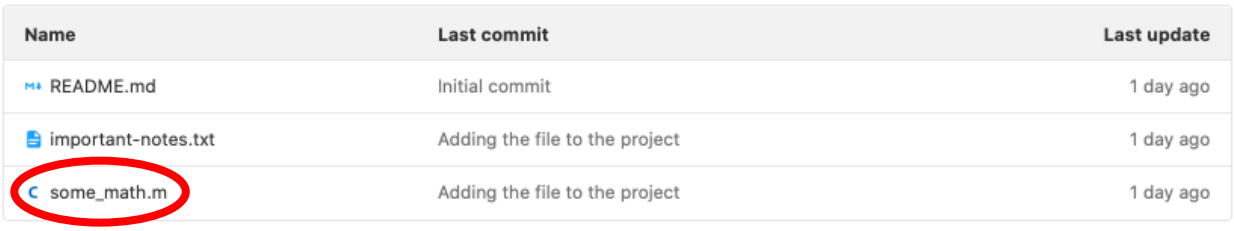
To return to the main project page, click the project's icon in the upper left corner of the web page:



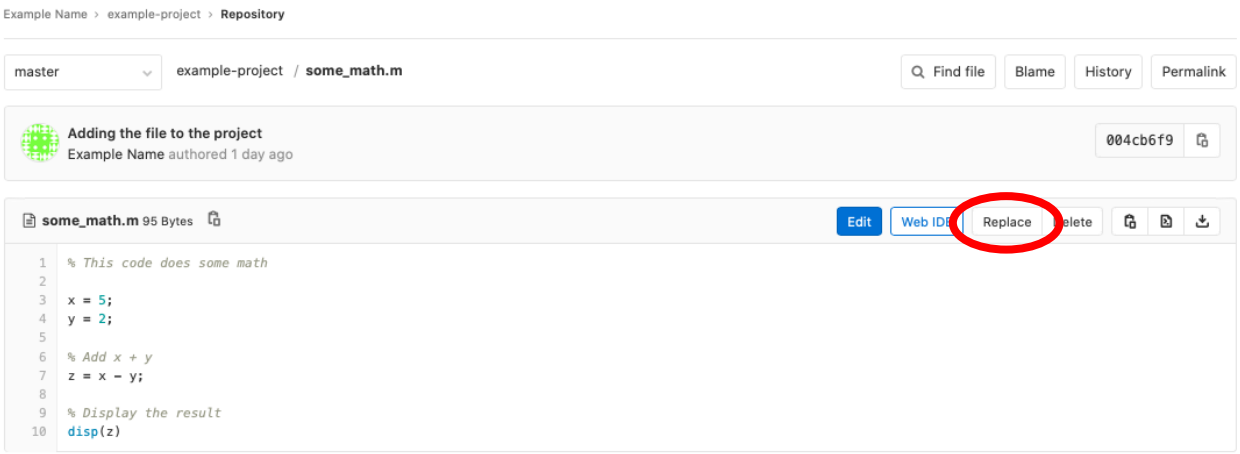
Once your files are added, the project's home page will reflect those additions:



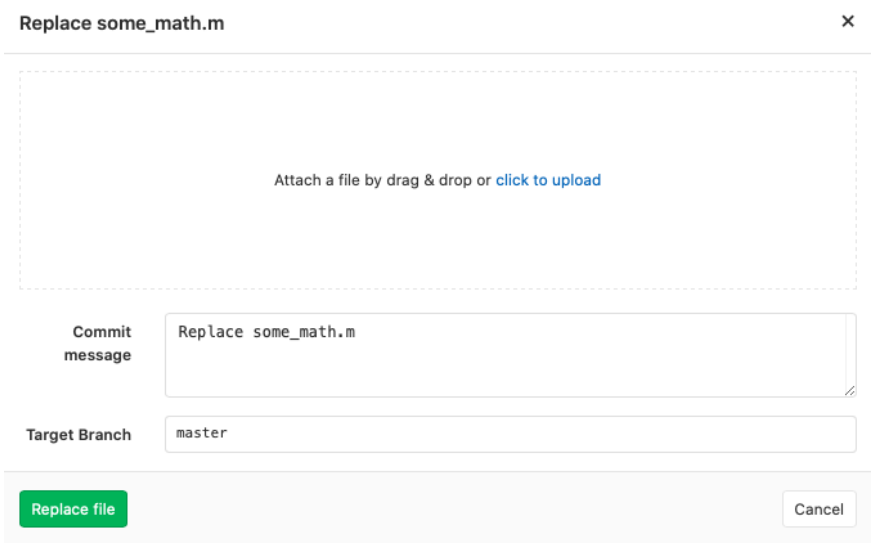
Now let's say that you have changed some_math.m and would like to store the new version on GitLab. Click on the file you would like to update:



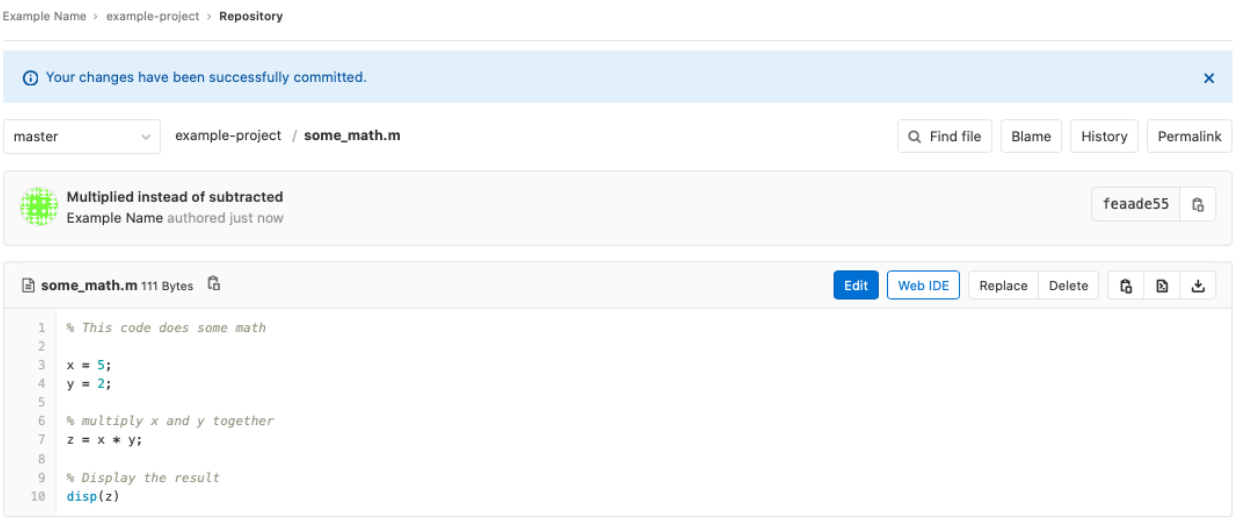
You will be directed to a page that shows you the contents of the file. Click on the "Replace" icon.



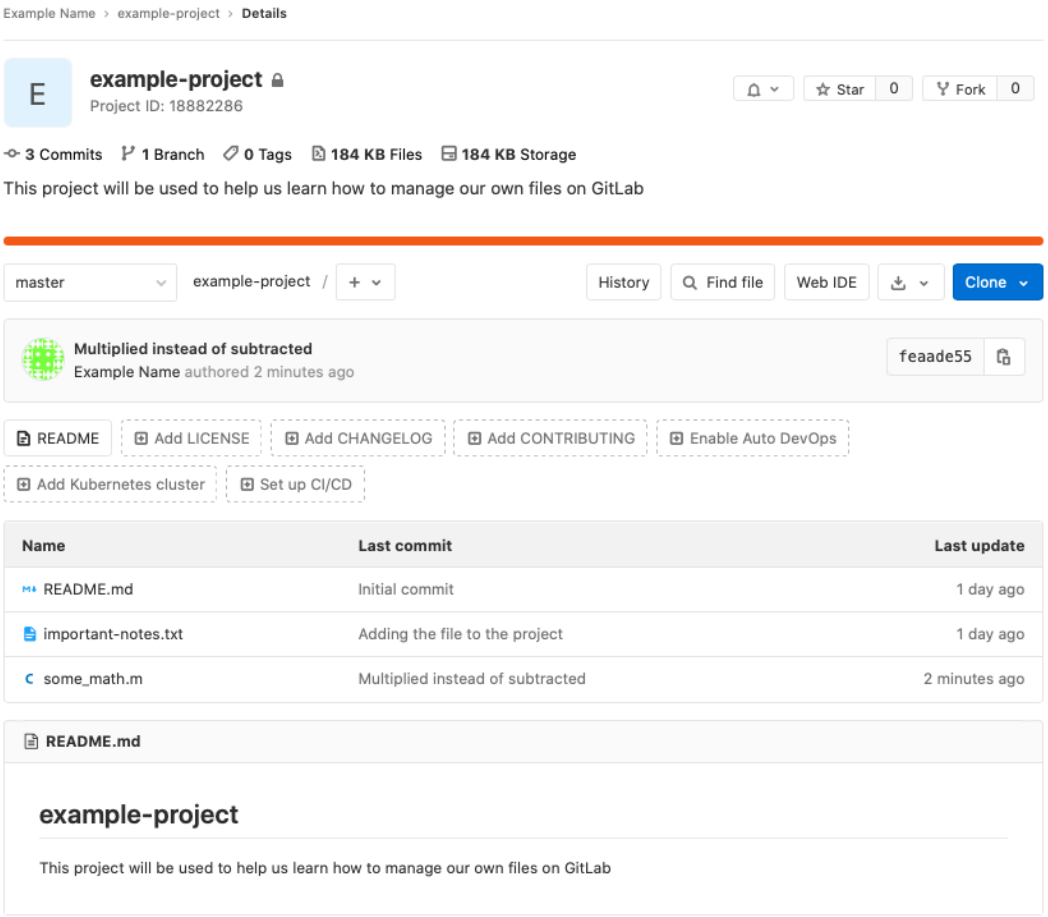
The following box will appear:



Click the blue text to upload the new version of the file, write a simple commit message, and then click on “Replace file”. You will then see a page that shows the new version of the file:



Return to the main project page, and you will see the updated file with its commit message:

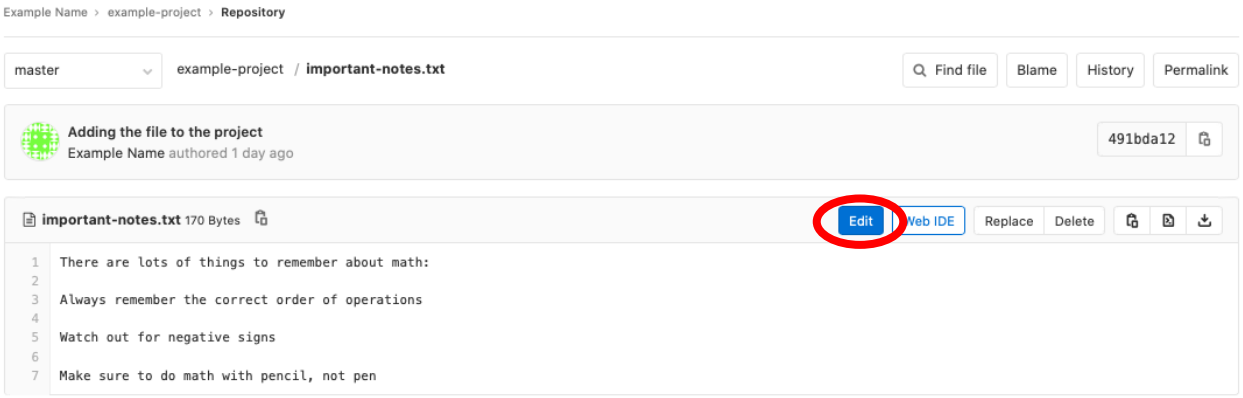


Congratulations, you can now upload files and new file versions to GitLab. This will help you to keep your files safely stored online.

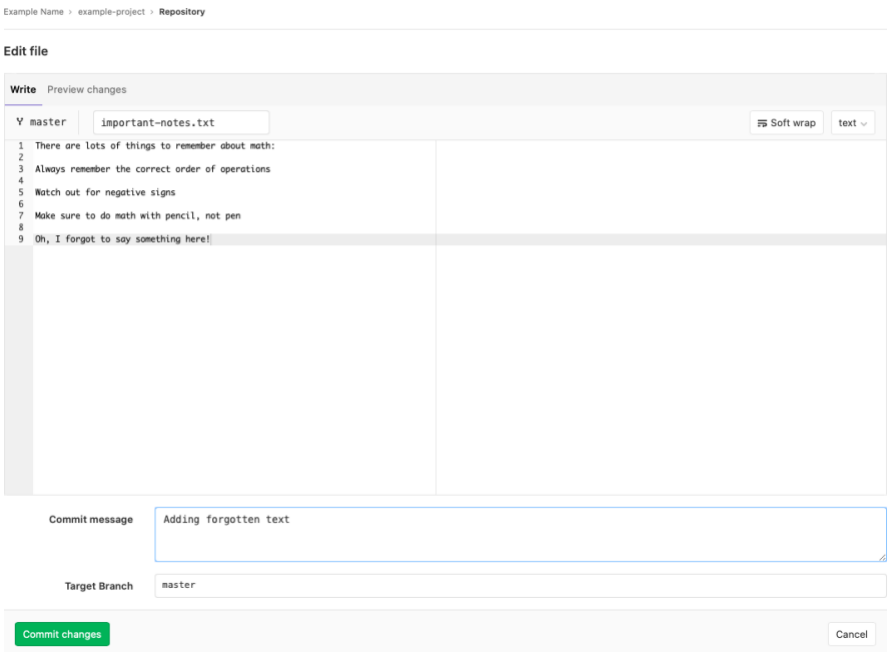
2.1.2.3 Editing Files Online

Although editing files online is generally a risky practice, it is possible. It is risky because it means that the files on your computer will not have those same changes, and then if you try to upload a new version from your computer the changes you made online will disappear. However, if you find yourself in a situation where it is appropriate to edit a file online, this is how you do it.

Click on the file that you would like to edit; in this example we'll edit the important-text.txt file. This will take you to a page that displays the contents of the file. Once here, click on the blue "Edit" button.



Make some changes, and then type an informative commit message before clicking on the green "Commit changes" button.



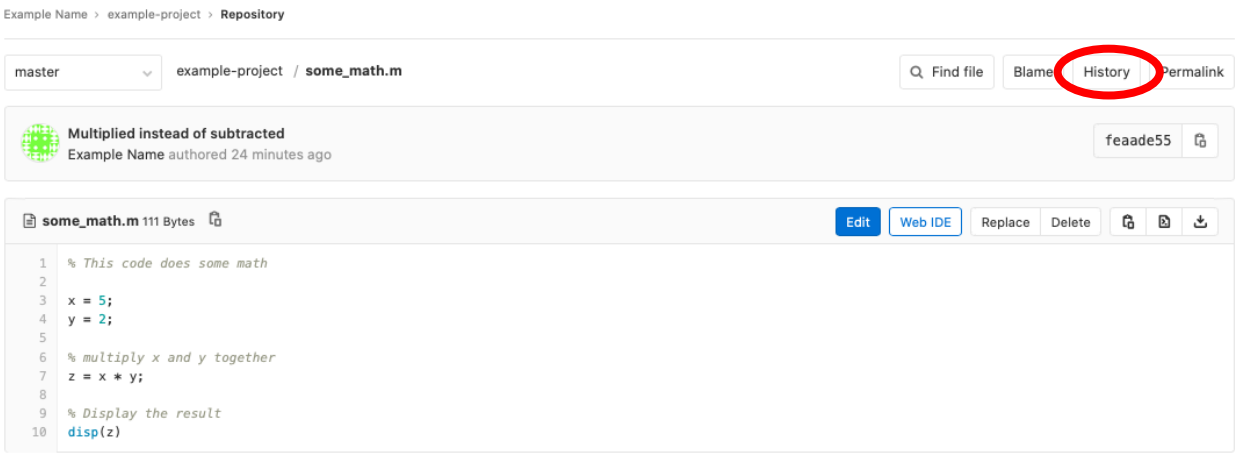
Navigating back to the main project page, you will see the updates:

Name	Last commit	Last update
README.md	Initial commit	1 day ago
important-notes.txt	Adding forgotten text	just now
some_math.m	Multiplied instead of subtracted	15 minutes ago

You can now make edits purely online.

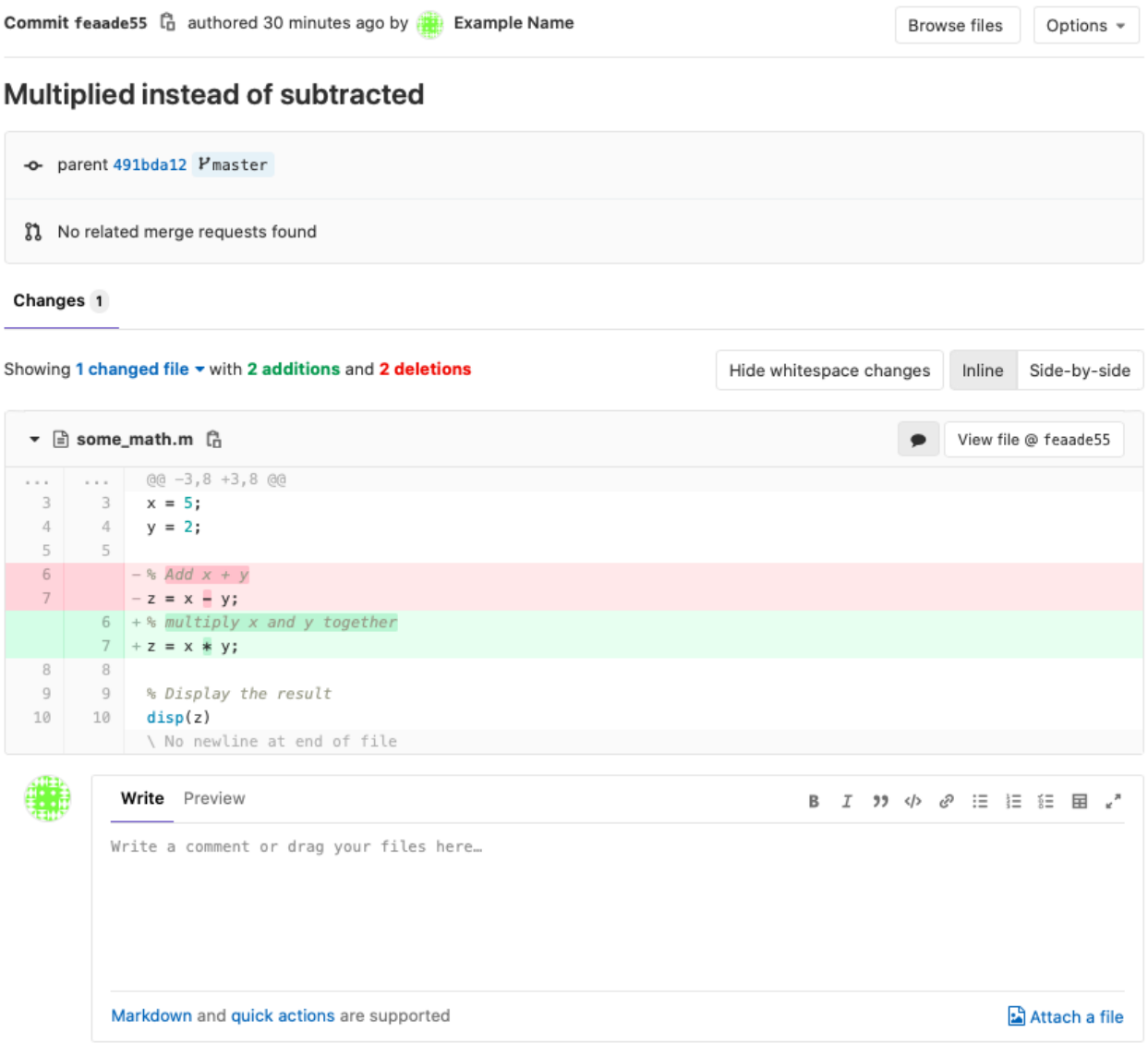
2.1.2.4 File History and Comparing File Versions

Now let's say that you want to see how your files have changed over time. This is useful if some recent changes you made have introduced new bugs or if you want to see when certain changes were made. To do this, first click on a file in the main project page to view its contents. This will lead you to the following page, where you will then click on the "History" icon:



You will see a page with the different versions of that file. Click on one of them.

You will now see a page that shows you the differences between the files. If you want to see the differences side-by-side, then click on the "side-by-side" icon on the right-hand side of the page.

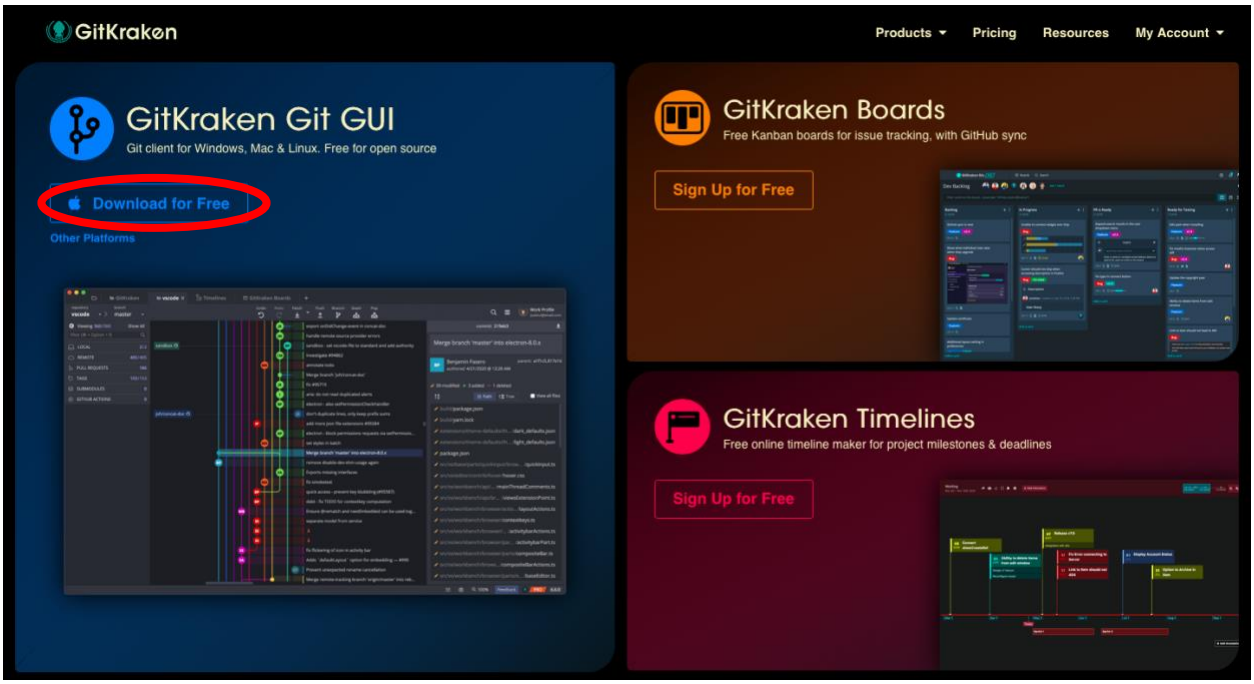


And there you go; you can now begin using GitLab for basic version control with your files. Read Section 3.3 for more information about working in groups using GitLab.

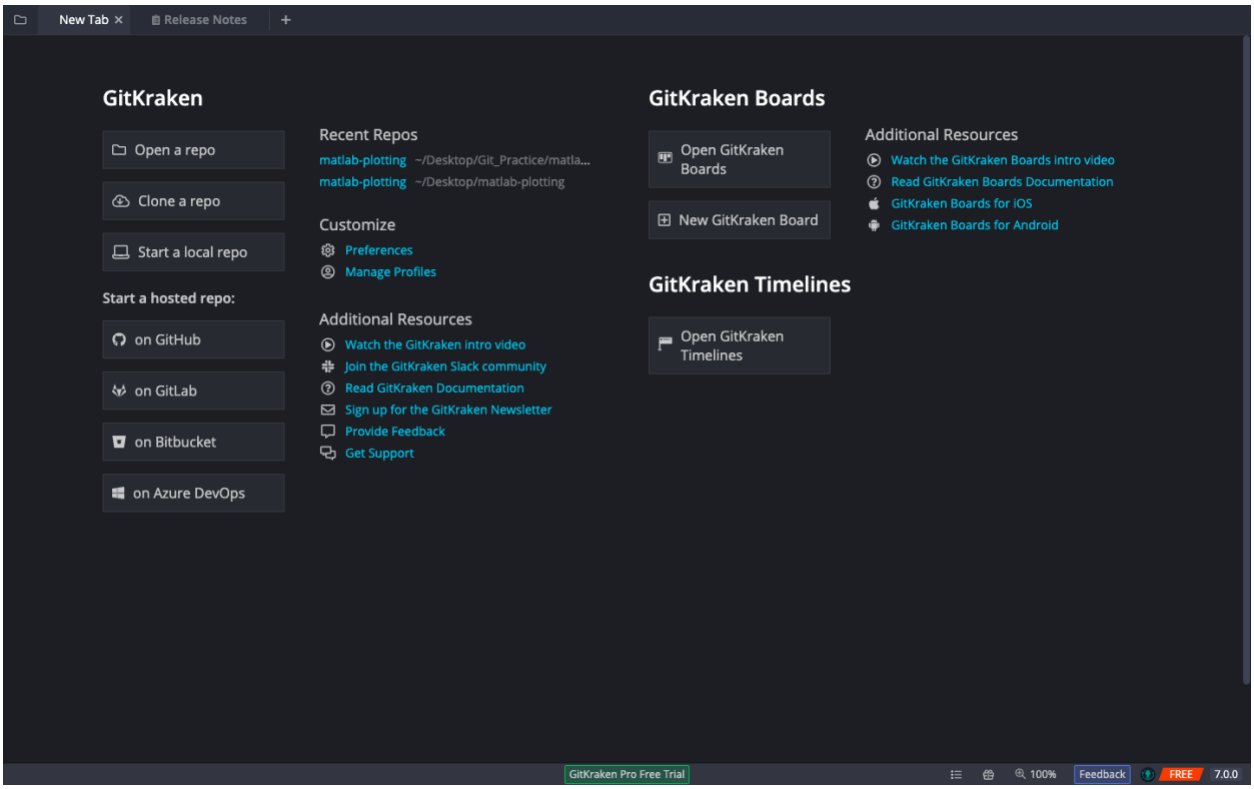
2.2 Local (On Your Computer) Providers

2.2.1 GitKraken

GitKraken is an application that you can install on your computer. GitKraken will allow you to use Git in a GUI interface, which means that you can navigate Git with your mouse in a user-friendly environment. To install, go to https://www.gitkraken.com and select the "Download for Free" option. The web page should automatically detect your operating system and pull up the proper download. If not, select "Other Platforms" to find the download for your operating system.



You may be brought to a web page thanking you for downloading the file. Once the file is downloaded, install GitKraken using the download on your computer. Once it's installed, open it. Your computer may warn you about it being an application downloaded from the internet. Open it anyway. If a box appears that asks you to sign in with either a GitHub or GitKraken account, go ahead and exit out of that dialogue box. This will bring you to the following page:



GitKraken is now successfully installed on your computer. Before we continue with this tutorial, you will need to connect your GitKraken account to either a GitHub or GitLab account so that we can access repositories that are stored online. To do this, click on the "Manage Profiles" button underneath the "Customize" section.

For students and faculty, you can upgrade to a Pro version of GitKraken by associating your GitHub[11] account with your university email address, discussed in Section 5.1 of the appendix. For others, you can upgrade your account to Pro according to GitKraken's pricing.[12] The key difference is that if you use the Pro version you can have private repositories that only you can see. If you decide to stick with the free account that's just fine, as long as all of your online repositories are public rather than private. You can always get the student upgrade later. It's important to note here that GitKraken says this free version of GitKraken Pro for students is only valid for one year.[13]

2.2.1.1 Creating a Repository

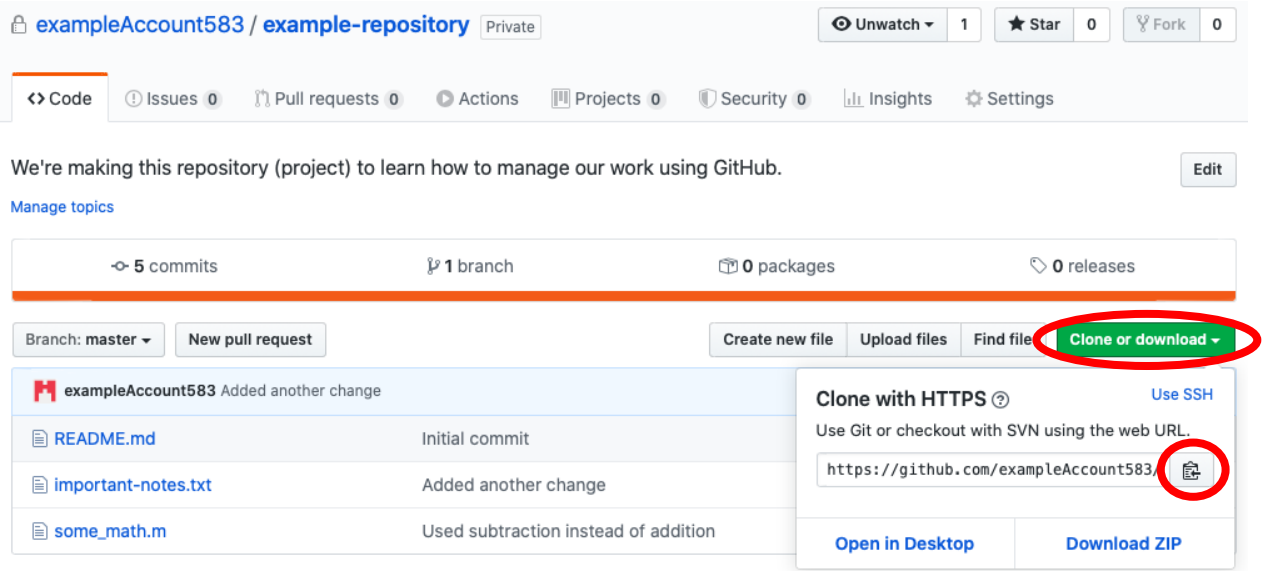
GitKraken provides lots of options for creating a repository. You can open a repository that you've already made, clone a repository from an online provider like GitHub or GitLab, or start a repository that stays only on your computer and is not synced up to an online provider. Lastly,

[11] Unfortunately, this is only offered through GitHub, so if you want the free upgrade to GitKraken Pro you will need to create a GitHub account, even if for the sole purpose of getting the free upgrade to GitKraken Pro.
[12] https://www.gitkraken.com/pricing
[13] As of the present writing in June 2020. This of course may change in the future.

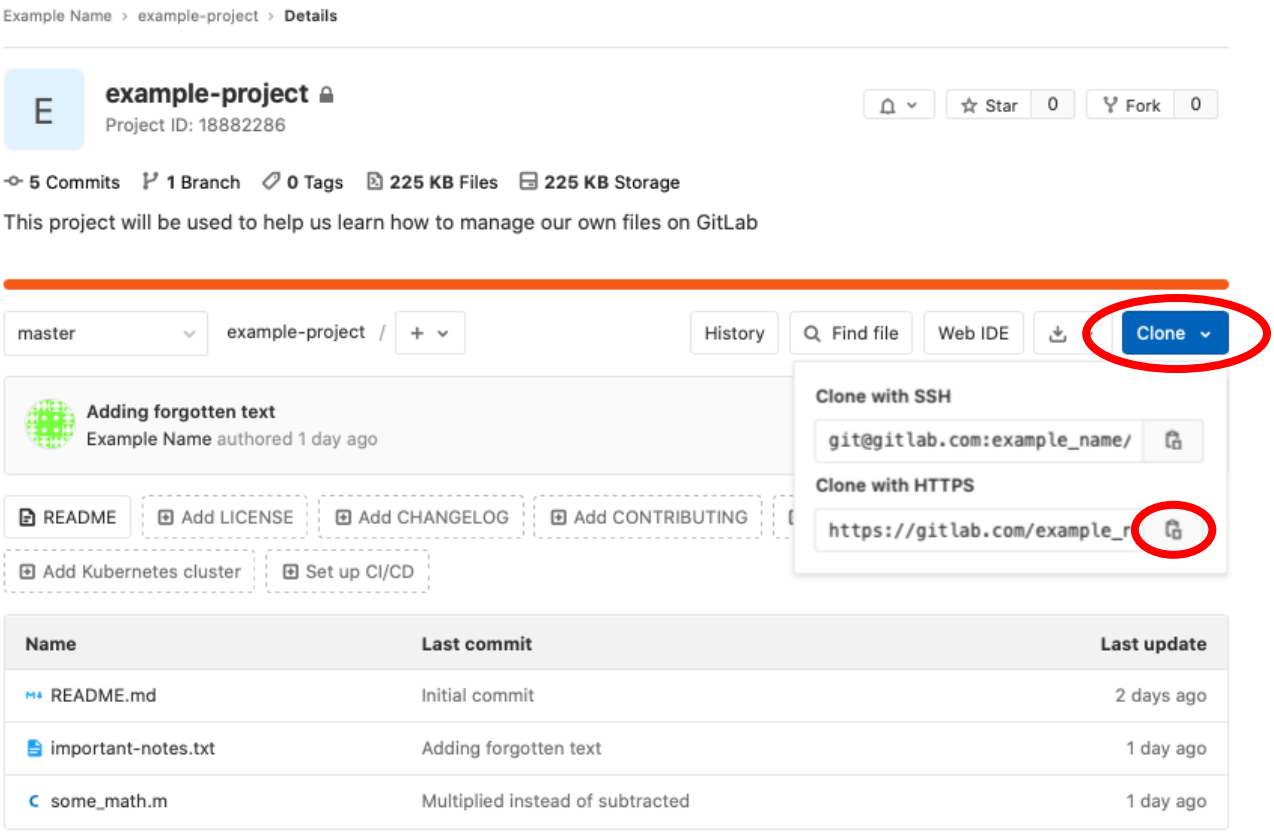
you can create a repository that is automatically synced up to an online provider. This last option is called “Start a hosted repo”. These are all good options that you can learn more about as your skills with Git and GitKraken grow. In this tutorial, we will take an existing repository from GitHub and “clone” it to our computer. Cloning means that we will make a copy of the project on our computer, where we can make our own edits, test them out, and then “push” those changes back to the online repository where they will then become available for other users.

2.2.1.2 Cloning a Repository

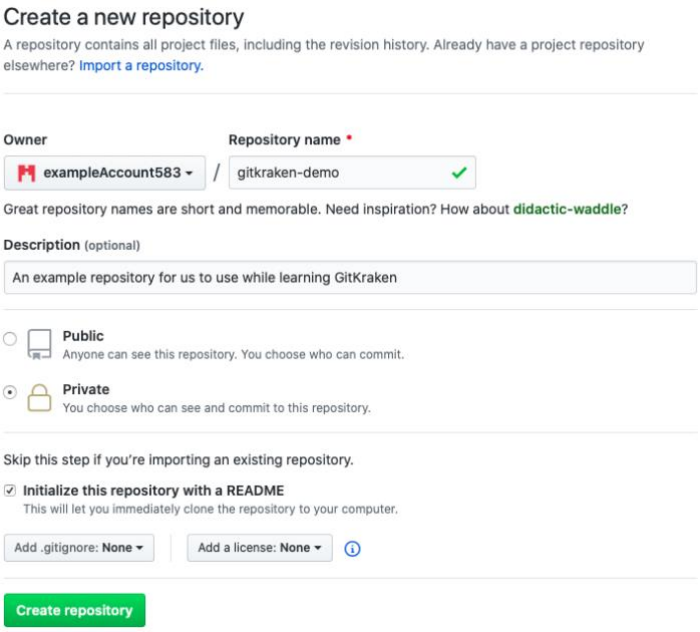
To clone a repository, open the repository on either GitHub or GitLab and click on the “clone” icon. This will open up a box where you should copy the link to clone with “HTTPS”. For GitHub, this is shown in the following screenshot:



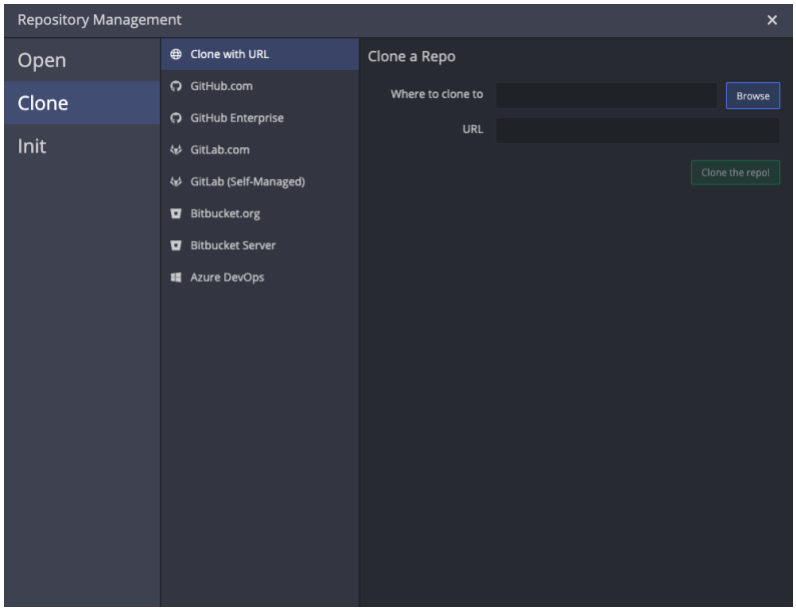
If you want to clone a repository from GitLab, this is shown in the following screenshot:



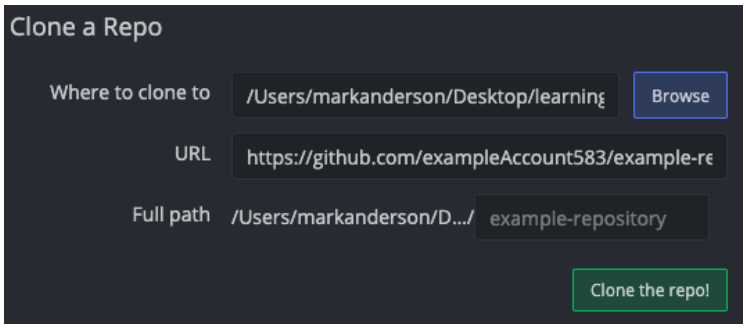
For this GitKraken tutorial, I have gone into GitHub and created a new empty repository called “gitkraken-demo”. You can follow along closely with this tutorial by creating a similar repository either on GitHub or GitLab. For a tutorial of how to create a repository on GitHub or GitLab, refer to Sections 2.1.1.1 and 2.1.2.1 respectively. For your reference, below is the repository creation page on GitHub:



Once the repository is created, click on the green “Clone or download” icon and copy the HTTPS link. Then return to GitKraken and click on the “Clone a repo” icon on the left-hand side of the welcome screen. You will see the following box:

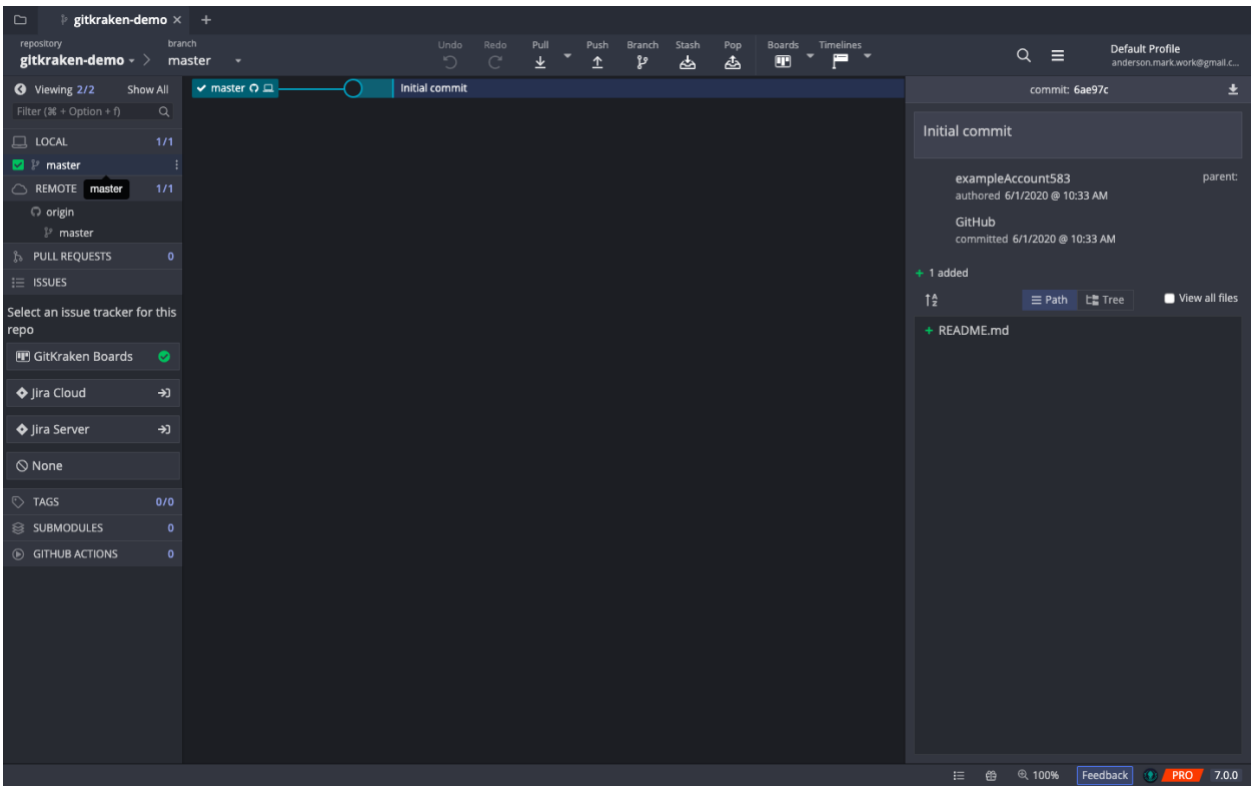


Click on the “Browse” button to select a place on your computer to store the repository. Then paste the “HTTPS” link that you copied from either GitHub or GitLab and paste it in the “URL” box.



Click on the green “Clone the repo!” button. After the repository is successfully cloned to your computer, you will see a banner across the top of the screen that says:

Go ahead and click on the “Open Now” button to open the repository. You will be brought to the following page:



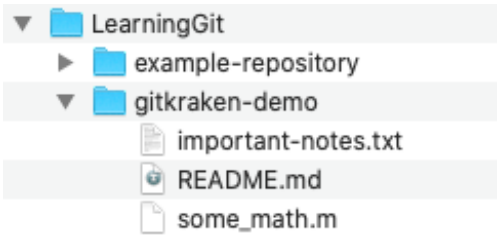
This page may look daunting with all of the buttons everywhere, but we will go through everything you need to know to get started with GitKraken.

2.2.1.3 Getting files into GitKraken

Let’s say that you have two files that you would like to keep track of with GitKraken:

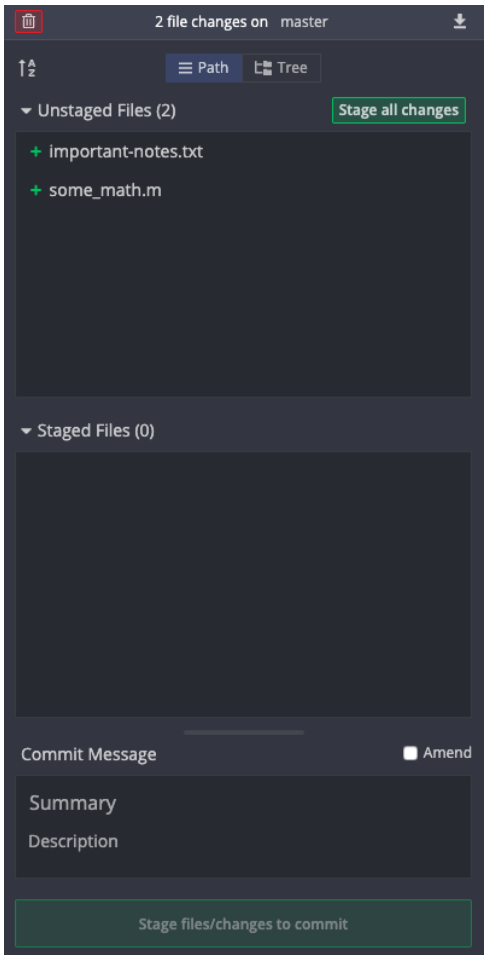
some_math.m
important-notes.txt

When you cloned the repository to your computer it created a folder named after the repository. Find that folder and put the files that you want to keep track of in that folder. Here is an example of what this might look like:



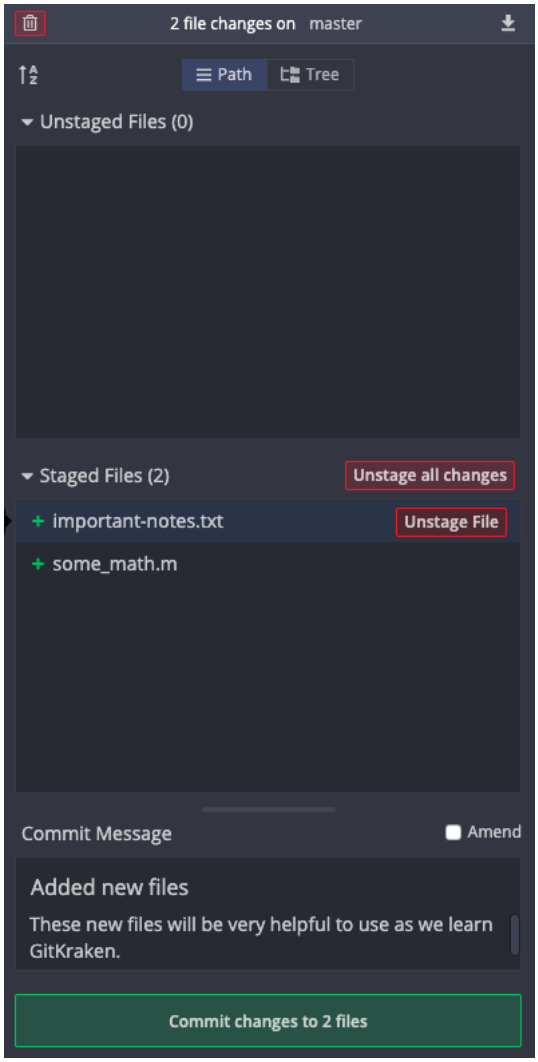
GitKraken is aware that there are now new files in the repository and when you return to the GitKraken screen you will see some changes on the screen. In particular you will see this in the "tree":

The little "+2" means that there are two file changes (adding the files counts as changing them). The box next to the "+2" is grayed out because the changes represented there have not been committed to Git yet. To commit them, click on the grayed-out box. On the right-hand side of the screen you will see:



Right now, the two new files have not been staged. If we try to do a commit right now, they will not be saved because we never told Git that we wanted them to be part of the commit. If we want them to be committed then we need to add them to the staging area, which tells Git that we want them to be committed to the repository on the next commit. To stage a file, hover over it and press the green “Stage” icon that appears. Or if you want to stage all of the files press the green “Stage all changes” icon.

It is very important to include a commit message explaining what you changed about the files (in this case we just added them, but if we had modified existing files then we would summarize those changes). In the “Summary” section put a very brief summary of the changes and if you need more room to explain the changes then type the detailed explanation in the “Description” section. Once this is complete the right-hand-side of the screen should look something like:



Once you're satisfied that these are the changes you want to make and that you've described them well, click on the green "Commit changes" icon at the bottom of the right-hand-side of the screen to commit the files. Now the center display will look like this:

Each line represents a commit, and the commit message for that commit is displayed next to it. The box that says "master" with a computer and checkmark next to it represents the version of the files that are on your computer. The other "master" box indicates the version of the files that are on GitHub (or GitLab, depending on where the online version of your repository is). It's important to note that the repository on your computer and the repository online are now out of sync. This bring us to the following section.

2.2.1.4 Syncing Changes with a Remote Repository

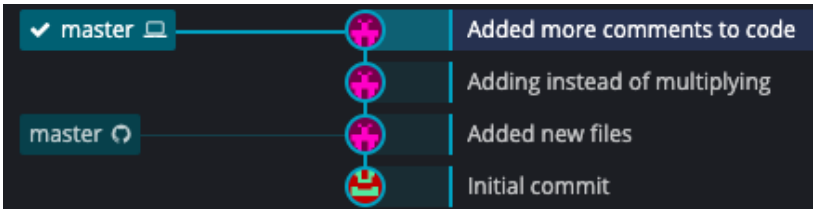
We need a way to re-sync our remote (online) repository with our local (on our computer) version of the repository. First, we must make sure that we download any changes that the remote repository has that we don't have on our computer. This is unlikely if you are working alone, but very likely if you are working in a team. To do this, go to the actions toolbar at the top of the screen:

Click on the "Pull" button to download any changes that the remote repository has that your local repository does not. After doing this (and perhaps making sure that your changes don't interfere with someone else's changes) click on the "Push" button to send the changes that you have made to the remote repository. Once successful, you will see that the tree looks like this:

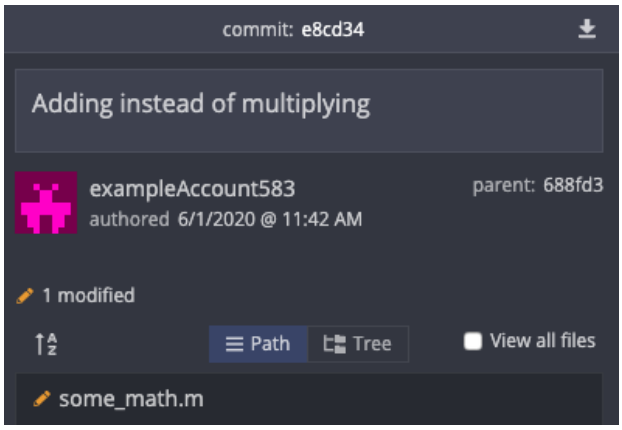
Now your remote and local repositories are back in sync. If you go online and look at the repository you will see that the changes you made (adding those two files) are officially part of the repository online. You do not need to push your changes to the remote repository after every commit, instead you can do it after every few commits or whenever is most useful for your particular project.

2.2.1.5 Comparing Past Versions of the File

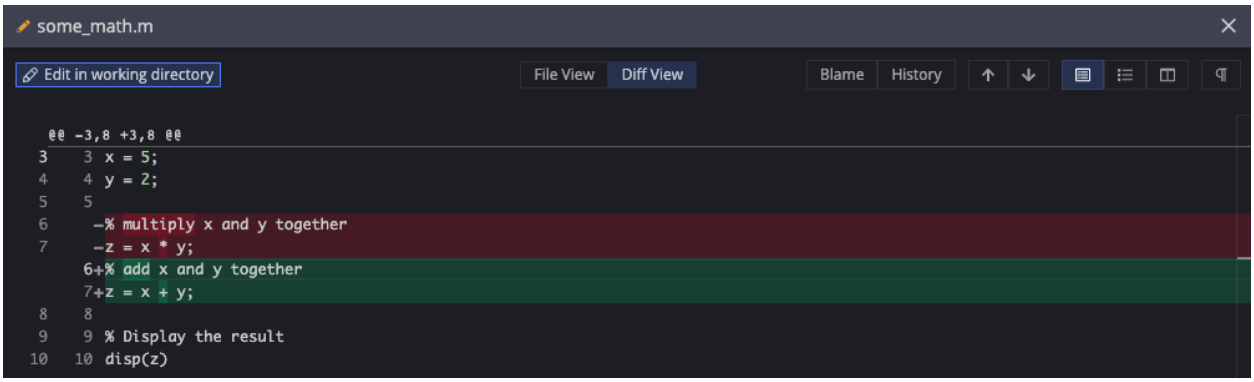
Let's say that you've made some changes to some_math.m and you have committed them to your local repository (the one on GitKraken) and your tree looks like this now:



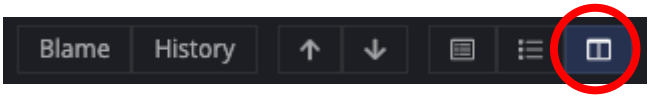
To see how the files have changed between commits, click on a commit and go to the right-hand-side of the screen:

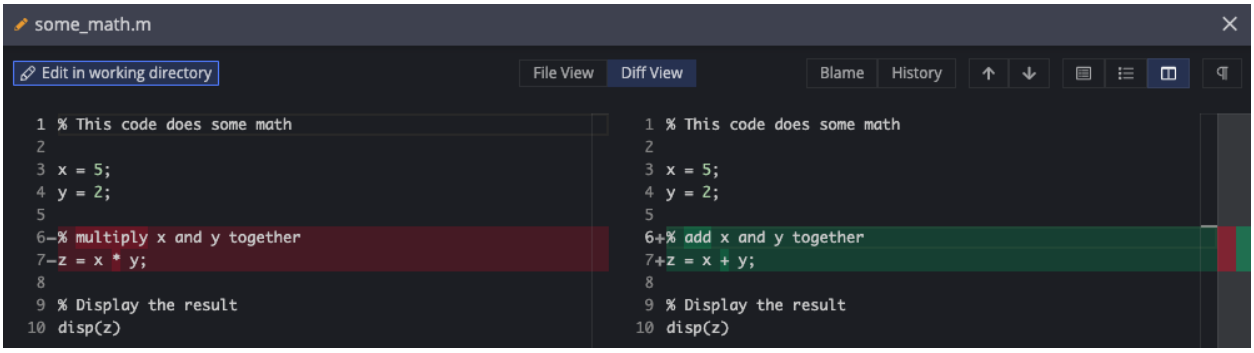


Click on the file that you want to view (in this case some_math.m) and the center of the screen will now show the changes that happened:



The red regions show what was removed and the green regions show what was added. In the upper right of this display there are some options for different views. The split view is a convenient way to view the file differences:





There you go! You can now use GitKraken for managing your files.

2.2.2 The Terminal / Command Line

The terminal is the original interface for Git and continues to be arguably the most efficient and effective way to use Git. Understandably, interfacing with Git this way may appear daunting and you may feel discouraged. This is normal, and this tutorial will tell you everything that you need to know to start using the terminal, and once you know this basic functionality, the other features will come to light as needed.

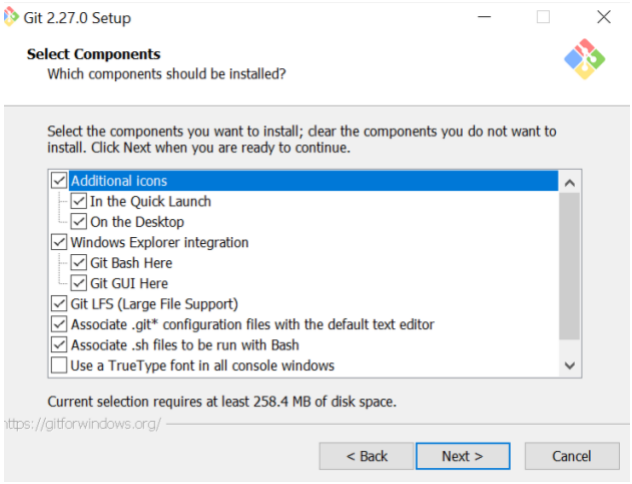
There are several useful online references for how to install git on your computer[14,15]. These websites have instructions on how to install Git on Windows, macOS, and Linux computers. In this guide we will focus on computers that run Windows and macOS. Although terminal output may vary slightly between Windows and macOS computers, all of the commands that you will enter are the same regardless of the operating system.

2.2.2.1 Windows

When using a Windows computer, we will install a program called "Git for Windows". This installs a Git along with a program called "Git Bash" that will enable us to interact effectively with Git. To do this, go to https://git-scm.com/downloads and click on the download icon. You should be brought to a new page and the download should automatically begin. If not, you can manually download it on the web site.

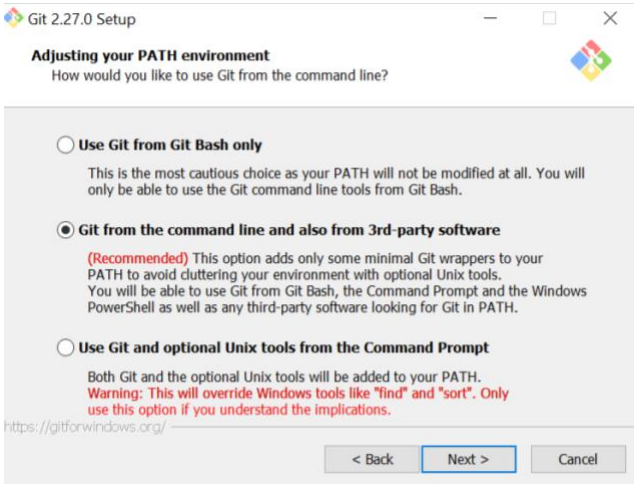
Open the download to begin the installation process. Most of the default settings during the installation are already set for what we want, but here are some recommended adjustments:

1. You want to add an icon to both the Quick Launch and Desktop locations on your computer for easy access to Git Bash.

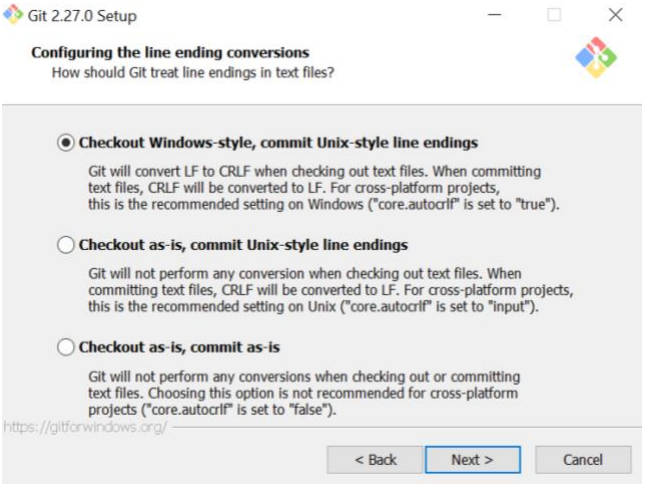


2. You want to select the "Git from the command line and also from 3rd-party software" option when it appears.

[14] https://git-scm.com/book/en/v2/Getting-Started-Installing-Git
[15] https://www.atlassian.com/git/tutorials/install-git



3. If you work with multiple operating systems, or if anyone on your team owns a non-Windows computer, you will want to select the "Checkout as-is, commit Unix-style line endings" option when it appears. This will make it so that all of your files in the repository have the same type of file ending.



After the installation is complete, you will have a shortcut on your Desktop called "Git Bash". This will provide us with the terminal-style interface for Git. From here on, whenever we refer to the terminal, go ahead and use Git Bash. Jump ahead now to Section 2.2.2.3 to finish setting up Git. If you'd like another command-line interface for Git, a tool that you might find helpful is Windows PowerShell[16], but it is not required to complete this portion of the tutorial.

[16] https://docs.microsoft.com/en-us/windows-server/administration/windows-commands/powershell

2.2.2.2 MacOS

Go to https://git-scm.com/downloads and click on the download icon. This will pull up a web page with several installation methods. Although you can use any of the methods, the simplest two approaches are to either use XCode or Homebrew. Both work well and are described below.

2.2.2.2.1 Installing Git with XCode

Open your terminal app and type:

```
git --version
```

You may be surprised to find that if you have XCode then you might already have Git, since it comes with XCode. If you already have Git then typing the above command in the terminal will tell you which version you have, and you're good to go! If you do not have Git, your computer will prompt you to install it. You can navigate the installation without actually installing XCode, but if you have a hard time navigating the installation without installing XCode, go ahead and allow your computer to install it for simplicity.

2.2.2.2.2 Installing Git with Homebrew

If you decide to use the homebrew installation, make sure to install homebrew first[17]. After homebrew is installed open your terminal and type:

```
brew install git
```

This will install Git for you.

2.2.2.3 Setting Up Git Once It's Installed

Regardless of which method you used to install Git, to verify that you now have Git on your computer, open the terminal and type:

```
git --version
```

Your computer should respond by outputting:

```
git version <some numbers>
```

Congratulations! You have successfully installed Git on your computer. Now it's time to tell Git your name and email address. This information is important because every time you do something with Git your name and contact information will be associated with it, and this will make working in groups easier for you and your team. To tell Git your name, type:

```
git config --global user.name "example name"
```

[17] https://brew.sh

where you put your name in the quotes. Then, to tell Git your email address, type:

```
git config --global user.email "example@email.com"
```

Replace the example email with your own email in quotes.

2.2.2.4 Navigating the Terminal

The terminal lets you navigate your computer using only your keyboard. As you become accustomed to using your computer this way it will become a very efficient tool. This subsection provides some of the basic navigation commands so that we can begin using Git.
When you first open the terminal, you will be inside of your home directory. To see what directory you're in use the "present working directory" command. Type:

```
pwd
```

Now that we know what directory we're in, we can use the "list" command to tell us what subdirectories are in the current directory so that we can navigate to any location on our computer. Type:

```
ls
```

This will provide you with a list of the directories within the current directory. To move into one of those folders use the "change directory" command. For example, if there was a folder called "Work" in the current directory and you wanted to move into it, type:

```
cd Work
```

If there is a space in the directory name, you'll want to wrap the directory name in parenthesis. Again, use the "list" command to tell you what directories and files are within this new directory, and repeat until you have navigated to where you want to go. If you get lost, you can use the following two commands:

`cd`	(this command, if left empty, will return you to the home directory)
`cd ..`	(this command, if followed by two periods, will take you up one directory)

The best way to understand these commands is to practice navigating around your computer using them. There are plenty of other commands out there that you can learn if you want but are not necessary for this Git tutorial.

2.2.2.5 Creating a Repository

Let's say that you have some files in a folder on your computer that you want to turn into a Git repository. This can be done one of two ways:

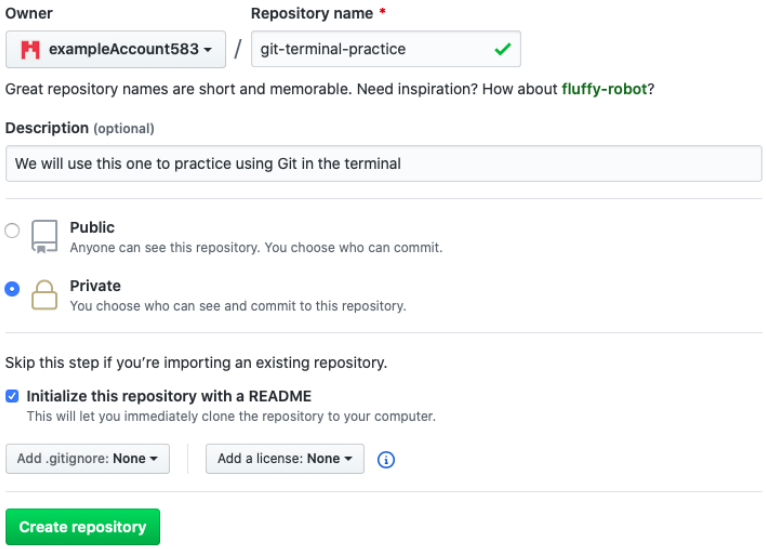
1) Create the repository on your computer and then push it to an online service

2) Create the repository online and then clone it into the folder on your computer

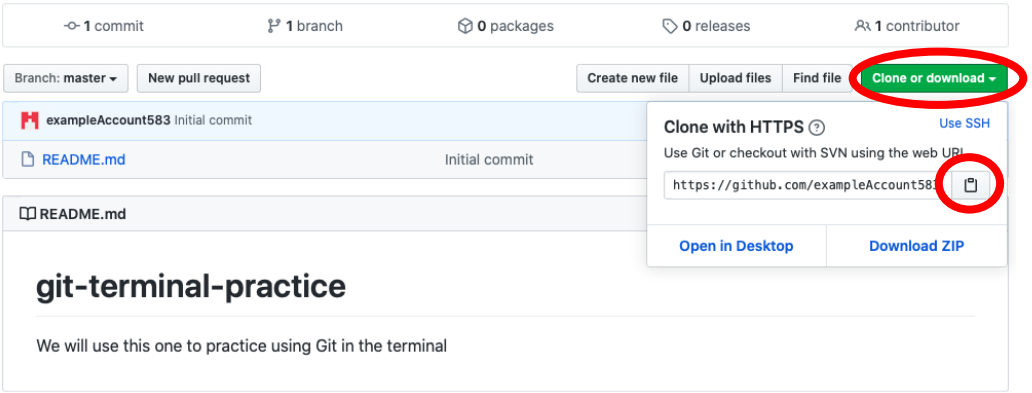
The second method is the most straightforward for us to use here. We will use GitHub, but the principle is exactly the same for GitLab: create an empty repository and clone it into the folder on your computer that contains all of your files.

2.2.2.6 Cloning a Repository

Go ahead and create an empty repository on either GitHub or GitLab and give it the name "git-terminal-practice" or something similar to that. For your reference, here is the repository creation information for the repository used in this tutorial:



To clone a repository on GitHub, click on the green "Clone or download" icon and copy the link provided. On GitLab, click on the blue "Clone" icon. In both cases, make sure you copy the HTTPS link. On GitHub, this looks like:



To clone the repository, use the terminal to navigate to a folder that you want to put the repository into. Cloning the repository will create a folder named after the repository in whatever directory you are in. To clone, use the "git clone" command:

```
git clone https://github.com/exampleAccount583/git-terminal-practice.git
```

If this is your first time using Git in the terminal, you will be prompted for your account's username and password. If you have used Git before in the terminal with this account (exampleAccount583 in this instance), then the repository will be cloned without the need for a username and password. Lastly, if you have used Git with a different GitHub account from your terminal, your terminal might get confused and not let you clone the repository because the username and password that you have in the system does not match the one to clone the repository from this new account. Below is an example of how to clone a repository for the first time, where commands that I typed are highlighted in green and the rest is the terminal output:

```
markanderson@MarkAndComputer ~ % pwd
/Users/markanderson
markanderson@MarkAndComputer ~ % cd Desktop
markanderson@MarkAndComputer Desktop % git clone https://github.com/example
Account583/git-terminal-practice.git
Cloning into 'git-terminal-practice'...
Username for 'https://github.com': exampleAccount583
Password for 'https://exampleAccount583@github.com': Password goes here
remote: Enumerating objects: 3, done.
remote: Counting objects: 100% (3/3), done.
remote: Compressing objects: 100% (2/2), done.
remote: Total 3 (delta 0), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
markanderson@MarkAndComputer Desktop %
```

The repository is now cloned! You will generally want to have a better place to store your repositories than your Desktop, so consider creating a designated "projects" or "repositories" folder somewhere on your computer. If you open the folder you will see that anything that was on the online repository has now been downloaded to your computer.

2.2.2.7 Uploading Files

Let's say that you have two files that you now want to add to this repository:

some_math.m
important-notes.txt

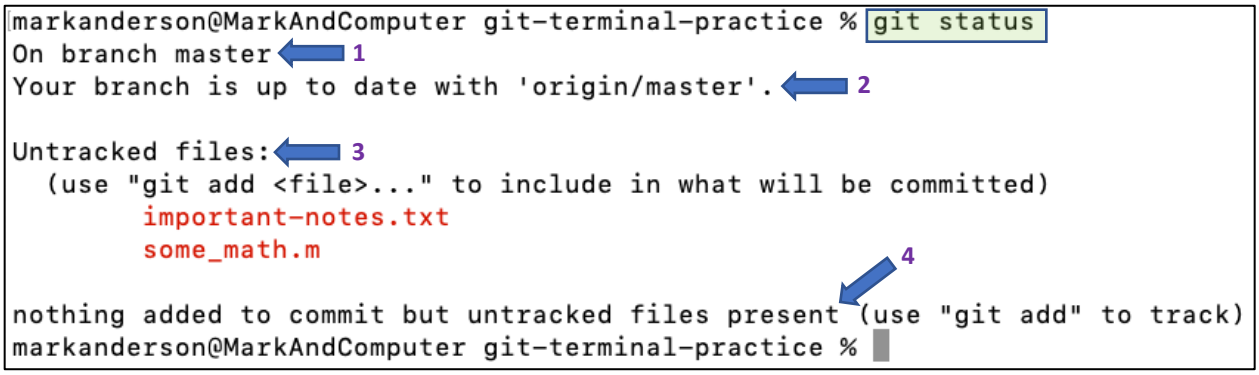
If you made these files before cloning the repository to your computer then go ahead and paste them into the repository folder. If you made them after cloning the repository, you can just save them directly into the folder. However you went about it, you now have two files in your "git-terminal-practice" repository folder and you want to make sure that Git keeps track of them. First, use the "cd" command to move into your repository:

```
cd git-terminal-practice
```

Now if you use the “ls” command the terminal will show you the repository’s contents.

2.2.2.7.1 The “git status” Command

One of the first commands to learn is the “git status” command. This command will tell you the current state of your repository:



1. You are on the main branch of the repository, more on branches in Sections 1.2.4.
2. The branch you are on (the master branch) is currently up to date with the online repository. This means that you have not made any new commits that the online repository isn’t aware of, and vice versa.
3. Git sees that there are new files in the folder and is telling you that it sees them but is not yet tracking them. Often you will actually have files in your repository folder that you don’t want Git keeping track of, so this is a useful feature.
4. Git is again telling you that there are untracked files, and to use the “git add” command to add them to the staging area so we can commit them to the repository.

2.2.2.7.2 Staging

The staging area is a place to say, “Let’s bundle together all the files that I want to commit to Git on my next commit”. You can add several files, or just one, all at once or at different times. The next time that you commit the files to Git, you will commit all of the files that are currently in the staging area. To add a file to the staging area, use the “git add” command:

```
git add some_math.m
```

If you want to add all of the files in your project at once, then use a period instead of a specific file name. Type:

```
git add .
```

Those files are now set aside and ready to be committed. Here is an example:

```
markanderson@MarkAndComputer git-terminal-practice % git add some_math.m
markanderson@MarkAndComputer git-terminal-practice % git add important-notes.txt
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is up to date with 'origin/master'.

Changes to be committed: <- 1
  (use "git restore --staged <file>..." to unstage) <- 2
        new file:   important-notes.txt
        new file:   some_math.m

markanderson@MarkAndComputer git-terminal-practice %
```

1. You have made changes to the repository (added files) and staged the files that have been changed (being added technically counts as being changed)
2. Git is telling you how to remove the files from the staging area. This is useful if you decide you're not quite ready to commit a file. If you wanted to remove important-notes.txt from the staging area (we won't actually do this), then you would type:

`git restore --staged important-notes.txt`

2.2.2.7.3 Committing

To commit all of the files that are in your staging area, use the "git commit" command. This will tell Git to take a snapshot of your files in the staging area as they are at this moment and save that snapshot forever. This will let you see how your files change over time and even restore old versions if you need them later.

`git commit -m "type a message here"`

The commit message[18] denoted by the text within the quotes is very important because in that short message you describe the changes that you made to the files being committed, which will be useful to you in the future as you look at the file histories. Below is an example commit:

```
                                                    1
markanderson@MarkAndComputer git-terminal-practice % git commit -m "Added math and notes files"
[master ad2d475] Added math and notes files
 2 files changed, 21 insertions(+) <- 2
 create mode 100644 important-notes.txt
 create mode 100644 some_math.m
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit. <- 3
  (use "git push" to publish your local commits) <- 4

nothing to commit, working tree clean <- 5
markanderson@MarkAndComputer git-terminal-practice %
```

[18] The "-m" lets you add the message in the same line as the commit command. Otherwise it will open a window that prompts you for a commit message and that window that can be quite confusing. If you find yourself in this window press "ctrl + X" on your keyboard. If asked whether you want to save press "N".

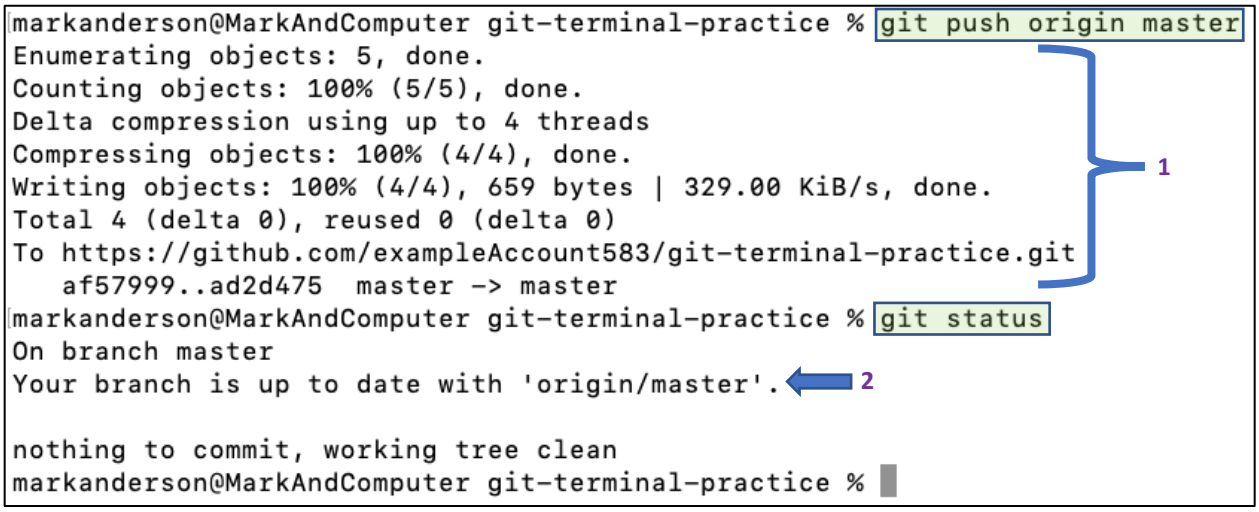
1. A unique number associated with the commit.
2. How many changes were made (adding new files will count every line of each file as a change).
3. The repository on your computer (the local repository) is ahead of the online repository (the remote repository or "origin/master") by one commit. In other words, you have made changes to the repository, but only on your computer. If you want to sync those changes with the online repository, you will need to "push" the changes, discussed in the next section.
4. Telling you how to "push" the local repository to the online repository
5. Git is unaware of any new changes to your files since your last commit.

2.2.2.7.4 Pushing

Now that the files are committed, they are forever remembered by Git in whatever state they were in at the time of the commit. These changes are saved locally on your computer, so if you want them to be permanently saved online then you need to push the commits to the online repository. This is done by typing:

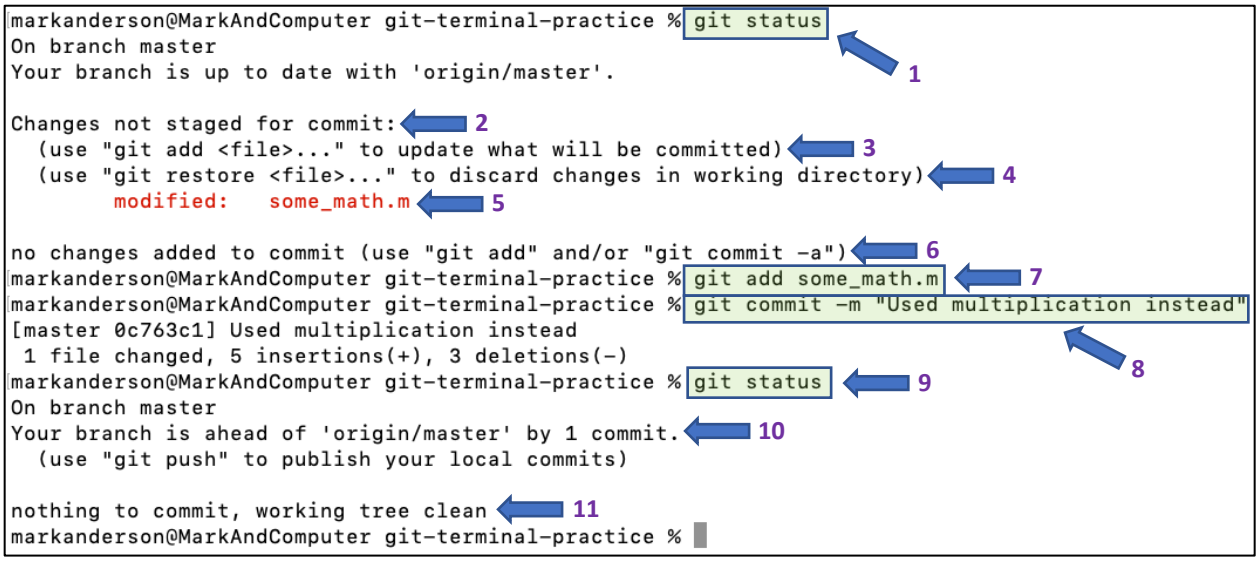
```
git push origin master
```

Now your changes are synced with the online repository. When you do this, all commits that haven't been uploaded to the repository will be uploaded, so you don't need to do this every time you make a commit, but rather at least once a day should be sufficient, or whenever you decide is appropriate. Below is an example push:



1. Git is uploading the files to the remote repository
2. Your local repository and the remote repository are now in sync

Let's say that you have made some changes to some_math.m and would like to commit those changes to Git. Go ahead and commit those to Git as discussed above. Here is an example:



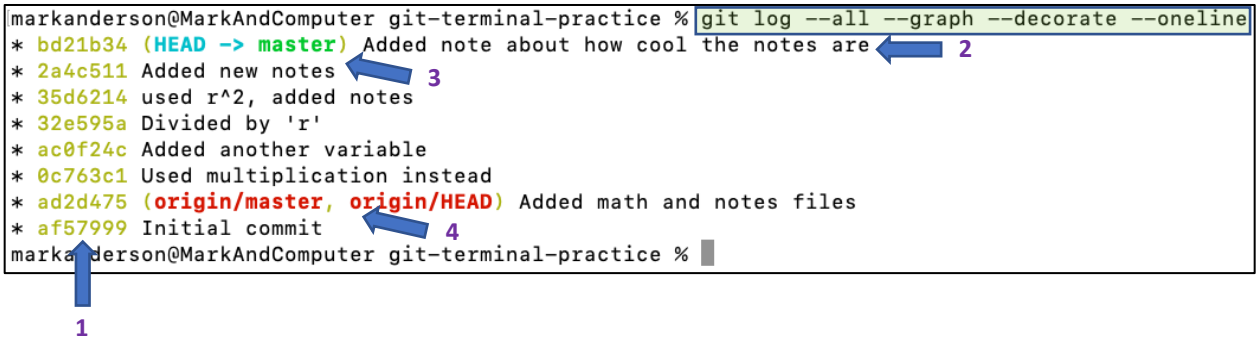
1. Begin by checking the repository status
2. There are changes in the repository, but they are not staged. This means that if you do a commit right now, those changes will not actually be committed.
3. A reminder for how to add files to the staging area
4. This tells us how to restore a file to how it looked on the last commit. This is useful if we have made changes, decided we didn't like them, and then wanted to revert the file to the way it was on the last commit.
5. The file that has been modified in our repository is some_math.m. There could be many modified files, but in this example we just have the one.
6. A reminder for how to add files to the staging area. Also included is a shortcut for how to add a file to the staging area and commit it in one line.
7. Adding some_math.m to the staging area
8. Committing all the files in the staging area (just some_math.m in this case) to Git.
9. Checking the repository status after the commit to make sure it looks as expected.
10. We are now ahead of the remote repository again by one commit. If we want, we can push our changes to the remote repository now. We can also keep making changes and commits and decide to push those changes to the remote repository later. When we do push to the remote repository, all of the commits we have made will be added to the remote repository so we can access any versions of our files that were ever committed.
11. There are no more changes in the repository since the last commit

2.2.2.7.5 Comparing Past Versions of Your Repository

To view the history of your repository, use the "git log" command. Using this command will show each commit and information about it, like it's commit message. You can also add options to the "git log" command to make the output more readable. I recommend using:

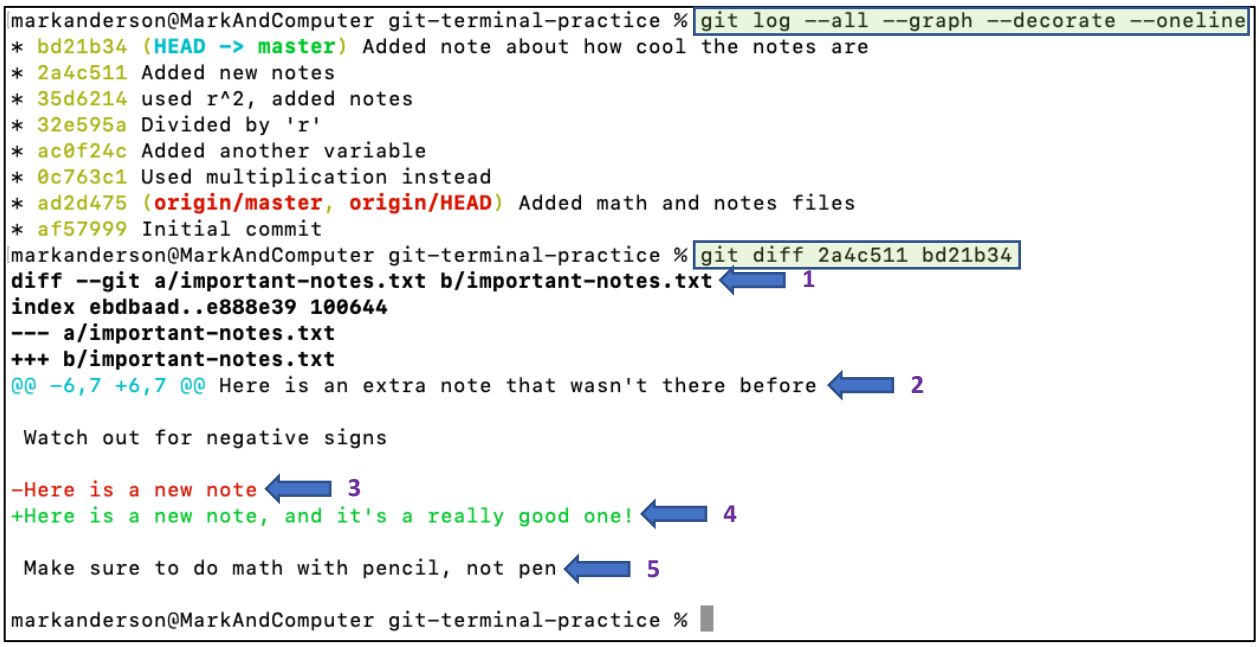
```
git log --all --graph --decorate --oneline
```

This will output each commit, along with its ID number and commit message, line by line. Because this is a longer command, I recommend making it into an alias. My personal alias for this command is “hist” so I can just type “git hist” and get the proper output. Here is an example of what this output may look like:



1. The commit IDs listed with the newest at the top
2. The associated commit messages
3. The (HEAD -> master) tells you that you are currently on the master branch in your repository.
4. The (origin/master, origin/HEAD) tells you what commit the remote repository is on. If you go to the remote repository now it will have the files as they were back immediately after that commit.

We can use the commit ID numbers to look at the differences between files from one commit to another. To do this we use the “git diff” command, followed by the two ID numbers we would like to compare. Here is an example:



1. It looks like important-notes.txt was changed between these two commits
2. Git will show us some context for the changes in the file, here is the beginning of the file.
3. Here is a difference, this is what the first commit says on this line. The first commit is denoted by the first ID number after the “git diff” command
4. Here is what the second commit says on that same line. Even though no text was actually deleted, Git considers a change to a line as though the line were deleted completely and then replaced by the new text.
5. End of context in the file.

And there you have it! You can now use Git in the terminal!

3 Working in a Group

Git is a useful tool for personal use, but it truly shines when you are collaborating with others. This chapter is applicable to anyone working in a team on a common project. Many of the principles like branching and merging are useful for solo users too, and reading this chapter will be beneficial to any Git user.

3.1 More Detailed Workflow Description

Let's talk about what a typical workflow or a typical day might look like with your team using Git. Let's say that you have three team members: Sarah, Alexa, and Josh. You are working together on a code that will analyze the sound emitted by an airplane.

- Sarah creates the project on a remote repository and calls it "airplane-sound". She gives access to Alexa and Josh.
- They begin putting files and code into the repository and quickly get a very simple version of the code to work.
- Josh recognizes that he can add a new function to the code that will make analyzing the sound easier. However, he knows that they want to always have a version of their code that works in case they ever want to produce some quick results for a meeting with their boss. So, he creates a branch of the repository and calls the branch "sound-simplifier-function". This puts him in an "alternate universe" of the repository, where he can make any changes he wants and it won't affect the main, or "master" branch.
- Once Josh completes his new function and makes sure that it works properly, he's ready to merge his branch into the master branch. After getting the go-ahead from Sarah and Alexa, he merges his improvements into the master branch, and they all have access to this new and improved version of the code.
- Alexa has also been making some changes in a branch called "improving-data-structure", where she is improving the way that the code structures the data. When she's done making her changes and is ready to merge her improvements into the master branch, she gets an error that says there are "merge conflicts". It turns out that when Josh made his improvements that he edited some of the files that Alexa was also editing.
- Alexa takes a look at the conflicting files that she and Josh have both edited. Git allows her to choose which version of the changes to keep and which to discard. If the changes are small and inconsequential, she can quickly decide which versions of the changes to keep. If the conflicts are big then they should meet as a team and discuss which changes to keep, and either Josh or Alexa can adjust their code to work with the newest improvements. In this case, they decide to keep Alexa's improvements, and Josh then adjusts his code to work with hers.
- After all improvements are made and reconciled, they push all the functioning improvements to the master branch.
- Sarah, Alexa, and Josh continue to work like this for months as they develop better and better versions of their airplane-sound repository, all the while knowing that they have a functioning version of their code on the master branch that they can use at any time. It's

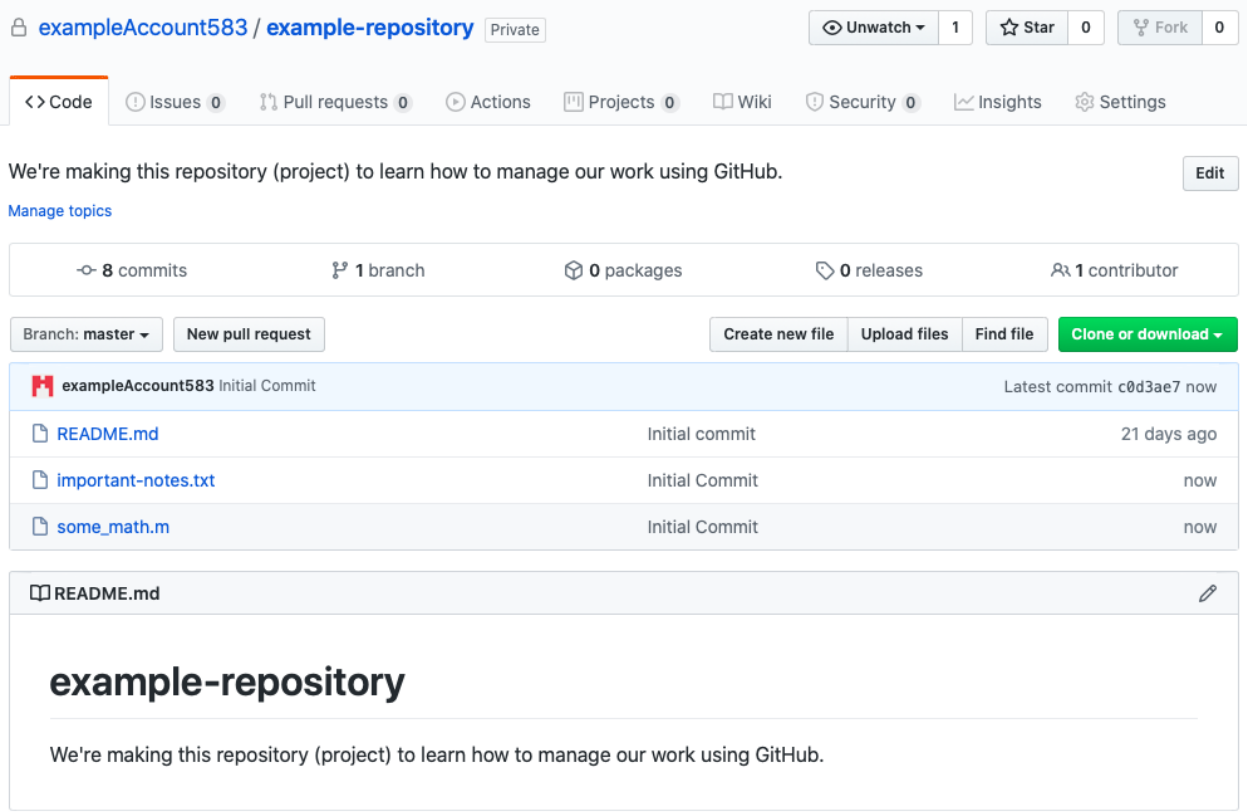
also nice to know that Git keeps a history of their files that they can look at and revert to any time they need to.

The rest of this chapter will deal with creating branches, merging those branches into the master branch, and resolving any merge conflicts that arise. This will be done using the same four Git providers used in the previous chapter: GitHub, GitLab, GitKraken, and the terminal.

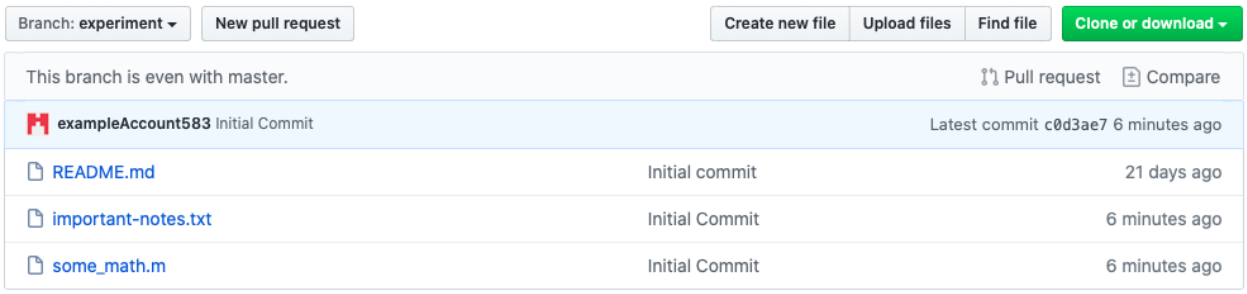
3.2 GitHub

3.2.1 Creating Branches

Let's say that your repository looks something like this:

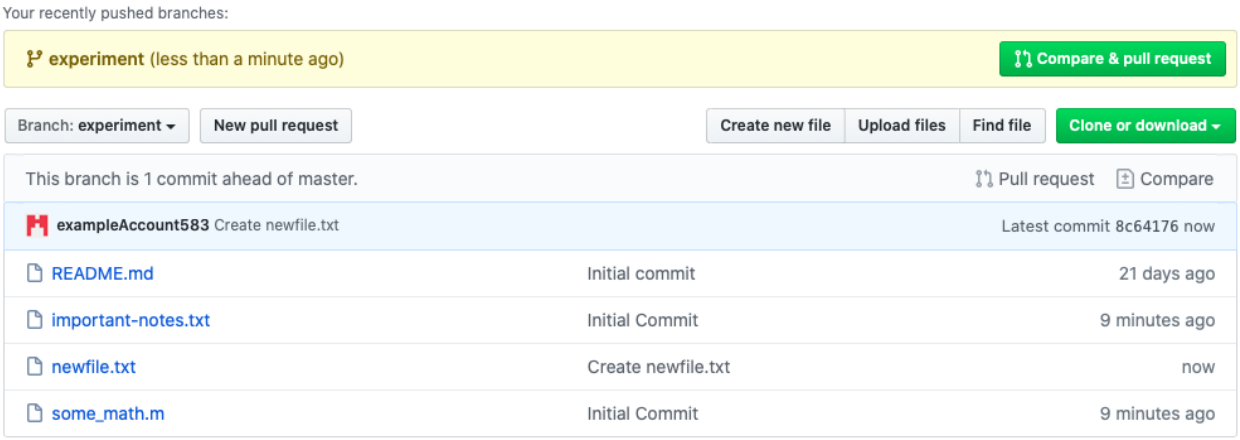


Now let's say that you want to make some experimental changes to some_math.m. You will want to create a branch of the repository so that you keep the master branch in good health while you make your changes. To do this, click on the "Branch: master" icon toward the left side of the page, type the name of the branch you want to create (we will call ours "experiment"), and hit enter. You will see that GitHub automatically puts you into the new branch:

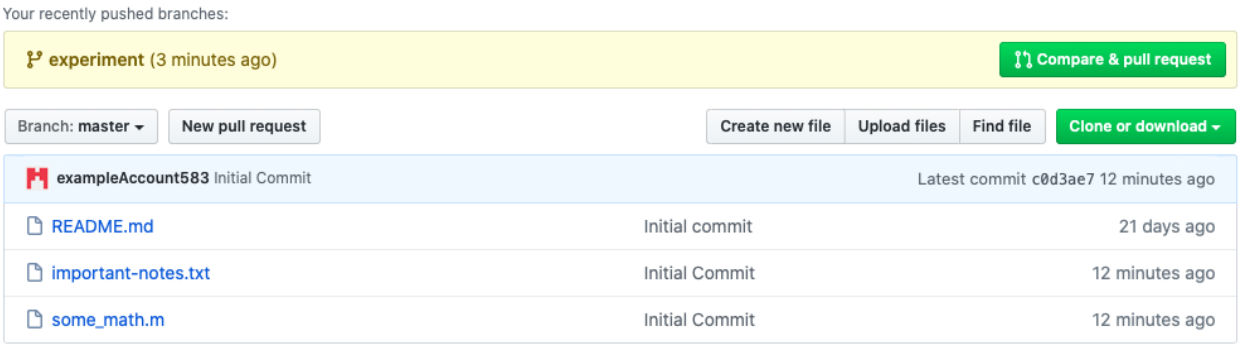


You are now in an "alternate universe". Any changes that you make here will be saved to this branch of the repository, but not to the master branch. To demonstrate this, create a new file

called “newfile.txt” in the current branch. You can do this either by creating it online or by uploading it from your computer.



Go ahead and ignore that yellow box at the top about pushing and pulling branches for now, we’ll get to that later. You have successfully created a new file in the experiment branch. Now, to toggle between branches, click on the “Branch: experiment” icon on the left side of the page. Return to the master branch.[19]



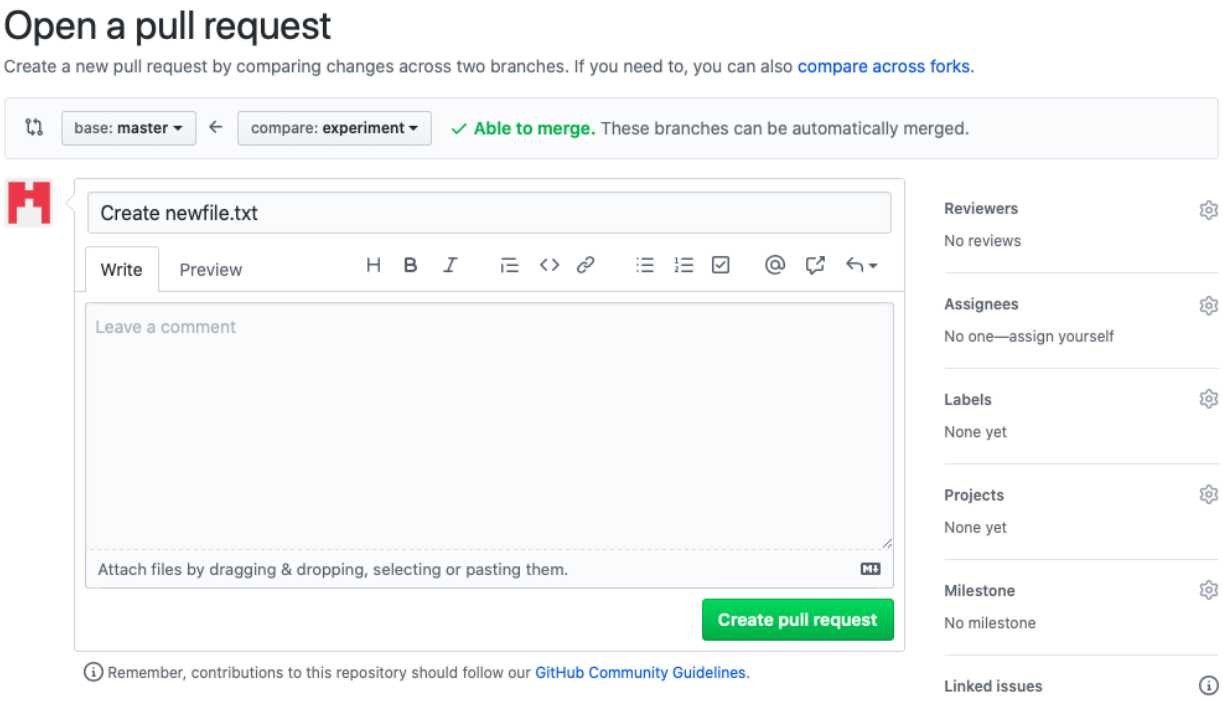
The changes you made in the experiment branch are not seen in the master branch because the experiment branch is an alternate universe. You can do literally anything in the experiment branch, and it will have exactly zero impact on the master branch. Feel free to experiment away in another branch, fail, succeed, and improve your code until you’re ready to pull those changes into the master branch.

3.2.2 Merging Branches

Let’s say that we like “newfile.txt” and want to bring it into the master branch. Click on the green “Compare and pull request” icon at the top of either the master or experiment branch (it appears

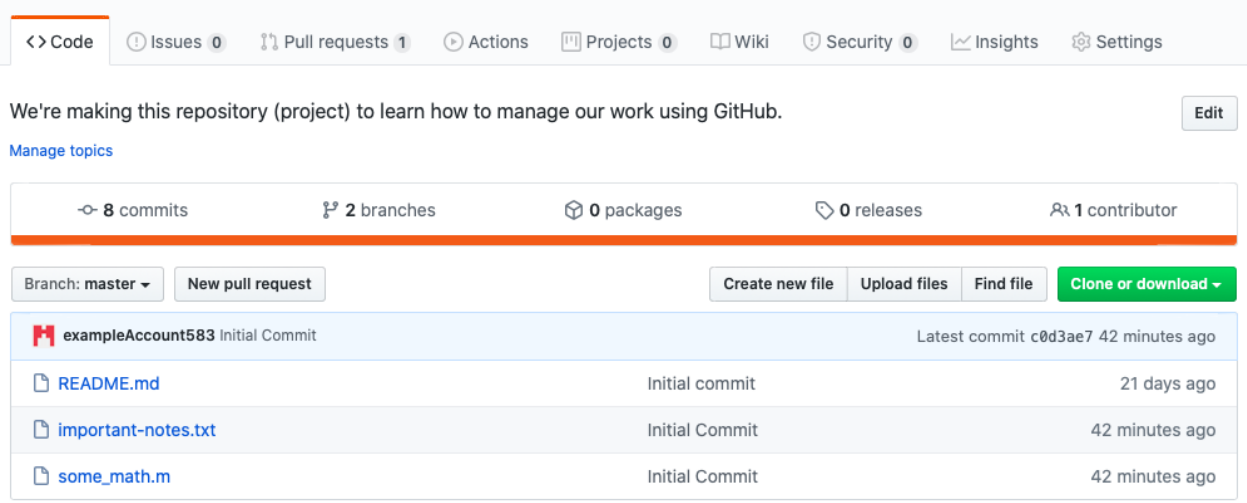
[19] Sometimes you’ll find that GitHub defaults to putting you into the master branch. It’s a good idea to check back often and make sure that you are in the correct branch.

in both). This is basically saying “Hey, you made some changes to another branch, do you want to pull those into the master branch?” After clicking on the icon, you will see the following:

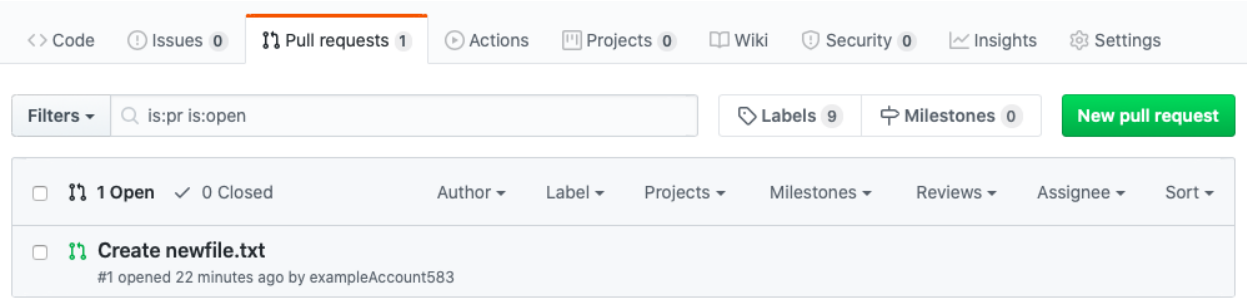


Look at the top where it says “base: master”. This means that you’re going to be merging changes into the master branch. The “compare: experiment” icon means that you’re going to be merging those changes from the experiment branch. Also notice the text that says, “Able to merge.” This means that the changes in the experiment branch can be merged into the master branch without any conflicts (which makes sense, all we did was add a file). We’ll cover the case where there are conflicts later. Leave a comment about the changes and then click “Create pull request”.

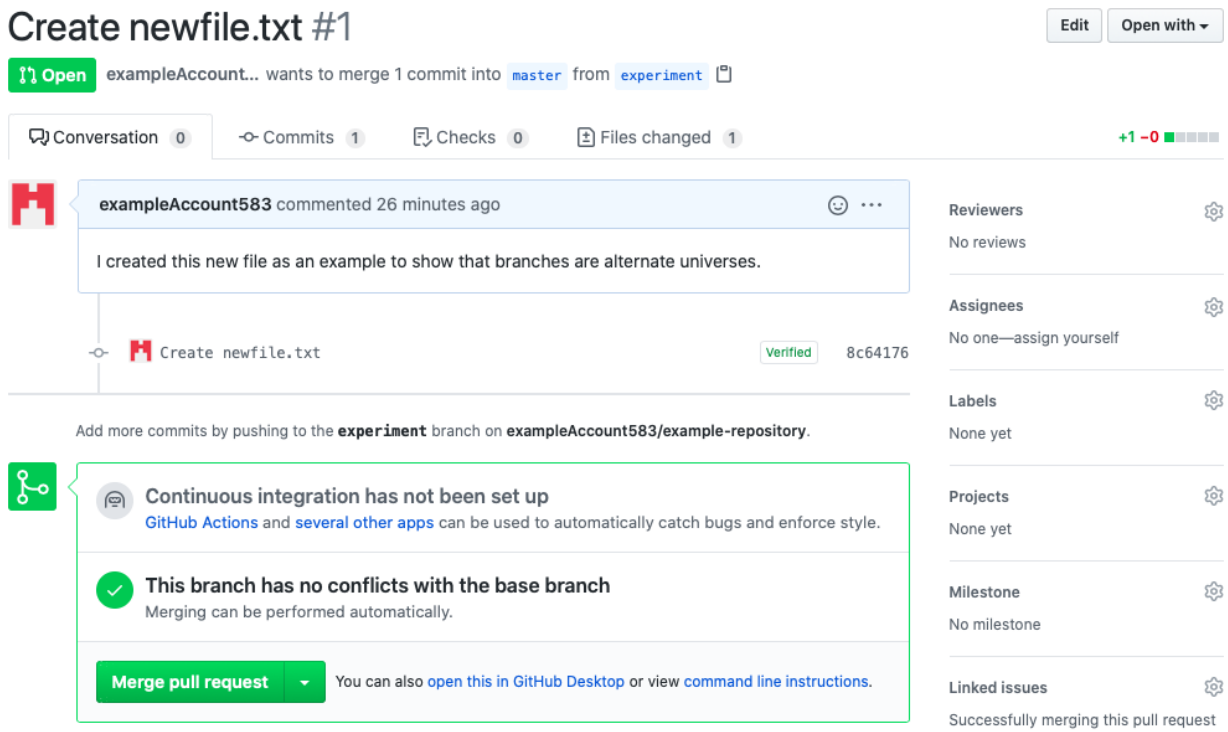
You’ll be brought to a frightening-looking page. We’re going to return here, but for now go ahead and navigate to the main repository page on the master branch. You’ll see the following:



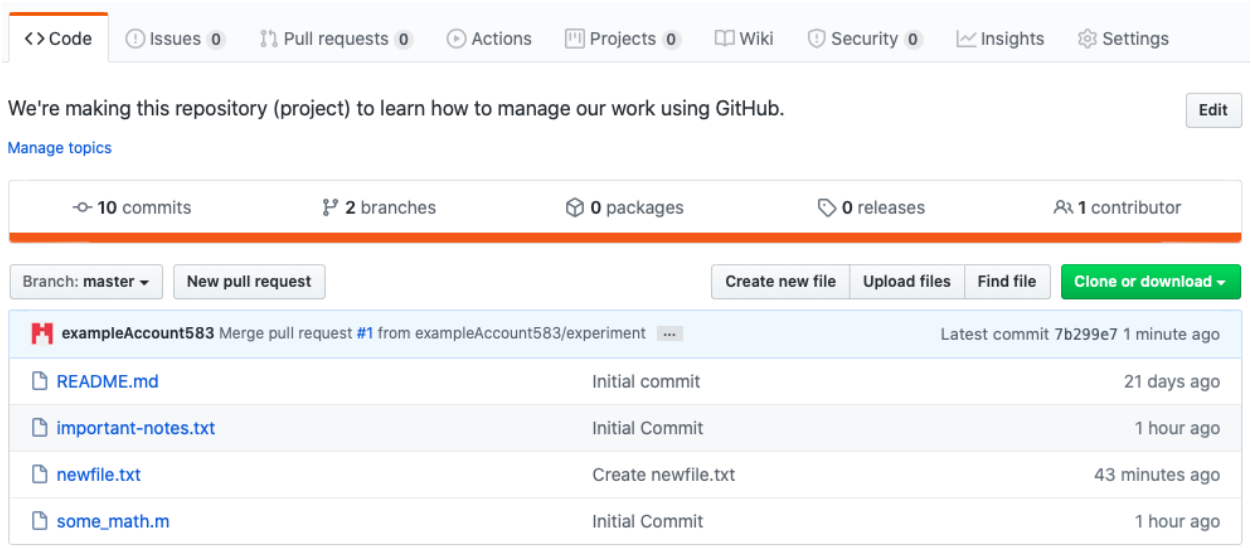
Notice the top banner, where it says, “Pull requests”. There is now a little “1” next to it. This means that someone (in this case yourself) has created changes in another branch and would like to merge it into the master branch. In other words, they are requesting you to pull those changes into the master branch. Click on the “Pull requests” icon and you’ll see the following:



If you ever want to create a pull request but don’t have that yellow banner that we used earlier, this is where you come to create a pull request. As you use bigger projects with more branches, this is where you’ll find a list of all the pull requests. Again, a pull request is someone else (in this case yourself) saying that they are ready to send changes to the master branch and wants whoever has access/permissions to the master branch to pull those changes in. Click on the request and you’ll be brought back to that frightening-looking page from earlier:



The green check mark indicates that there are no conflicts, so go ahead and click on the “Merge pull request” icon and then “Confirm merge”. After the merge is complete, navigate back to the main repository page of the master branch and you’ll see:



The experiment branch was successfully merged with the master branch. If you are done with the experiment branch, you can delete it by clicking on the “2 branches” icon and then deleting the branch from within that page.

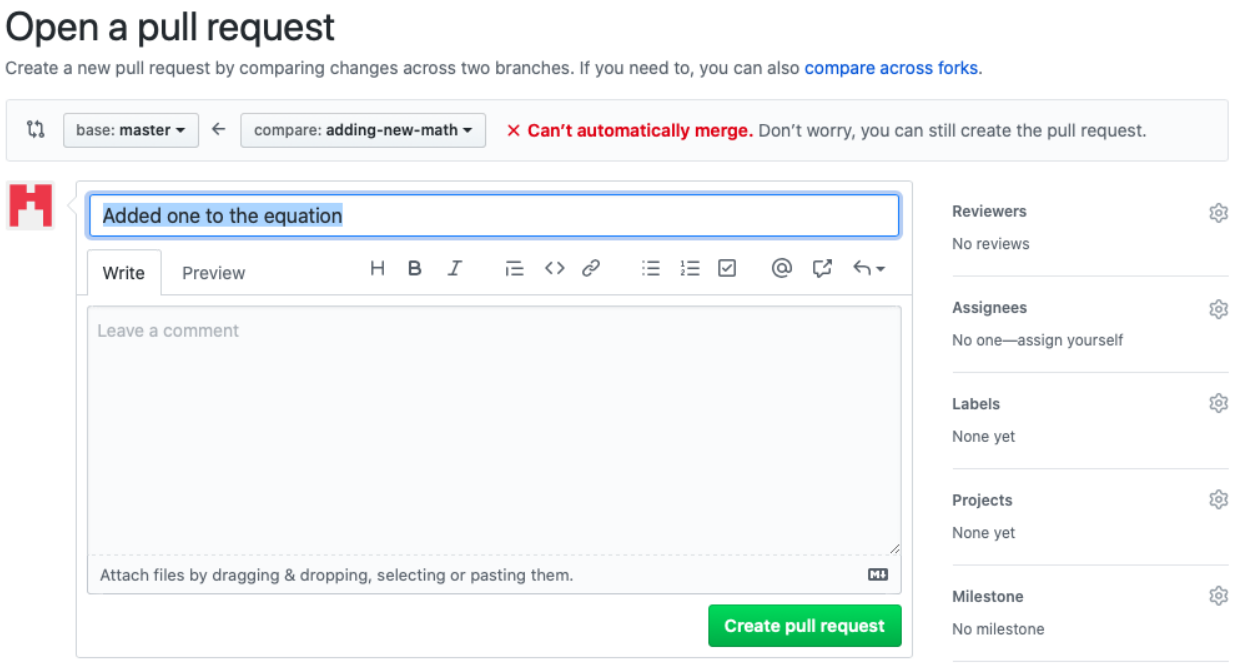
3.2.2.1 Resolving Merge Conflicts

Now let's create a new branch called "adding-new-math". Within this new branch, open up some_math.m. Mine looks like this:

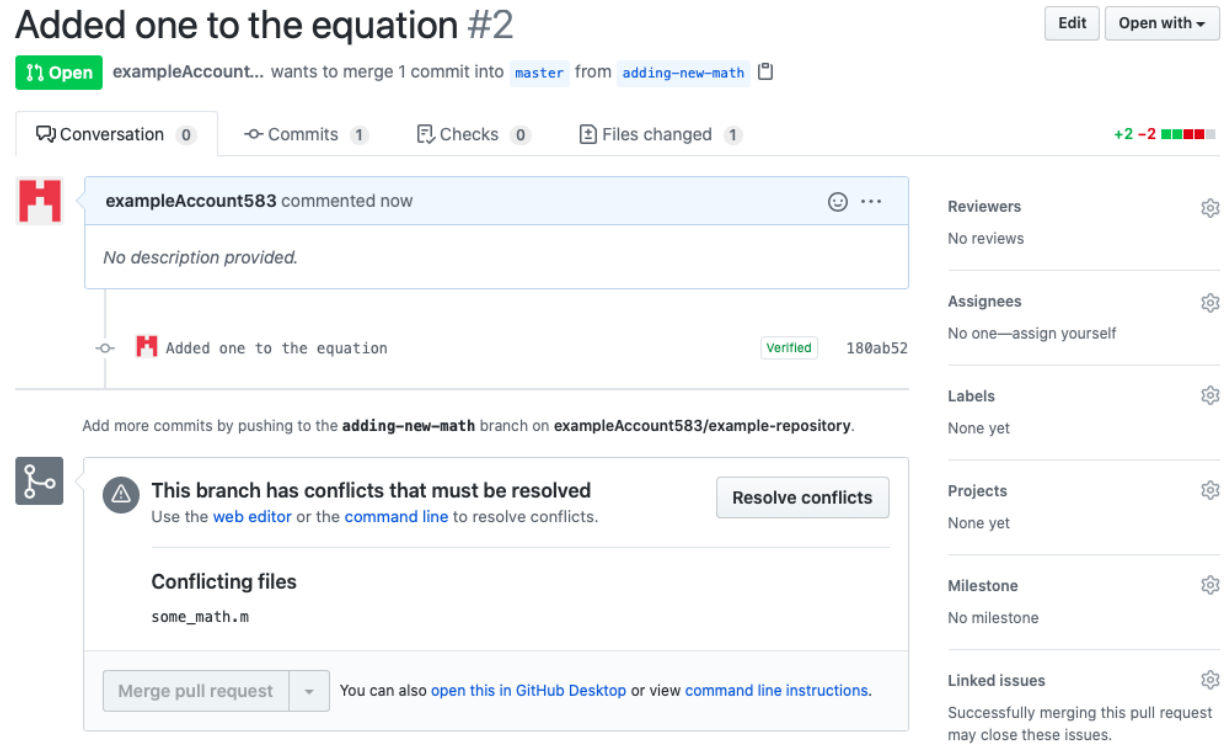
Now let's say that we change line 10 to say: z = x * y / r^2 + 1.

Now that this is changed in the adding-new-math branch, let's return back to the master branch and make a similar change. Return to the master branch and open some_math.m. Again, notice how the changes we made in the alternate universe of "adding-new-math" are not reflected in the master branch. I will now change line 10 in this file to read: z = x * y / r^2 + 2.

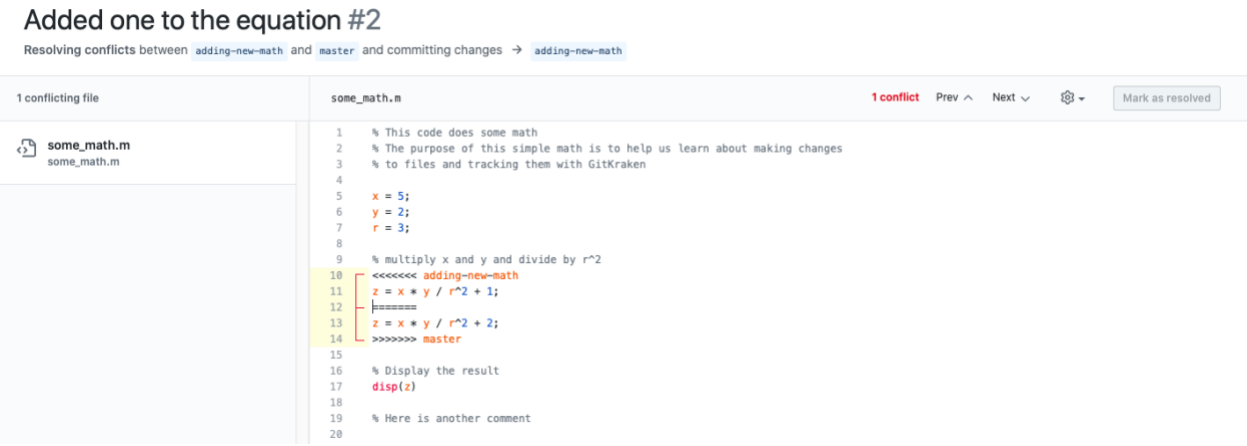
Both the master and adding-new-math branches have conflicting changes. Now if we merge the two branches, we will have to decide which change to keep. To get started on this merge, navigate again to the main repository page (on either branch) and click on the "Compare & pull request" icon. This time we see:



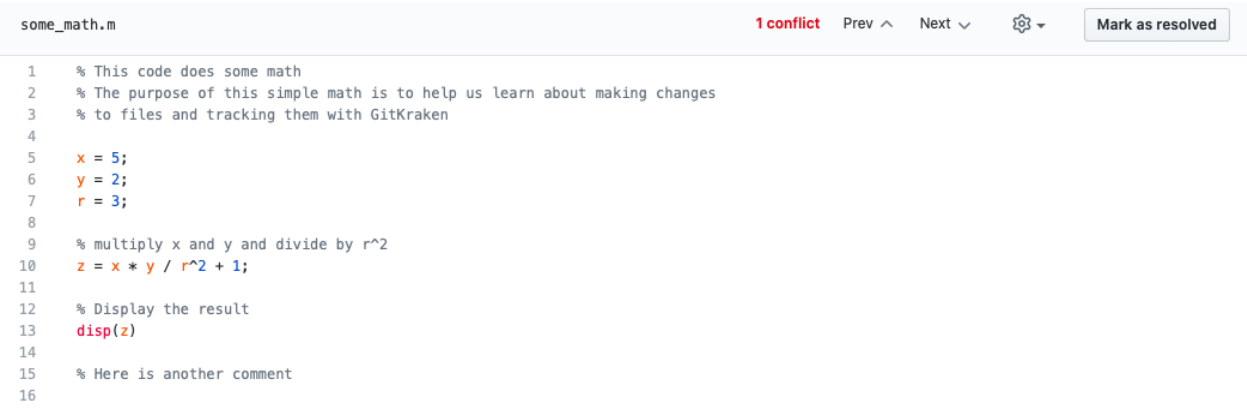
Notice the red text that says, "Can't automatically merge". This is telling us that later we'll need to do a manual merge when the changes are pulled into the master branch. Click on "Create pull request" and this time you'll see the following:



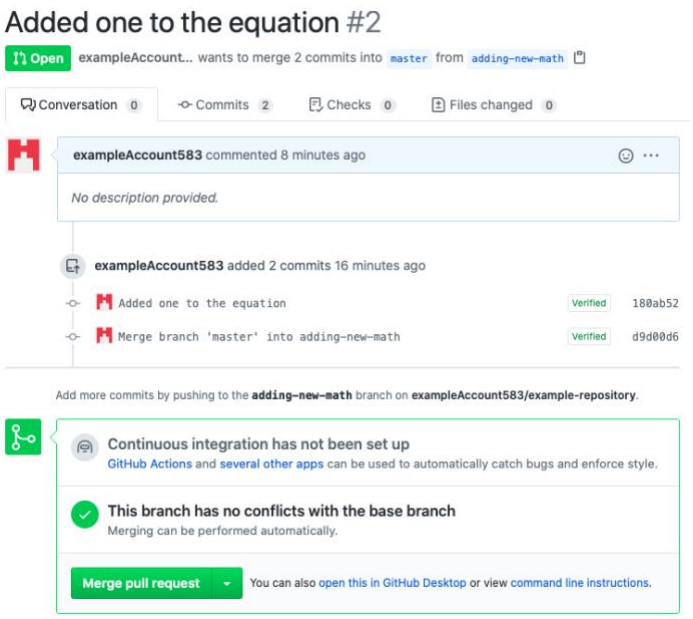
Notice the warning that says there are conflicts that must be resolved. To resolve them, click on the “Resolve conflicts” icon. You’ll see something like this:



To resolve the conflict, you will need to manually delete the conflict markers (all the arrows and equal signs) along with the changes that you don’t want to keep. For example, if we decided that we wanted the “+ 1” from the adding-new-math branch rather than the “+ 2” from the master branch the final result would look like:



If you had multiple merge conflicts you would use the previous and next buttons at the top of the window to navigate to the different conflicts, manually choosing which changes to incorporate for each conflict. When complete, click on “Mark as resolved” and then on the green “Commit merge” icon. You will be returned to the pull request page, where you will see that there are no more merge conflicts and your page looks something like this:



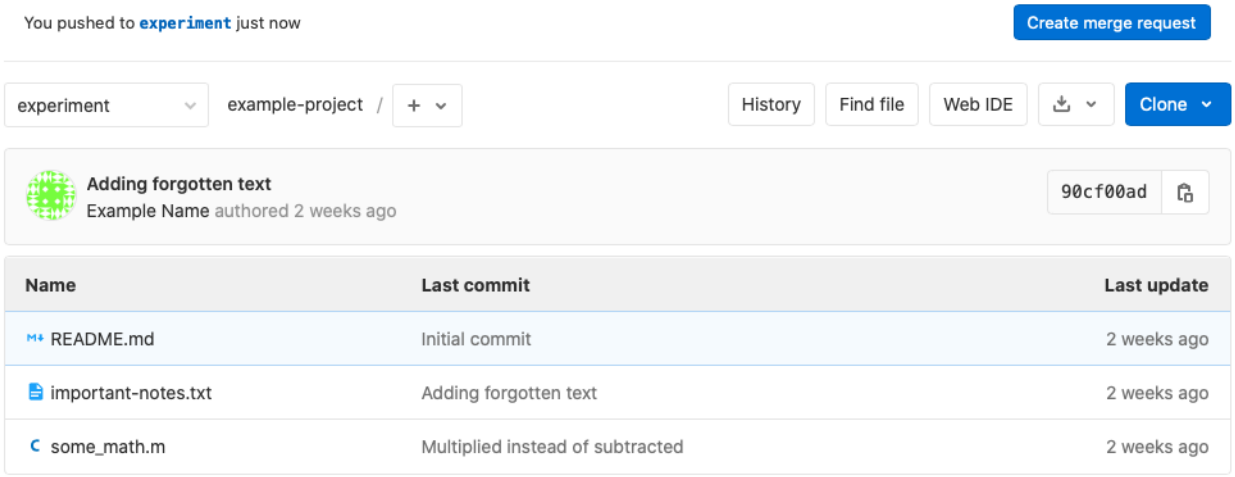
Click on “Merge pull request” and “Confirm merge”. Then, return to the main repository page of the master branch. You will see that any changes are now reflected in the master branch. To see a history of the file, click on it from the main repository and then click on the “History” icon. You will see that the changes are reflected in the history of the file.

Congratulations! You can now do teamwork in GitHub! There is much to learn, so feel free to explore around and look up even more tutorials about GitHub to refine your skills.

3.3 GitLab

3.3.1 Creating Branches

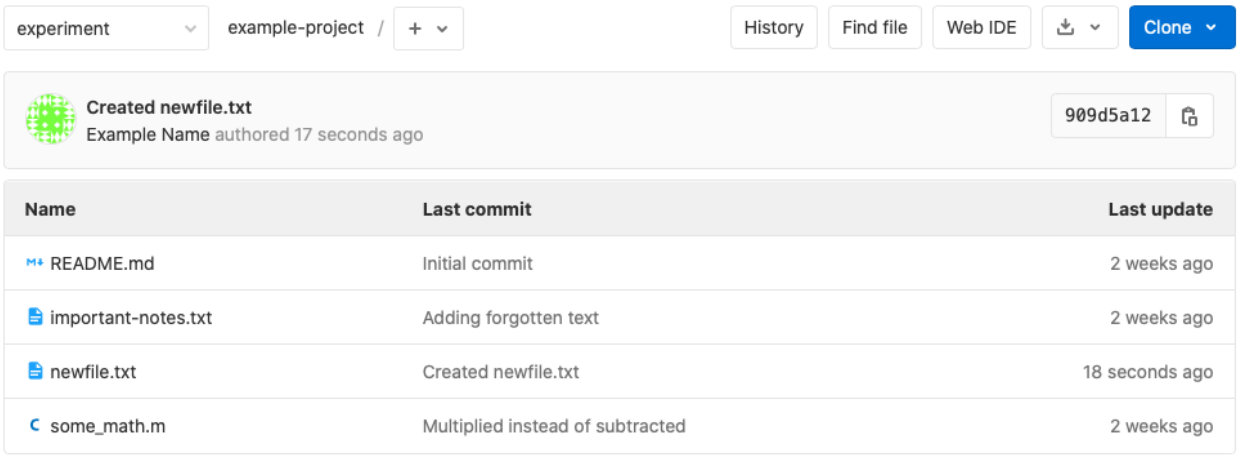
To create a branch, go to the main project page, click on the "+" drop-down box, and select "branch". You'll be brought to a page to give the branch a name, and for this example will use the name "experiment". You can also select which branch from which to create this new branch.[20] In this example, go ahead and select the master branch. Then click on "Create branch". You'll be redirected to the main project page for this new branch:



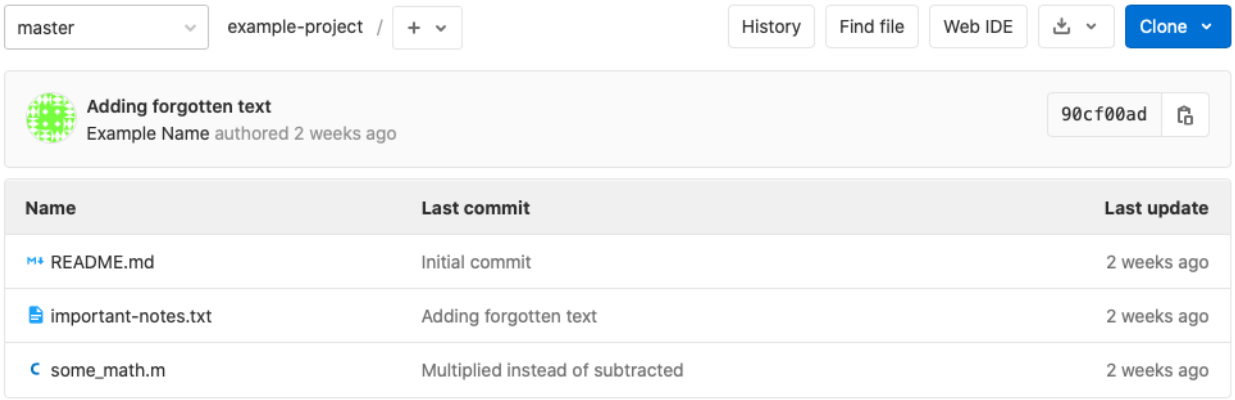
For now, ignore the top banner that talks about creating a merge request. You have successfully created an "alternate universe" of your project. Any changes that you make to this branch will have exactly zero impact on the master branch. This is a place where you can really test out some crazy ideas, mess everything up, and fix problems all with the knowledge that you have a safe, working version of your project in the master branch. If things don't work out in this branch the way you hoped, you can delete it, create a new branch, and move on to the next thing you'd like to try out.

To demonstrate this, go ahead and create a new file called "newfile.txt" in this experiment branch. You can do this either online or by uploading it from your computer. Once this is complete, the main project page of the experiment branch will reflect those changes:

[20] This would be useful if you have lots of branches. For big projects you may find yourself creating a branch off of another branch.



To toggle between branches, click on the branch name (in the upper left of the above screenshot) and you'll see a list of your branches.[21] Toggle over to the master branch and notice that it does not have that new file in it[22]:



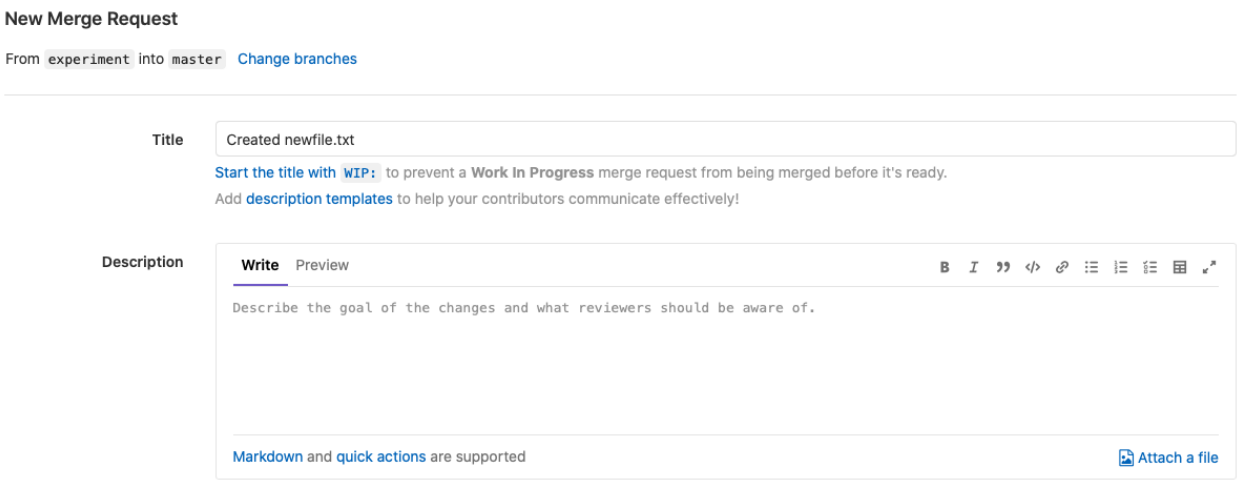
Hopefully this adds to your confidence that the different branch truly is its own "alternate universe" for you to use to experiment in.

3.3.2 Merging Branches

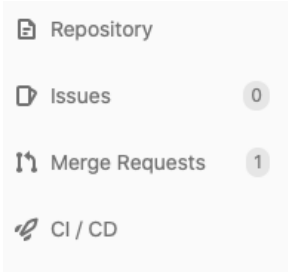
Now let's say that you like the changes that you've made in the other branch (in this case, adding newfile.txt) and you want to add those changes to the master branch. You do this by merging the two branches together. To begin this process, click on that banner that we ignored earlier, on the button that says, "Create merge request". This will bring you to the following page:

[21] Sometimes GitLab will default to placing you into the master branch, so it's a good idea to check back often to make sure that you're on the correct branch.

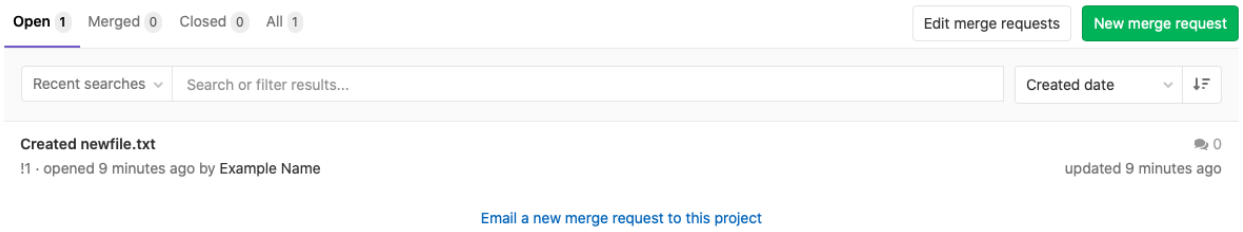
[22] In fact, if you had made changes to the contents of any of the files, those changes would disappear when you toggle over to the master branch and then reappear when you return to the experiment branch.



Type a message explaining why you want to merge your branch with the master branch, and then click on the green "Submit merge request" icon. You will be brought to a frightening-looking page. We will come back to this soon, but for now go ahead and navigate back to the main project page of the master branch. Notice that on the sidebar you see the following:

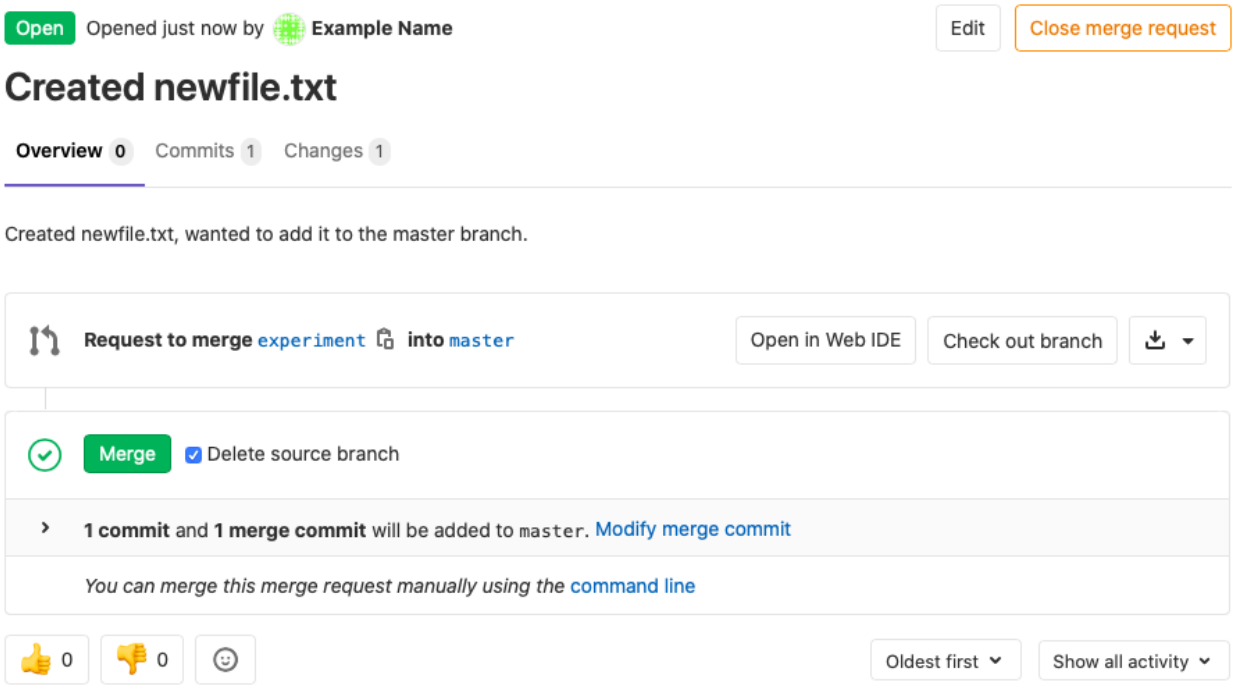


That "1" next to the "Merge Requests" tab indicates that you have a merge request. This comes in handy when you're working on a project and someone else submits a merge request, and perhaps they do not have the proper permissions to actually complete the merge to the master branch, but you do. To complete the merge, click on the "Merge Requests" tab. You will be brought to a list of all of the current merge requests from all of the branches of the project, which right now is only one:[23]

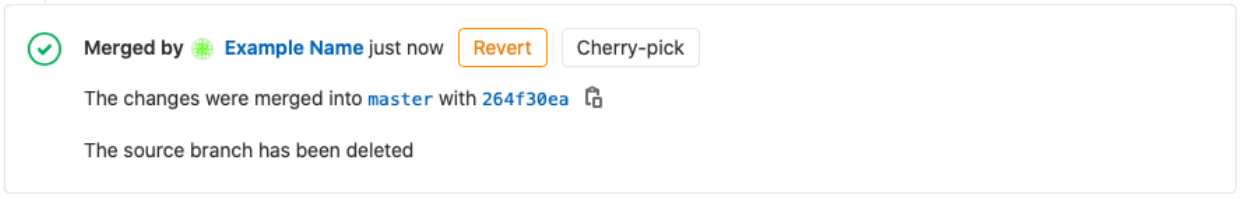


[23] If you do not have that banner that we used to select "Create merge request", then you can come to this tab and click on the green "New merge request" icon to create a new merge request.

Click on the merge request that you would like to complete, and you will be brought back to that frightening-looking page from earlier:

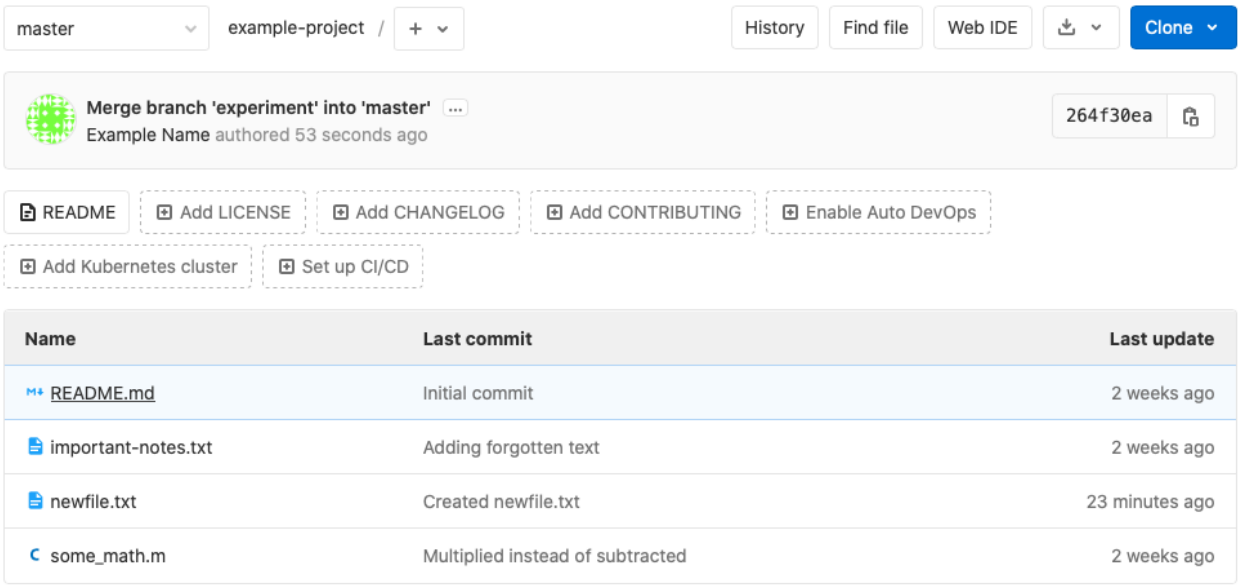


This is a relatively simple merge because we have no conflicts. A conflict would occur if a single file were edited in both the experiment and master branch, and would have to be manually resolved, which is the topic of the next section. Because all we did was create a new file, this is an easy merge. Click on the green "Merge" icon. Note that GitLab assumes that you are done with the branch it and will automatically delete that branch after the merge.[24] You can avoid this by unchecking the "Delete source branch" box. After clicking on the "Merge" icon you will see:



To verify that the merge was successful, return to the main project page of the master branch:

[24] When you are done with a branch, it is good practice to delete the branch so that it doesn't clutter the workspace. GitLab assumes that when you're merging your branch to the master branch that you are done with the branch.

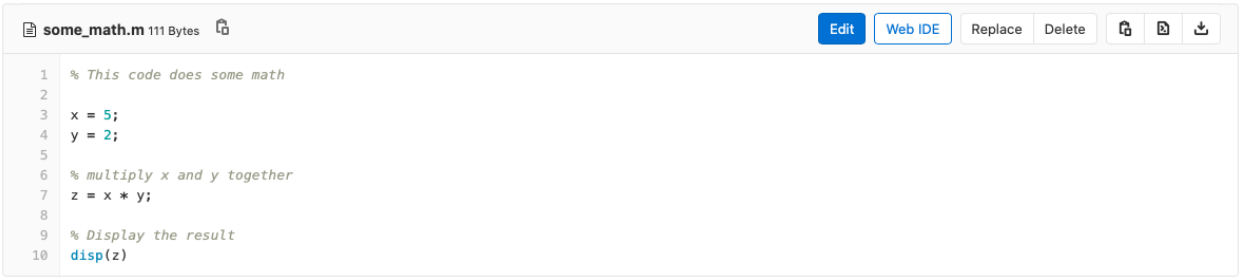


The newfile.txt file has successfully been added to the master branch.

3.3.2.1 Resolving Merge Conflicts

Let's say that you have made changes to some_math.m in a branch, meanwhile some changes were made to some_math.m in the master branch, and that these changes conflict with each other. When the time comes to merge your changes into the master branch you will have to resolve these conflicts. To demonstrate this, let's create a new branch called "adding-new-math".

Open the file some_math.m. For reference, mine looks like this:

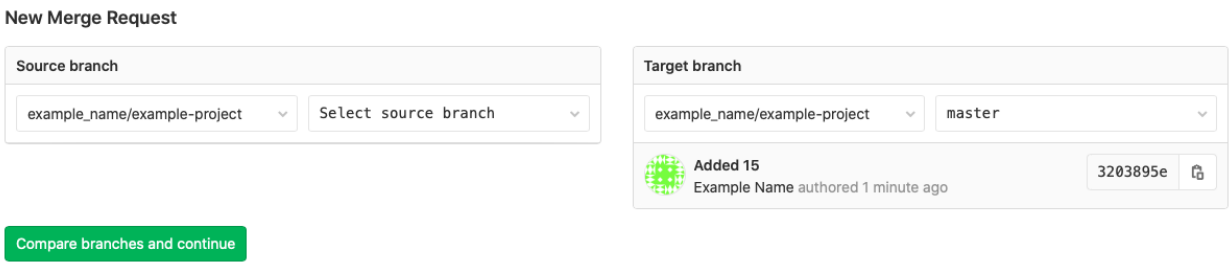


In this example I am going to change line 7 to read: z = x * y + 10. This could be done either by editing the file in the browser or by uploading a new version of the file with that revision. The latter is similar to what you'll be doing on a day-to-day basis.

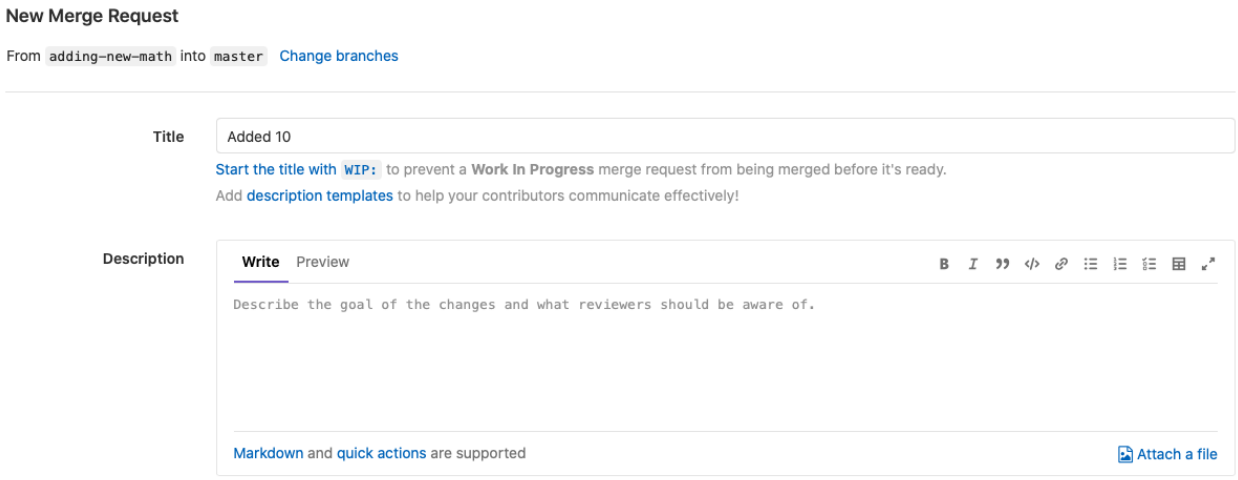
Once that revision is complete, we will return to the master branch and make a very similar change, this time making line 7 read: z = x * y + 15.[25] Now that we have conflicting changes to

[25] This simulates what you would experience if a team member had merged changes from one of their branches with the master branch.

our files, go ahead and create a merge request by clicking on the “Merge Requests” tab on the sidebar and then on the green “New merge request” icon.[26] You will see the following:

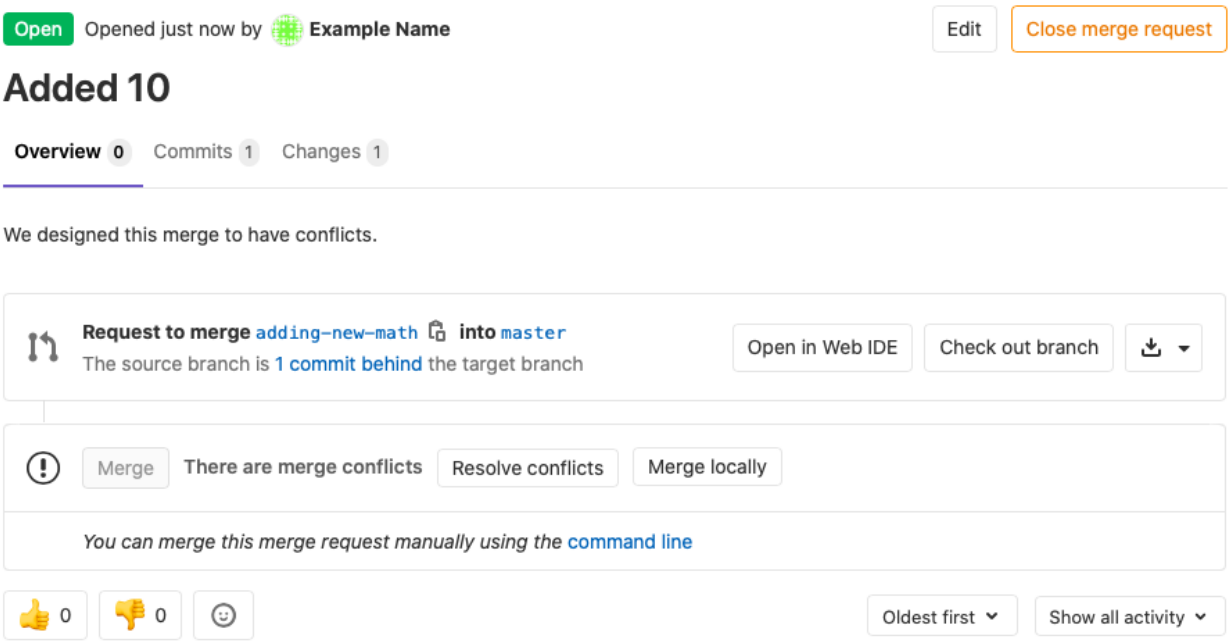


For the source branch, select adding-new-math and then click on the green “Compare branches and continue” icon. You will be brought to the following page:

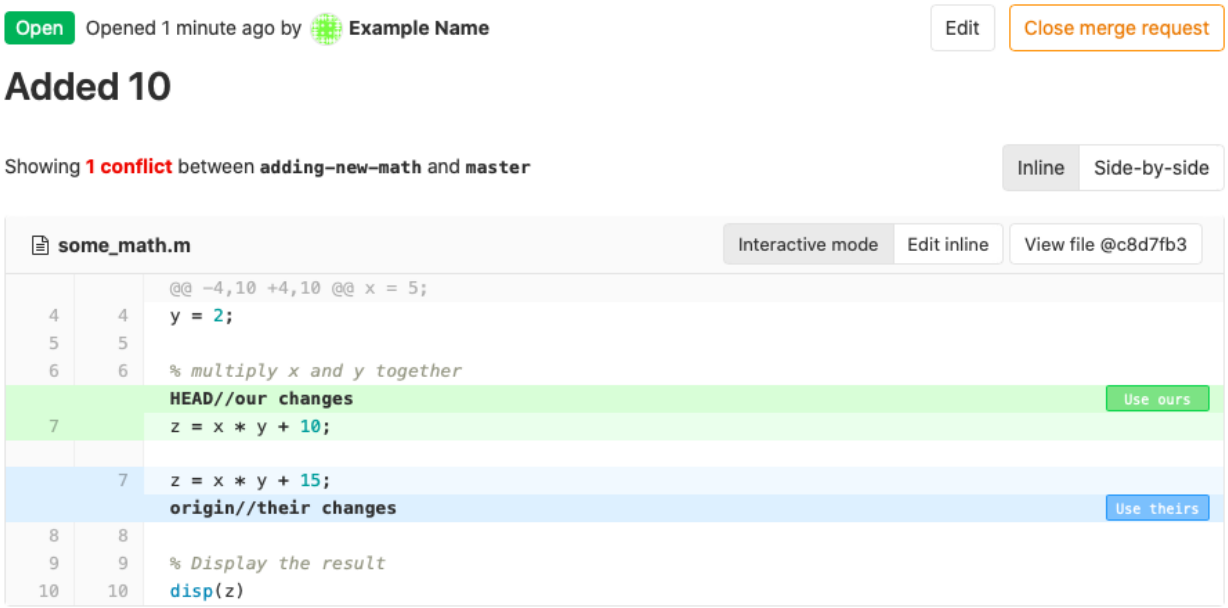


This is the same page that you saw in the previous section when creating a new merge request. Change the title if you want, add a description of the merge, and then click on “Submit merge request”. You will be brought to the following page:

[26] Notice that this is the longer and more proper way of submitting a merge request, compared to the method discussed in the previous section.



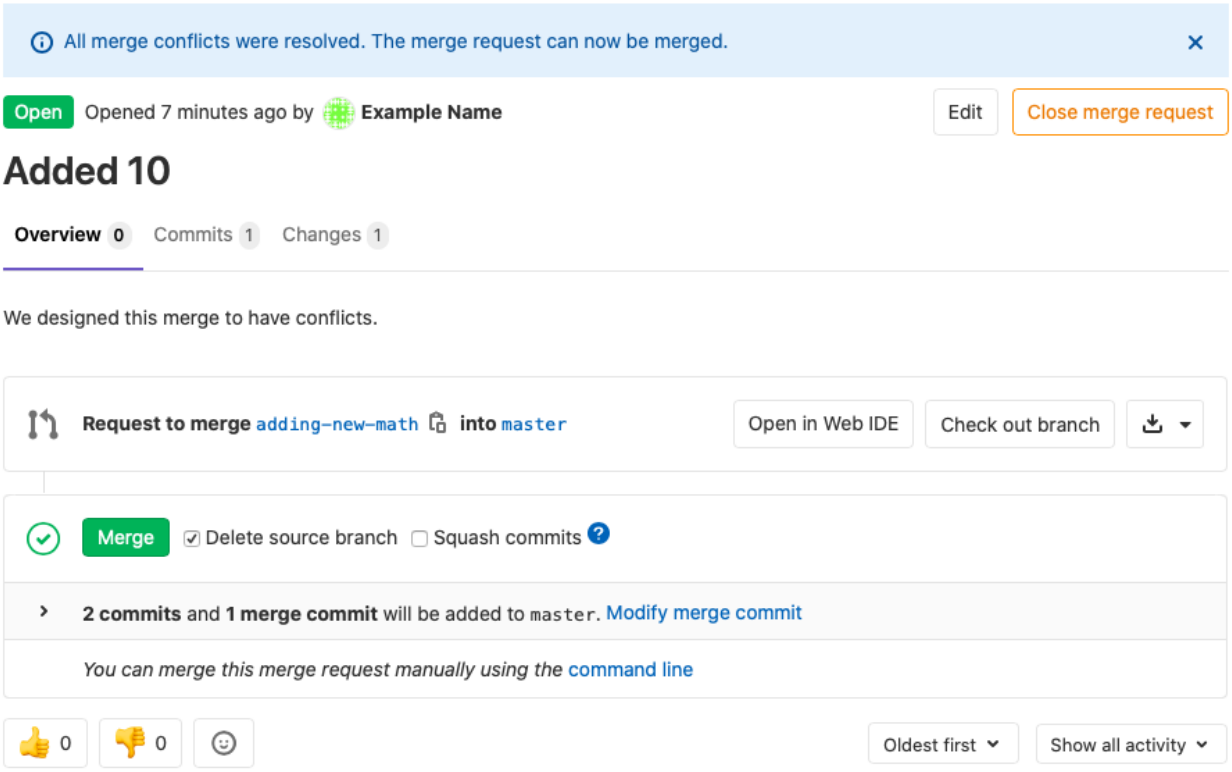
Notice the box that says there are merge conflicts.[27] Click on "Resolve conflicts". You'll be brought to the conflict resolution page:



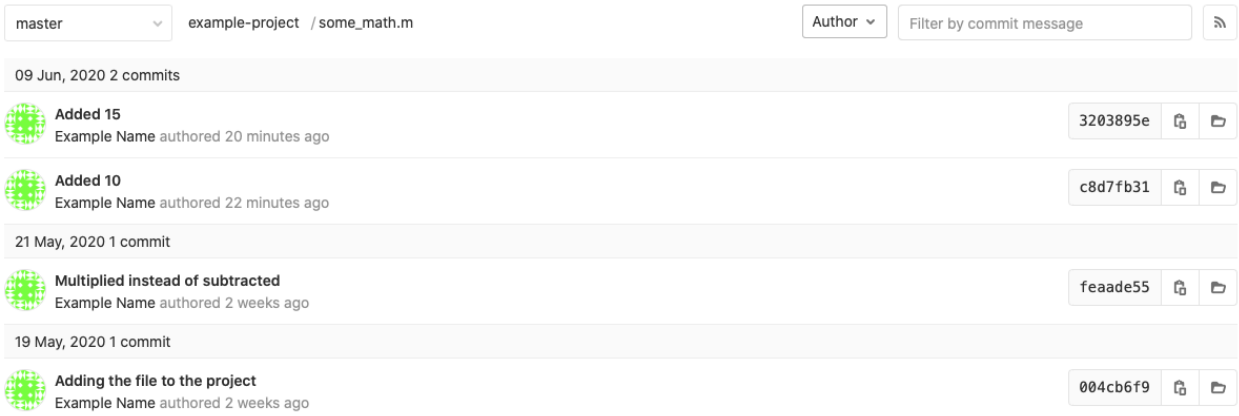
The different colors indicate the different versions trying to be merged. Decide which version you want to keep (discussing with your team as needed) and then click on either "Use ours" or "Use theirs" respectively to make your selection. In this example I selected "Use ours". After selecting

[27] You may even get an email alerting you to the conflicts.

your choice, click on the “Commit to source branch” icon. You will be brought to the following page:



You may need to wait for a few seconds for the green “Merge” icon to appear. Once it does, click on it to merge the branches. Now if you navigate to some_math.m on the master branch you will see that its contents have changed to match whatever you decided in the conflict resolution part. If you now click on the “History” icon you will see the file history:

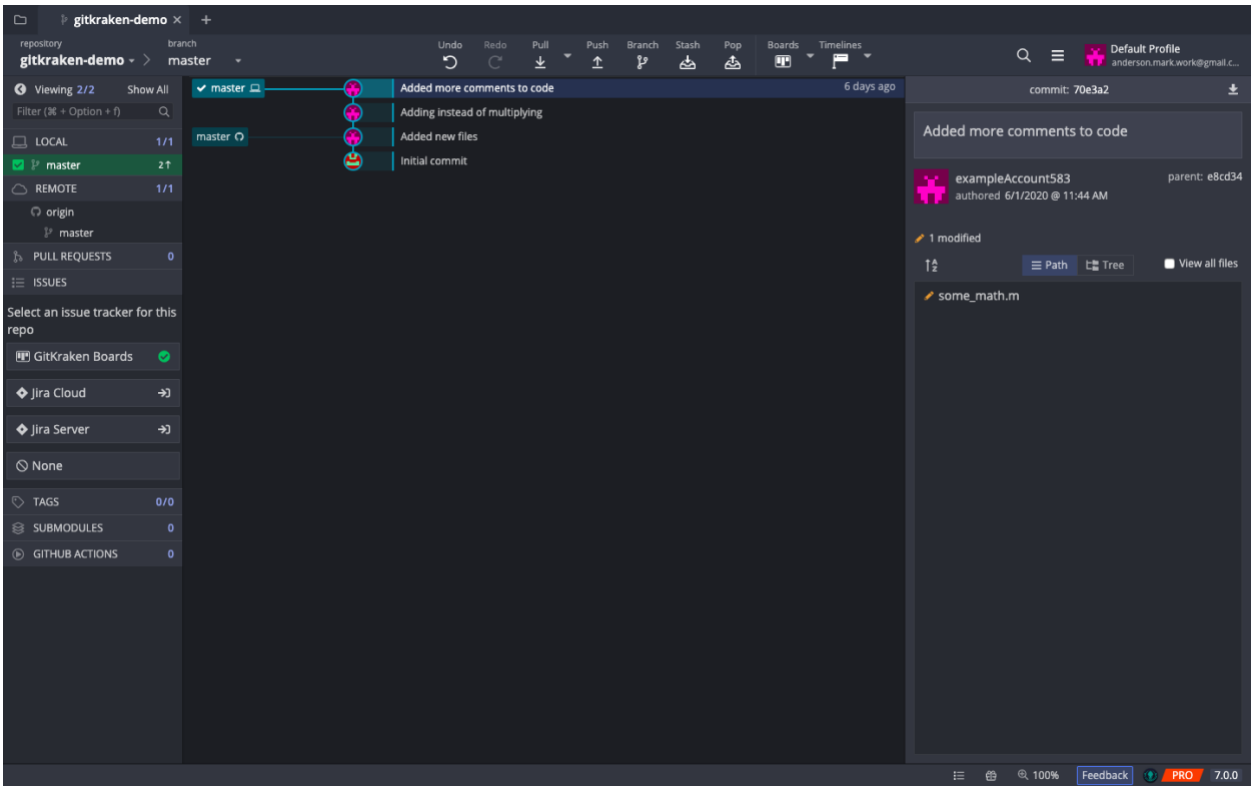


In this example, I made the “Added 10” changes to the adding-new-math branch about two minutes before making the “Added 15” changes to the master branch. Notice that the “Added

10” changes are shown in this history as coming before the “Added 15” changes, even though the merge chose to take the “Added 10” changes instead of the “Added 15” changes. This is just something to watch out for: after merge requests the history of an individual file may appear to be a little out of order.

3.4 GitKraken

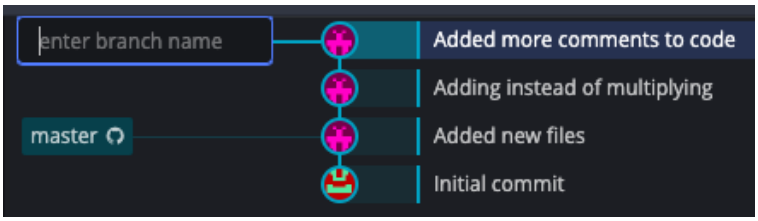
Let's open up a repository called "gitkraken-demo". If you're continuing this section from the previous GitKraken section then this is the same repository we used previously. If not, go ahead and create a repository on GitKraken or on either GitHub or GitLab and then clone it to GitKraken (all of which is explained in Section 2.2.1). Opening the repository, here's what mine looks like:



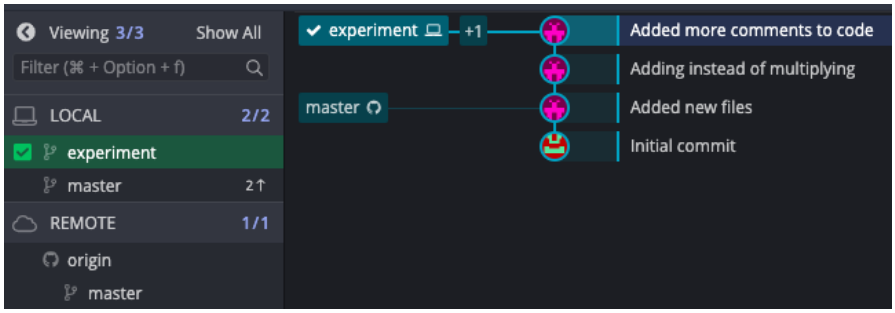
Notice the changes to the timeline (or "tree") are still stored from the last time we used this repository in Section 2.2.1. Also notice that the remote repository on GitHub is behind by two commits (indicated by the location of the "master" box that has the little GitHub symbol next to it). This is okay, and we can sync the changes we make in this section later. When we refer to the "master" branch in this section of the tutorial, we will be referring to the master branch as stored locally on your computer, unless we specify that we are referring to the remote repository. One of the benefits of using Git is that we can do a bunch of work on our computer now and know that we can always sync it up later.

3.4.1 Creating Branches

To create a branch, click on the "Branch" icon in the toolbar at the top center of the GitKraken window. A box will be provided in the tree for you to give the branch a name:

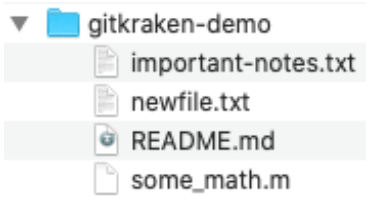


Let's call this branch "experiment". Once you name your branch and hit enter, your screen will reflect those changes:

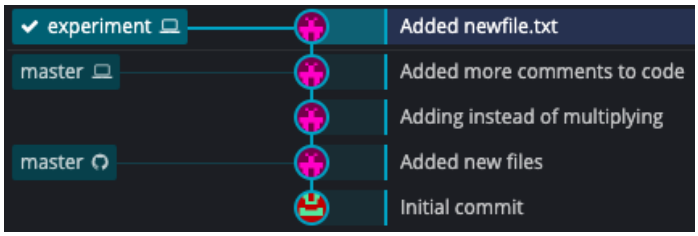


On the left-hand side of this screenshot you can see all of the branches in your repository. Remember that "Local" means they are on your computer and "Remote" means that they are online and stored with the remote repository. In the center of this screenshot you can see that our timeline is updated to show the master branch on GitHub, the "experiment" branch on your computer, and the "+1" indicates the master branch on your computer (which is ahead of GitHub for now in this particular example).

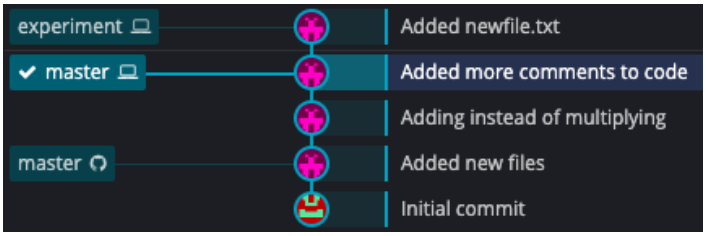
Now that you've entered a new branch, you have essentially entered an "alternate universe" where any changes you make to this branch have exactly zero impact on the rest of the repository, meaning that you can make all kinds of experimental changes and know that you can return to the master branch and see your code exactly as you left it when you branched off. To demonstrate this, go ahead and create a new file in your repository called "newfile.txt", making sure that it's saved to the same folder as the rest of your files, like this:



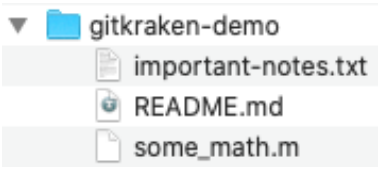
Commit the change to GitKraken, just like in Section 2.2.1.3. After you make the commit, notice that your tree looks like this:



This indicates that now you have changes in your experiment branch that are not in the master branch. To get a feel for what this means, double-click on the master branch, causing it to be highlighted and checked:



Then, return to the repository folder on your computer and notice that "newfile.txt" seems to have disappeared!



When you double-clicked on the master branch, you switched to the master branch. Because of this, the files on your computer were changed to reflect the contents the master branch. If you had made edits to the contents of either important-notes.txt or some_math.m while in the experiment branch, then they those changes would also disappear when you enter the master branch. Hopefully this helps to build your confidence that when you create a new branch you truly enter an "alternate universe" where you can literally make any changes to your files and know that you have a safe backup stored on the master branch.

If you want to push your creation of this branch and any changes you have made within it to the remote repository, switch back to the experiment branch and click on the "Push" icon in the toolbar at the top of the GitKraken screen. Because this is a new branch, GitKraken will ask you where you want this branch to be pushed. Good practice would be to specify a remote branch that matches the name of the local branch. In this case, that would be a branch called "experiment"[28]. Hit "Submit" to push the branch to the remote repository.

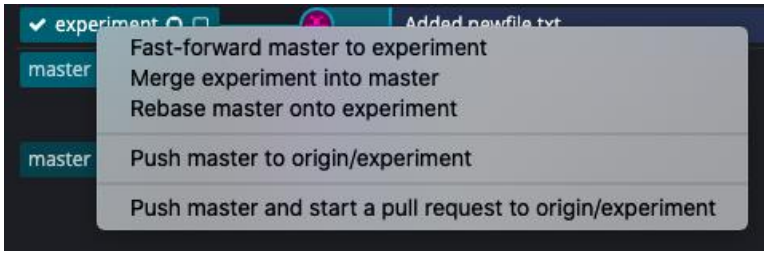
[28] GitKraken may ask you for some authorization to connect with GitHub (or wherever your remote repository is stored). Go ahead and complete the authorization, which should be similar to the authorization found in Appendix Section 5.1.2.

Once submitted, you can go to the remote repository and see that this new branch is safely backed up to the remote repository.

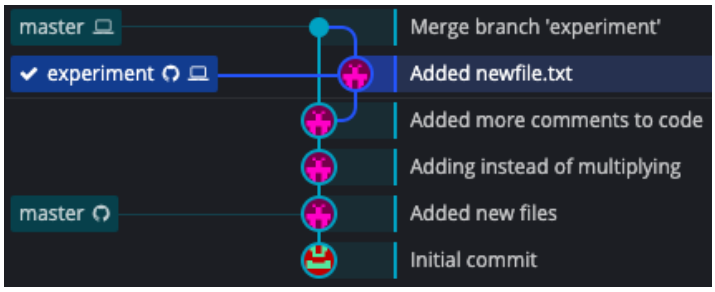
3.4.2 Merging Branches

Let's say that you're satisfied with your creation of "newfile.txt" from the previous section and want to bring that change into the master branch. To do this, you perform a merge.

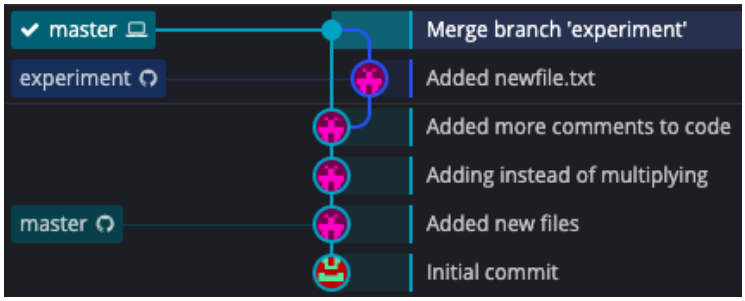
If you want to merge the changes from the experiment branch into the master branch, you will drag the master branch icon onto the experiment icon. This will bring up the following box:



Click on "Merge experiment into master". Because simply adding a file is an easy merge, GitKraken will automatically complete the merge for you, and your new tree will look something like this:



You can see that the two branches (master and experiment) have now been merged, and if you go into your repository folder on your computer you will see that newfile.txt has been merged into the master branch. If you are done with the experiment branch, go ahead and delete it by switching to a different branch and then right-clicking on the experiment branch and selecting "Delete experiment". When prompted about whether you really want to delete it, select that you do. Our tree now looks like this:



Notice that if you pushed the experiment branch up to your remote repository, your tree will reflect the fact that the experiment branch still exists on the remote repository. If you want to delete the remote version of the experiment branch, right-click again on the experiment branch and this time select “Delete origin/experiment”. When this is done, and it may take a few seconds, your tree will look something like this:

Congratulations, you can now create branches and perform simple merges!

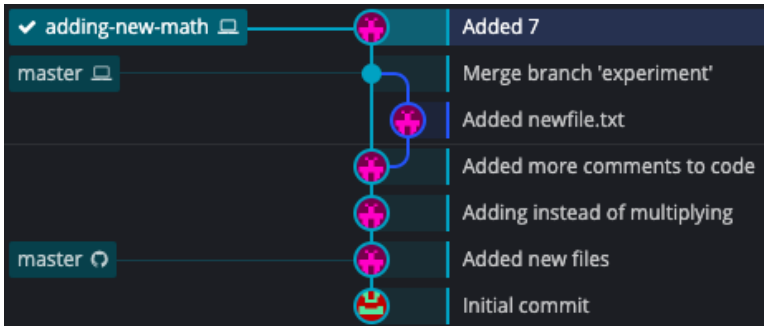
3.4.2.1 Resolving Merge Conflicts

3.4.2.1.1 Conflicts with Yourself

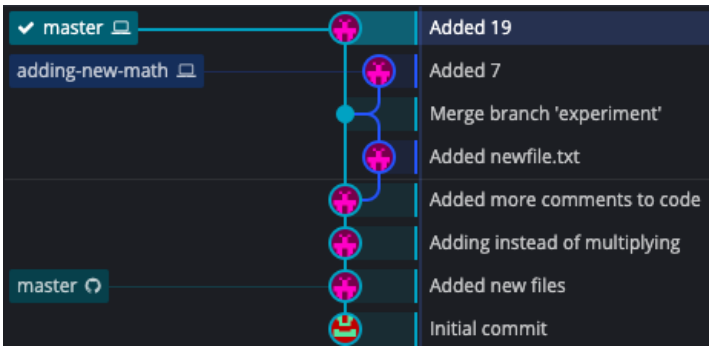
There will come times when you have made edits to a file and reached the point where you are ready to merge your changes into the master branch, but for some reason the master branch has changed in ways that conflict with your changes. To illustrate this, create a branch off of the local master branch called “adding-new-math”. Once inside the adding-new-math branch, open some_math.m and make a change. For example, my version of some_math.m looks like this:

```
1  % This code does some math
2  % The purpose of this simple math is to help us learn about making changes
3  % to files and tracking them with GitKraken
4
5  x = 5;
6  y = 2;
7
8  % add x and y together
9  z = x + y;
10
11 % Display the result
12 disp(z)
```

I will be changing line 9 to read: z = x + y + 7. After committing that change the tree looks like:

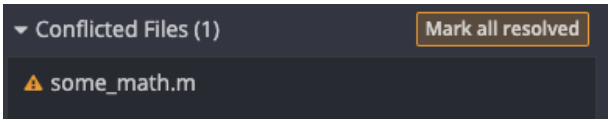


Now, switch back to the master branch and make a change on the same line. In this example, I will change line 9 to read: z = x + y + 19. Once this change is committed the tree will show that both branches have been changed:



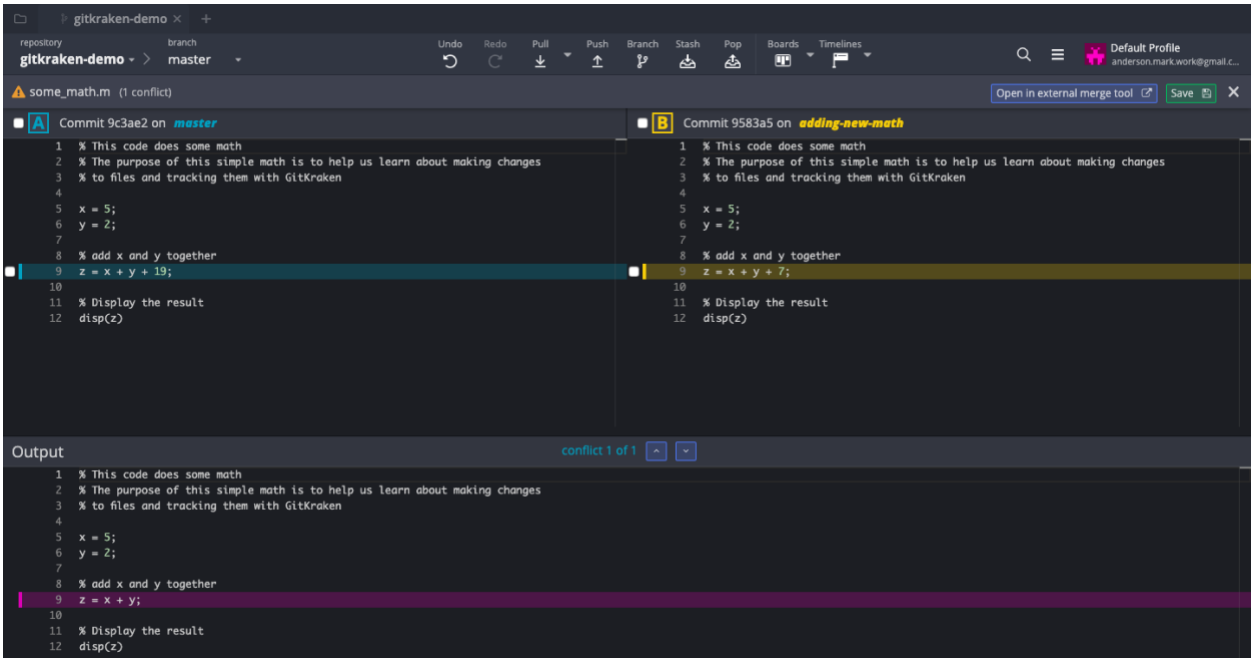
To merge adding-new-math into master, drag the adding-new-math icon onto the master icon and select “Merge adding-new-math into master”. It’s a good idea to always double check that the merge is going the way you expect and not the other way around.[29]

This time, instead of merging automatically, you will get notifications that say that the merge failed and that there are “merge conflicts”. To resolve these conflicts, go to the right-hand side of the screen and select the conflicted file from a list:



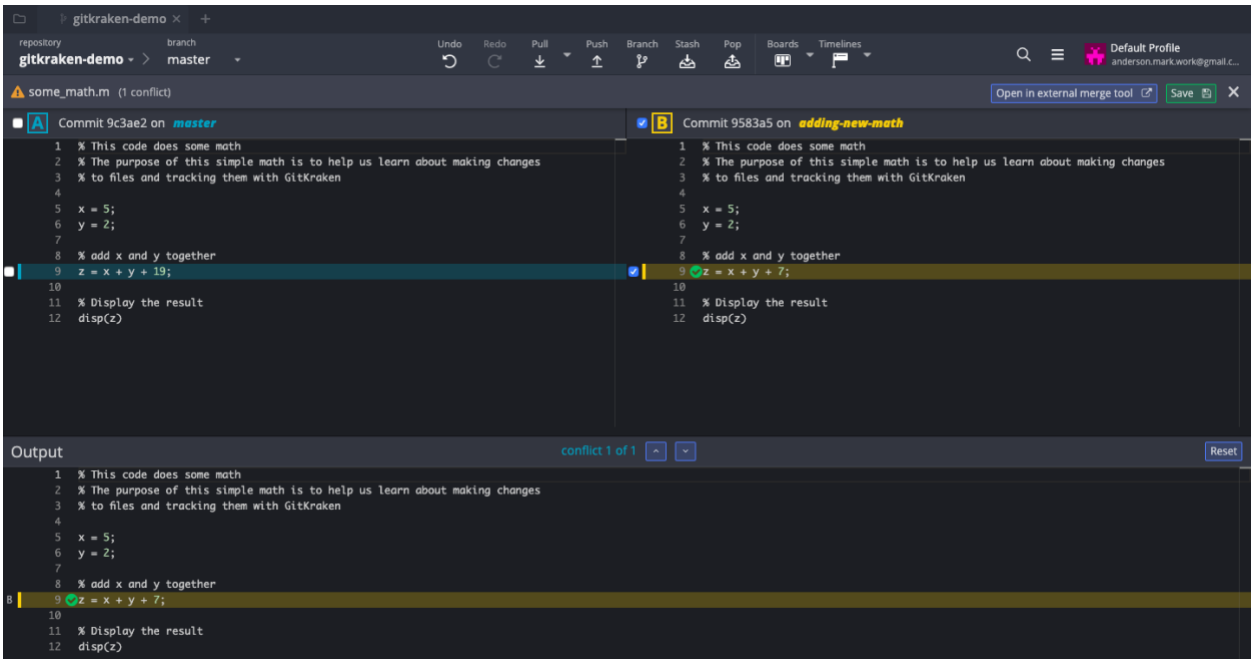
This will open GitKraken’s built-in **merge tool**, and you’ll see a screen similar to this:

[29] GitKraken’s “Undo” button can help you undo any accidental backwards merges.

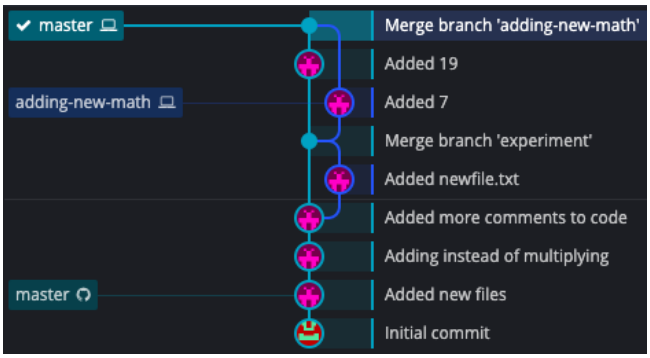


On the upper left in this screenshot we have the master branch version of some_math.m. On the upper right we have the adding-new-math version. On the bottom we have the final result of the merge. Right now, the final result is showing the version that it knows about before either of the conflicting changes occurred.

Next to the highlighted lines there is a little checkbox. Select the checkbox next to the version that you want to keep, and that will be reflected in the final version at the bottom of the screen, like so:



When you're done, hit the green "Save" icon in the upper-right corner. Then, on the right-hand side of the screen you'll see that some_math.m is now staged for committing. Go ahead and commit the changes and your tree will now look something like:



Again, go ahead and delete the adding-new-math branch if you're done with it and there you go! You can now handle merge conflicts using GitKraken!

3.4.2.1.2 Conflicts with Others

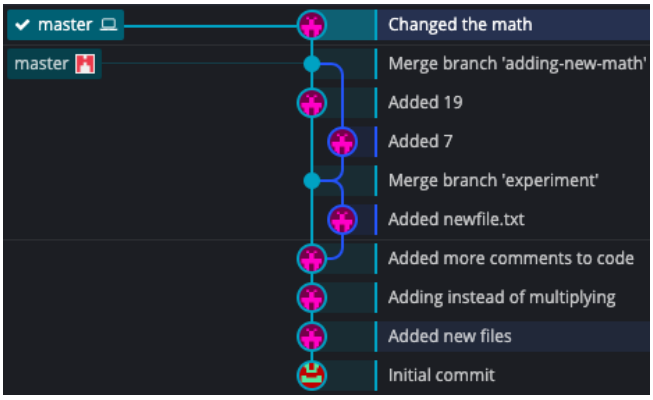
While the previous section covered merge conflicts that you find between branches on your computer, the more likely scenario is that you will find merge conflicts between your files and the files stored on the remote repository. Resolving conflicts between the local and remote repositories is very similar to what we did in the previous section. To illustrate this, first go ahead and push your changes to the remote repository to make sure that your local and remote repositories are in sync with each other[30]. Then open the file some_math.m on your computer. For reference, here's what mine looks like for now:

```
1  % This code does some math
2  % The purpose of this simple math is to help us learn about making changes
3  % to files and tracking them with GitKraken
4
5  x = 5;
6  y = 2;
7
8  % add x and y together
9  z = x + y + 7;
10
11 % Display the result
12 disp(z)
```

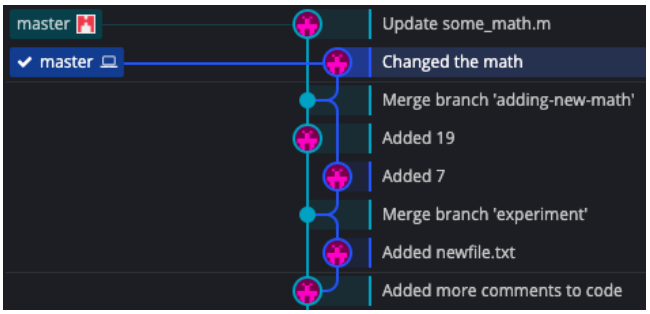
I will be completely changing line 9 to read: z = x + y/6 – r^3, along with the comment line above it to accurately reflect those changes.

Once complete, commit the latest revision of some_math.m to your local repository, without pushing it to the remote repository. Your tree now might look something like this:

[30] This way we don't end up with more conflicts than we expect for this section of the tutorial. This will help us keep it simple.



Notice how the remote repository is now lagging behind by one commit[31]. Let's now go online into the remote repository and make similar edits, this time changing line 9 to be: z = x + y/3 – r^6, along with the comment line above it. Making a change directly to the remote repository simulates what it would be like if a teammate made changes and pushed them to the remote repository. After a few minutes, your tree will reflect the fact that changes have been made to the remote repository by showing that your version of the master branch and the remote version of the master branch are effectively two distinct branches for the time being:



If you try to push your changes to the remote repository, you will get an error that will ask you to do a pull first.[32] A pull is the opposite of a push and brings changes from the remote repository into your local repository. This let you make sure your code works with the changes that have been made online before you push your changes to the remote repository. So, go ahead and click the "Pull" icon from the top toolbar. Since we have conflicting changes, we will get an error message. The next steps are identical to what was discussed in the previous section:

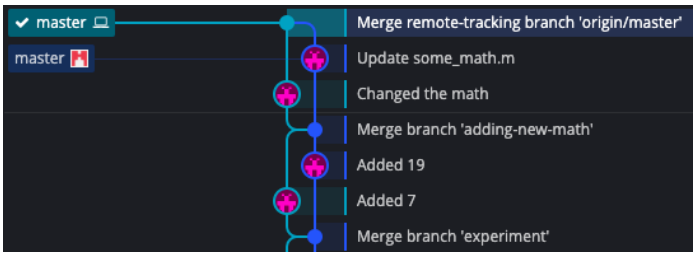
1. Click on the conflicted file(s) on the right-hand side of the screen
2. Select which versions of the changes you want to keep. In this example we will keep the local changes.

[31] The picture next to it has changed to reflect my automatically generated profile picture on GitHub. I'm not sure why it decided to change now and not earlier in the tutorial, but this is still the same old remote repository that we've been dealing with all along.

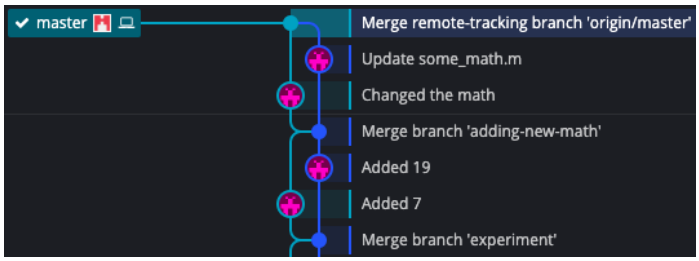
[32] It does give you the option to do a "Force push", but this is generally poor practice because it may result in overwriting someone else's work.

3. Hit “Save” in the upper right-hand corner
4. The file will automatically be staged for committing, so write a commit message and then commit the file.

After you’ve committed the file, the changes have successfully been merged on your computer, and your tree will look something like this:



Now that you’ve pulled the changes from the remote repository to your local repository, merged them successfully, and verified that everything still works the way you expect,[33] go ahead and push the changes to the remote repository by clicking the “Push” icon in the toolbar. Once complete, your tree will look something like this:



If you go to the remote repository, you will see your changes safely merged into the master branch. Congratulations, you can now handle merge conflicts with other people!

[33] It’s good practice to run the code and make sure that it works as expected after the merge. It would really be a bummer to push code to the master branch that doesn’t work.

3.5 The Terminal

Let's open up the repository that we used in Section 2.2.2. This will have some files for us to experiment with.[34] Let's first start off by making sure our local repository is up to date with the remote repository:

```
markanderson@MarkAndComputer ~ % ls                                    1
Box/        Documents/ Library/   Music/     Other/     Public/    keras/
Desktop/    Downloads/ Movies/    OneDrive/  Pictures/  VIA/
markanderson@MarkAndComputer ~ % cd Desktop/git-terminal-practice      2
markanderson@MarkAndComputer git-terminal-practice % git status       3
On branch master                                                       4
Your branch is ahead of 'origin/master' by 6 commits.                  5
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice % git pull         6
Username for 'https://github.com': exampleAccount583
Password for 'https://exampleAccount583@github.com':
Already up to date.
markanderson@MarkAndComputer git-terminal-practice % git push         7
Enumerating objects: 22, done.
Counting objects: 100% (22/22), done.
Delta compression using up to 4 threads
Compressing objects: 100% (19/19), done.
Writing objects: 100% (19/19), 2.08 KiB | 533.00 KiB/s, done.
Total 19 (delta 9), reused 0 (delta 0)
remote: Resolving deltas: 100% (9/9), completed with 2 local objects.
To https://github.com/exampleAccount583/git-terminal-practice.git
   ad2d475..bd21b34  master -> master
markanderson@MarkAndComputer git-terminal-practice % git status       8
On branch master
Your branch is up to date with 'origin/master'.                        9

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice %
```

1. Seeing what is in the current directory. I've used an **alias** to make the output colored and to specify which directories are actually folders.
2. Changing the directory to enter the git-terminal-practice repository
3. Checking the status of the repository
4. We are on the master branch (that will change soon)
5. We are ahead of the remote version of the repository by six commits. This is because I did not commit at the end of Section 2.2.2. But now, I want to sync my changes up to the remote repository.

[34] If you do not have this particular repository, go ahead and create a repository online with a few files in it and clone it to your computer. This will help you follow along with the tutorial.

6. Before pushing your changes, you want to make sure that nobody else has made changes to the remote repository. If you don't do a pull before you push you can accidentally delete someone else's work.
7. Once you have pulled any changes from the remote repository to the local repository, push your changes to the remote repository.
8. Once again, check the status of your local repository to make sure that everything's as you expect it to be.
9. Your local master branch is up to date with the remote master branch.

3.5.1 Creating Branches

Now that your repository is freshly synced-up with the remote repository, let's go ahead and create a branch. To do this, type:

```
git branch <name-of-branch>
```

Then, to actually move to the branch that you just created[35], type:

```
git checkout <name-of-branch>
```

For example, this looks like:

```
markanderson@MarkAndComputer git-terminal-practice % git branch experiment
markanderson@MarkAndComputer git-terminal-practice % git checkout experiment
Switched to branch 'experiment' <- 1
markanderson@MarkAndComputer git-terminal-practice % git status
On branch experiment <- 2
nothing to commit, working tree clean <- 3
markanderson@MarkAndComputer git-terminal-practice %
```

1. You have switched to the "experiment" branch
2. Verifying again, you are on the "experiment" branch
3. You haven't done anything yet, so you don't need to commit anything yet

You will likely want to add this branch that you've created to the remote repository too. To do this, type:

```
git push -u origin <name-of-branch>
```

Doing this looks like:

[35] You can create a new branch and move into it in one line by typing: git checkout -b <name-of-branch>

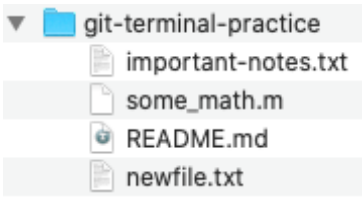
```
markanderson@MarkAndComputer git-terminal-practice % git push -u origin experiment
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'experiment' on GitHub by visiting:
remote:      https://github.com/exampleAccount583/git-terminal-practice/pull/new/experiment
remote:
To https://github.com/exampleAccount583/git-terminal-practice.git
 * [new branch]      experiment -> experiment
Branch 'experiment' set up to track remote branch 'experiment' from 'origin'.
markanderson@MarkAndComputer git-terminal-practice % git status
On branch experiment
Your branch is up to date with 'origin/experiment'.

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice %
```

1
2
3

1. Teaching you how to create a pull request (more on pull requests later)
2. A branch called “experiment” was set up on the remote repository that tracks the changes made in your local “experiment” branch.
3. Unsurprisingly, your local version of the experiment branch is up to date with the remote version at this point in time.

Now that you’ve created a branch, you are in a completely safe experimental space, which we can think of as an “alternate universe”. Any changes that you make while in this new branch will have exactly zero effect on the master branch. So, you can experiment away with a radical new idea and see if it works. If it doesn’t, it’s no big deal – just delete the branch and create a new one for the next big idea. All the while, a functional version of your code is safely stored in the master branch, accessible any time you need it.

To help demonstrate the freedom that a new branch gives you, make sure that you’re in the experiment branch and then go ahead and create a new file called “newfile.txt” and put it in your repository folder. Once complete, the repository folder should look something like this (assuming the same structure as used in Section 2.2.2):



Now go ahead and commit the change to your Git repository:

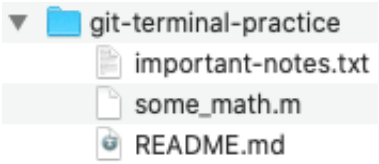
```
markanderson@MarkAndComputer git-terminal-practice % git status
On branch experiment
Your branch is up to date with 'origin/experiment'.

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

nothing added to commit but untracked files present (use "git add" to track)
markanderson@MarkAndComputer git-terminal-practice % git add newfile.txt
markanderson@MarkAndComputer git-terminal-practice % git commit -m "Added newfile.txt"
[experiment 60c5125] Added newfile.txt
 1 file changed, 1 insertion(+)
 create mode 100644 newfile.txt
markanderson@MarkAndComputer git-terminal-practice % git status
On branch experiment
Your branch is ahead of 'origin/experiment' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice %
```

Now here comes the cool part. Go ahead and checkout the master branch and then try to find the newfile.txt file that you just created.

```
markanderson@MarkAndComputer git-terminal-practice % ls
README.md            important-notes.txt  newfile.txt          some_math.m   <- 1
markanderson@MarkAndComputer git-terminal-practice % git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
markanderson@MarkAndComputer git-terminal-practice % ls
README.md            important-notes.txt  some_math.m          <- 2
markanderson@MarkAndComputer git-terminal-practice %
```

1. At this point we're still in the experiment branch. When we list the current files, we see that newfile.txt is in the repository.
2. After switching to the master branch, we see that when we list the current files, newfile.txt no longer exists!

Sure enough, when you look in the folder you see that newfile.txt is missing!



In fact, if you had made any changes to the contents of any of the files while in the experiment branch, then when you go back to the master branch those changes will be completely gone too. Hopefully this helps to boost your confidence that branches let you try new things in a care-free manner, trusting that if your idea doesn't work out, you can easily go back to the functioning code on the master branch.

3.5.2 Merging Branches

Let's say that you're really pleased with newfile.txt from the previous section, and you would like it to become part of the master branch. To do this, you must merge the two branches together. Switch to the master branch and then type:

```
git merge <name-of-branch>
```

This might look like:

```
markanderson@MarkAndComputer git-terminal-practice % git checkout master
Switched to branch 'master'
Your branch is up to date with 'origin/master'.
markanderson@MarkAndComputer git-terminal-practice % ls
README.md            important-notes.txt  some_math.m          <- 1
markanderson@MarkAndComputer git-terminal-practice % git merge experiment
Updating b778230..400448c
Fast-forward
 .gitignore  | 1 +                                            <- 2
 newfile.txt | 0
 2 files changed, 1 insertion(+)
 create mode 100644 .gitignore
 create mode 100644 newfile.txt
markanderson@MarkAndComputer git-terminal-practice % ls
README.md            important-notes.txt  newfile.txt          some_math.m   <- 3
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice %
```

1. Notice how the newfile.txt file is not in the master branch.
2. Here is a list of files that are going to be changed/added.[36] Your output may have different numbers and plus/minus signs next to them. That's okay and depends on what you actually typed into newfile.txt.
3. After the merge, newfile.txt appears in the master branch.

There you go, you can now handle simple merges. Once you're done with the experiment branch, go ahead and delete it by typing:

```
git branch -d <name-of-branch>
```

This looks like:

```
markanderson@MarkAndComputer git-terminal-practice % git branch -d experiment
Deleted branch experiment (was 400448c).
markanderson@MarkAndComputer git-terminal-practice %
```

[36] We didn't cover creating a .gitignore file in this section, but it is discussed in Section 4.2 if you're interested in what it means.

Sometimes you might find yourself with some errors while trying to delete branches. While it is best practice to try and resolve those errors if possible, should you ever need to force a branch to be deleted, type:

```
git branch -D <name-of-branch>
```

If you want to delete the branch on the remote repository too, type:

```
git push origin --delete <name-of-branch>
```

3.5.2.1 Resolving Merge Conflicts

Let's say that you created a branch off of the master branch and have been working in that branch for a while, and you're ready to merge your improvements into the master branch. However, in the meantime, the master branch has undergone some changes that directly conflict with yours. Because of these merge conflicts, we will need to do a **manual merge**. To illustrate this, let's walk through two examples:

3.5.2.1.1 Conflicts with Yourself

Go ahead and create a branch called "adding-new-math" in your repository. In this repository, open some_math.m and make a change. For example, here is what my version of some_math.m looks like:

```
1  % This code does some math
2  % The purpose of this simple math is to help us learn about making changes
3  % to files and tracking them with GitKraken
4
5  x = 5;
6  y = 2;
7  r = 3;
8
9  % multiply x and y and divide by r^2
10 z = x * y / r^2;
11
12 % Display the result
13 disp(z)
14
15 % Here is another comment
```

In the adding-new-math branch I will change line 10 to read: z = x * y / r^2 + 3.

In the master branch I will change line 10 to read: z = x* y / r^2 + 5.

The following terminal screenshot shows all of this and is included as a reference so you can see it all happen:

```
markanderson@MarkAndComputer git-terminal-practice % git branch adding-new-math          1
markanderson@MarkAndComputer git-terminal-practice % git checkout adding-new-math        2
Switched to branch 'adding-new-math'
markanderson@MarkAndComputer git-terminal-practice % git push -u origin adding-new-math  3
Total 0 (delta 0), reused 0 (delta 0)
remote:
remote: Create a pull request for 'adding-new-math' on GitHub by visiting:
remote:      https://github.com/exampleAccount583/git-terminal-practice/pull/new/adding-new-math
remote:
To https://github.com/exampleAccount583/git-terminal-practice.git
 * [new branch]      adding-new-math -> adding-new-math
Branch 'adding-new-math' set up to track remote branch 'adding-new-math' from 'origin'.
markanderson@MarkAndComputer git-terminal-practice % code some_math.m                     4
markanderson@MarkAndComputer git-terminal-practice % git status                           5
On branch adding-new-math
Your branch is up to date with 'origin/adding-new-math'.

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:   some_math.m

no changes added to commit (use "git add" and/or "git commit -a")
markanderson@MarkAndComputer git-terminal-practice % git commit -am "Adding 3"           6
[adding-new-math 8c5b676] Adding 3
 1 file changed, 1 insertion(+), 1 deletion(-)
markanderson@MarkAndComputer git-terminal-practice % git checkout master                  7
Switched to branch 'master'
Your branch is ahead of 'origin/master' by 1 commit.
  (use "git push" to publish your local commits)
markanderson@MarkAndComputer git-terminal-practice % git status                           8
On branch master
Your branch is ahead of 'origin/master' 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:   some_math.m

no changes added to commit (use "git add" and/or "git commit -a")
markanderson@MarkAndComputer git-terminal-practice % git commit -am "Adding 5"           9
[master 5d74075] Adding 5
 1 file changed, 1 insertion(+), 1 deletion(-)
markanderson@MarkAndComputer git-terminal-practice %
```

1. Creating the adding-new-math branch
2. Switching to the adding-new-math branch
3. Backing the adding-new-math branch up to the remote repository
4. Oops, this is a shortcut I made that opens the file using Visual Studio Code. You can set up such shortcuts in your terminal if you like, or you can just open the file manually or type “open <name of file>” to open it in a default editor.
5. Immediately before typing this command, I changed line 10 of some_math.m in the adding-new-math branch. That change is reflected in the status of the repository.
6. Committing the change to the adding-new-math branch. I used the shortcut -a to add the files in the same line that I commit them.
7. Checking out the master branch. The edits made in the adding-new-math branch are now gone and irrelevant for now.

8. Immediately before typing this command, I changed line 10 of some_math.m in the master branch. I already had the file open so there was no need to use the terminal to open it. The change is reflected in the status of the repository.
9. Committing the change to the master branch

Now that you have two conflicting changes on your two branches, try merging them. You'll see the following output:

```
markanderson@MarkAndComputer git-terminal-practice % git checkout master
Already on 'master'
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
markanderson@MarkAndComputer git-terminal-practice % git merge adding-new-math
Auto-merging some_math.m
CONFLICT (content): Merge conflict in some_math.m
Automatic merge failed; fix conflicts and then commit the result.
markanderson@MarkAndComputer git-terminal-practice %
```

1

1. There is a conflict because the two branches have changes to the same line in some_math.m.

Git will need your help deciding which version to keep. There are several ways to do this:

- Open the file and manually edit the standard conflict-resolution markers
- Abort the merge, push both branches to the remote repository, and perform the merge online
- Abort the merge, open a program like GitKraken, and perform the merge in there

For information about the last two methods, consult the relevant sections in this tutorial. For now, we are going to focus on doing it the first way. Open the file that has a conflict. In this case, open some_math.m from the current branch (the master branch in our case). There are all kinds of fancy editors that you can use, and I personally use Visual Studio Code, which has many built-in Git and merge tool features. However, if it really comes down to it, you can definitely use a normal text editor to resolve merge conflicts. Here, I have opened some_math.m in a standard text editor (you can use the default text editor on your computer):

```
% This code does some math
% The purpose of this simple math is to help us learn about making changes
% to files and tracking them with GitKraken

x = 5;
y = 2;
r = 3;

% multiply x and y and divide by r^2
<<<<<<< HEAD
z = x * y / r^2 + 5;
=======
z = x * y / r^2 + 3;
>>>>>>> adding-new-math

% Display the result
disp(z)

% Here is another comment
```

The space beneath the "<<<<< HEAD" and above the "======" represents what the current branch (the master branch) has in that spot for this file. The space below "======" and above ">>>>> adding-new-math" represents the incoming change from the adding-new-math branch. Depending on the consequences at stake, you may need to consult with your team before choosing a version to go with. However, for now let's choose the incoming change from the adding-new-math branch. To do this, remove all of the conflict-resolution markers along with the version that you don't want to keep, leaving:

```
% This code does some math
% The purpose of this simple math is to help us learn about making changes
% to files and tracking them with GitKraken

x = 5;
y = 2;
r = 3;

% multiply x and y and divide by r^2
z = x * y / r^2 + 3;

% Display the result
disp(z)

% Here is another comment
```

Here, we have decided that we wanted the changes in the adding-new-math branch to "overrule" the changes made in the master branch. We can now return to the terminal, add the changes to the staging area, commit them, and then we're done merging.

The entire process in the terminal looks something like this:

```
markanderson@MarkAndComputer git-terminal-practice % git checkout master   1
Already on 'master'
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)
markanderson@MarkAndComputer git-terminal-practice % git merge adding-new-math   2
Auto-merging some_math.m
CONFLICT (content): Merge conflict in some_math.m                   3
Automatic merge failed; fix conflicts and then commit the result.
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

You have unmerged paths.
  (fix conflicts and run "git commit")                  4
  (use "git merge --abort" to abort the merge)

Unmerged paths:
  (use "git add <file>..." to mark resolution)          5
        both modified:   some_math.m

no changes added to commit (use "git add" and/or "git commit -a")
markanderson@MarkAndComputer git-terminal-practice % git add some_math.m   6
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 2 commits.
  (use "git push" to publish your local commits)

All conflicts fixed but you are still merging.          7
  (use "git commit" to conclude merge)

Changes to be committed:
        modified:   some_math.m                         8

markanderson@MarkAndComputer git-terminal-practice % git commit -m "Merged branches"
[master 39862ad] Merged branches                        9
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 4 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean                   10
markanderson@MarkAndComputer git-terminal-practice %
```

1. Making sure we're on the master branch (or whatever branch you want to merge to)
2. Attempt to merge
3. Warning about conflict. We are instructed to fix the conflicts (like we did in the text editor) and then commit that file.
4. The paths have failed to merge. Instructions are given for how to abort the merge if you want.
5. The unmerged file(s) are specified. We are instructed that once we fix the conflicts in the text editor then we can simply add that file to the staging area to tell Git that the conflict is resolved.

6. Immediately before typing this command, we resolved the conflict by manually editing the file in the text editor. Now we are adding it to the staging area to tell Git that the conflict is resolved.
7. Instructions for how to conclude merge by committing the file(s).
8. some_math.m is now added to the staging area and is ready to be committed.
9. The branches have now successfully been merged.
10. Everything is now merged and clean. If you're done with the branch that you just merged into the current branch, go ahead and delete it (In this case we would delete adding-new-math).

3.5.2.1.2 Conflicts with Others

Now let's say that you have done lots of good work on your local repository and are ready to push those changes up to the remote repository. However, in the meantime someone else has made conflicting changes to the remote repository and now you are going to have a merge conflict. To simulate this, open some_math.m again from your local master branch and make an edit. Here is what my some_math.m file looks like right now:

```
1  % This code does some math
2  % The purpose of this simple math is to help us learn about making changes
3  % to files and tracking them with GitKraken
4
5  x = 5;
6  y = 2;
7  r = 3;
8
9  % multiply x and y and divide by r^2
10 z = x * y / r^2 + 3;
11
12 % Display the result
13 disp(z)
14
15 % Here is another comment
```

I am going to completely change line 10 to read: z = x + y / 4 + r^3, along with the comment above it to accurately reflect that change.

After that change is made and committed to the master branch (but not pushed to the remote repository), go to your remote repository online and make a conflicting change. I am going to change line 10 in the remote repository to read: z = x + y/3 + r^4, along with the comment above it to reflect that change.

Now that the remote and local repositories have conflicting changes, let's try and push the local changes to the remote repository and see what happens. First, you always want to use a "pull" command before pushing just in case someone else has edited the remote repository.[37]

[37] If you try to push without pulling first, you risk overwriting someone else's code. Thankfully, Git will generally detect whether there is a conflict and prompt you to pull first, but it is still best practice to actually do the pull before you push.

```
markanderson@MarkAndComputer git-terminal-practice % git status
On branch master
Your branch is ahead of 'origin/master' by 5 commits.
  (use "git push" to publish your local commits)

nothing to commit, working tree clean
markanderson@MarkAndComputer git-terminal-practice % git pull
remote: Enumerating objects: 5, done.
remote: Counting objects: 100% (5/5), done.
remote: Compressing objects: 100% (3/3), done.
remote: Total 3 (delta 2), reused 0 (delta 0), pack-reused 0
Unpacking objects: 100% (3/3), done.
From https://github.com/exampleAccount583/git-terminal-practice
   b778230..3aee728  master     -> origin/master
Auto-merging some_math.m
CONFLICT (content): Merge conflict in some_math.m  1
Automatic merge failed; fix conflicts and then commit the result.
markanderson@MarkAndComputer git-terminal-practice %
```

1. The “pull” command will try to merge the remote repository into your local repository, but it detected a conflict in some_math.m (as expected).

To resolve the conflict, open the some_math.m file in your local repository. It has received those conflict-resolution marks just like in the previous section.

```
% This code does some math
% The purpose of this simple math is to help us learn about making changes
% to files and tracking them with GitKraken

x = 5;
y = 2;
r = 3;

<<<<<<< HEAD
% add x and y/4, then add r cubed
z = x + y / 4 + r^3;
=======
% add x and y/3, then add r to the fourth power
z = x + y/3 + r^4;
>>>>>>> 3aee72816f0d463cafe1558f20065069fad52cfd

% Display the result
disp(z)

% Here is another comment
```

The big scary line underneath the second change represents the unique commit ID from that change in the remote repository. Let’s say that you talk it over with your teammate and they realize that they made a mistake and that your version is correct. Go ahead and remove the conflict resolution markers, along with their version of the changes, and your file will look like this:

```
% This code does some math
% The purpose of this simple math is to help us learn about making changes
% to files and tracking them with GitKraken

x = 5;
y = 2;
r = 3;

% add x and y/4, then add r cubed
z = x + y / 4 + r^3;

% Display the result
disp(z)

% Here is another comment
```

Now, save some_math.m, add it to the staging area, and commit it. Now that the conflict is resolved and committed, you can safely push your local repository to the remote repository.

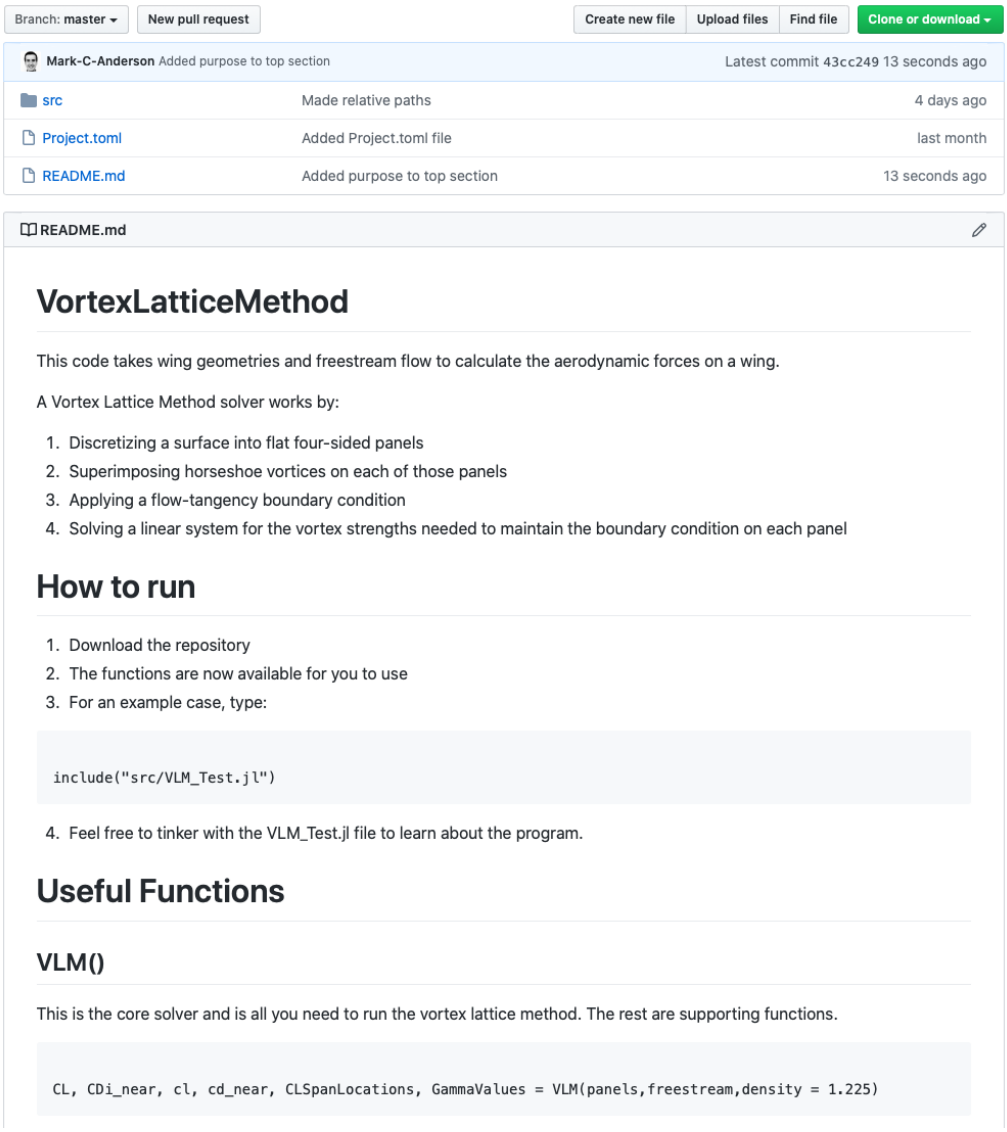
There you go! You can now handle conflicts with other users!

4 Advanced Topics

This section covers topics that go beyond the simple Git workflow. Most of this material will be taught using the terminal (command line) on macOS.

4.1 Writing Effective README.md Files

Online Git providers generally look for a file called “README.md” and place its contents on the main page of the repository, like this:



The purpose of the README.md file is to give you a chance to document your code and have it placed nicely on the front page of your repository. This documentation will help you and your collaborators quickly describe the repository’s contents for anyone viewing it. A good README.md file should include the following at a minimum:

- The project’s purpose

- Description of the inputs / outputs
- Any special instructions on how to run the code

Other nice things to add to a README may include:[38]

- Table of contents if the README is lengthy
- Input / output examples
- Descriptions of the company / individuals who have contributed significantly
- Instructions on how to contribute
- Frequently asked questions
- Where to go for support
- Theory (if relevant)
- Known bugs
- Notes about this particular release of the code (see Section 4.3)

4.1.1 The Markdown Language

The ".md" file extension means that the file is written in "Markdown". This is a simple language for writing formatted text. This is useful for the README.md files because you can easily specify headers, code blocks, lists, etc. while creating your file. Here are some good ones to know, and an example showing all of these in action will be given below.

- Headers: To create a header, put a "#" in front of the text. To create sub-headers, use multiple "#" symbols.

- Normal Text: This is easy, just start typing on a new line. However, be sure to leave one blank line between paragraphs. Otherwise markdown will keep going on the same line without making a new paragraph.

- Bold and Italicized Text: To italicize text, put it between a pair of asterisks. To bold text, put it between two pairs of asterisks. These can be combined to make text that is both italicized and bolded.

- Code Block: For code blocks, put the code between pairs of "```". That's three of the symbols found generally on the same key as the "~" (tilde) symbol on your keyboard.

- Lists: For a bulleted list, preface each line with an asterisk or hyphen. For a numbered list, preface each line with a number.

- Images: There are multiple ways to add images in markdown. The author prefers to use the html syntax offered through markdown as "<img src = "file_path" width = pixels>". To

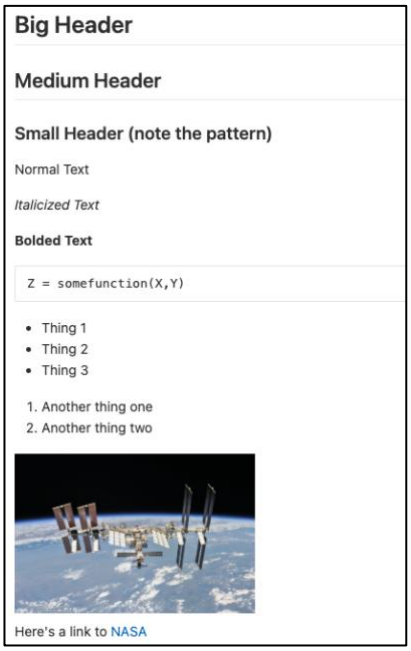
[38] https://gist.github.com/fvcproductions/1bfc2d4aecb01a834b46

do this, make sure that the file is accessible, and it should probably be stored somewhere in your repository so that it displays correctly for all viewers.

- Hyperlinks: To link to an external website, type “[link name](link directory)”.

Here are some examples. The raw markdown input is on the left and the final output is on the right.

```
# Big Header
## Medium Header
### Small Header (note the pattern)

Normal Text

*Italicized Text*

**Bolded Text**

```
Z = somefunction(X,Y)
```

* Thing 1
* Thing 2
* Thing 3

1. Another thing one
2. Another thing two

<img src="Images/ISS.png" width = 250>

Here's a link to [NASA](www.nasa.gov)
```



Hopefully you are beginning to see how nice these README.md files can be for documenting your code. They provide a simple format to create nice-looking documentation.

4.2 Ignoring Unwanted Files

There are many files that you may have in your repository folder but do not actually want to back up to Git. This may be because they are large data files or images or would unnecessarily clutter the remote repository if pushed online. Git provides a way to ignore such files from the repository while still letting you have them in the folder on your computer. The solution is to create a **.gitignore** file and place it in your repository. This is a file that you use to tell Git which files to ignore. A typical .gitignore file may look something like this:

```
*.bin
*.log
*.png
*.pdf
ID*
```

The "*" symbols represent any text. For example, any file that end in ".log" will be ignored from this particular repository. Additionally, any file that starts with "ID" will also be ignored from this repository.

Notice how the .gitignore file name begins with a period. This means that it will actually be a hidden file on your computer. Your computer and Git are very aware of its presence, but if you want to make any more changes you will need to enable hidden file viewing on your computer. The shortcut for this is shift+cmd+. on macOS. You may be surprised to find that there are actually a lot of hidden files on your computer. Use the same shortcut to disable this mode.

4.3 Using Tags

Tags are a way to create "releases" of your code. For example, if you had your code to a point where it was functional, but you know that you're going to keep working on it, you might give it the tag "v1.0.0". Git enables you and any other user to actually go and clone the repository at a particular tag, giving the files as they were at the time that tag was created. An application of this would be if you were creating a presentation or writing a paper for which you generated figures. Experience says that eventually you will want to go back and recreate those figures in the future. If you tagged your code at the time that you made your figures, then you can easily go and download the code exactly as it was when you generated those figures. It would also be wise to specify along with each tag exactly how the data were stored so you can re-use the same file system for your data, further minimizing the effort to regenerate those figures.

To create a tag, type:

```
git tag -a <tag> -m "Message"
```

For example, this could look like:

```
git tag -a v0.1.0 -m "First Release"
```

To see a list of all the tags in your repository just type:

```
git tag
```

To delete a tag, type:

```
git tag -d <tag>
```

Tags are not automatically pushed with a push command. To push a tag to a remote repository type:

```
git push origin <tag>
```

4.4 Submodules

Submodules are used to more or less embed a repository inside of another repository. A benefit of using submodules is that you can have some sort of "core files" repository for an overarching project and embed that "core files" repository inside of all of the other repositories in the overarching project. This way, you can ensure that everyone has a consistent file structure on their computers so that the code works universally. However, you may want to limit who has access to the "core files" so that you don't have any unexpected changes that will affect other projects. Using submodules to embed the "core files" into the other repositories lets the repositories access the "core files" without granting editing rights to the "core files" for team members working in the repository. As needed, you can release updated versions of the "core files" and have everyone update their versions of the repository to include this update to the "core files".

To add a submodule to a repository, navigate to the target repository on your terminal. Then type:

```
git submodule add <url/ssh>

git commit -m "added submodule"

git push
```

If changes have been made within the submodule (e.g. a new revision of "core files" has come out) then you can update that online by typing:

```
git submodule update --remote --merge

git push
```

You may need to commit the updates before pushing them. Now other team members can download the latest updates by opening their terminal and navigating to the repository on their computer and typing:

```
git pull
```

If they are cloning the repository then they need to change the clone command to account for the folders within the repository:

```
git clone --recursive <url/ssh>
```

If you want to add a submodule that in itself also has submodules, type:

```
git submodule add <urs/ssh>
```

```
git submodule update –init --recursive
```

To delete a submodule, first type:

```
git submodule deinit <path_to_submodule>
```

Then remove the leftover folder:

```
rm -rf <path_to_submodule>
```

Then delete the module from the .git folder:

```
rm -rf .git/modules/<name_of_submodule>
```

Now stage and commit your changes to Git. The submodule has been removed.

4.5 Rebasing

4.6 Terminal Configuration

4.6.1 General Terminal Configuration

If you are using macOS Catalina or higher, you will have the zsh shell for your terminal. The syntax for the commands is almost always identical to the previous bash shell. In order to access the configuration file for zsh, type:

```
open ~/.zshrc
```

This will open the zsh configuration file. If it says that the file does not exist, then type:

```
touch ~/.zshrc
```

and then open the file.[39] This is where you can now add special configuration settings to your terminal shell.

4.6.1.1 Creating Aliases

To create an alias, open the .zshrc file and type the alias you desire. An example of this could be:

```
alias ls="ls -GF"
```

This command makes it so that every time you type "ls", the computer actually will do "ls -GF", which will make the output prettier and easier to read. Your computer might be picky and require that there be no spaces on either side of the equals sign. Another example could be:

```
alias matlab="/Applications/MATLAB_R2020a.app/bin/matlab -nojvm -nodesktop"
```

This command actually opens MATLAB within your terminal environment, which is handy, but we will leave our discussion at that.

4.6.2 Git-specific Configuration

Similar to the .zshrc file that holds the configuration data for your terminal shell, the .gitconfig file holds configuration information for Git. You can create aliases, select merge and diff tools, and other handy tasks. To access this file, type:

```
open ~/.gitconfig
```

If this file doesn't exist, then type:

```
touch ~/.gitconfig
```

and then you can open it. An example of what this configuration file might look like after you've customized it is:

[39] Files that begin with a period, like ".zshrc" are hidden files on your computer. On macOS, type Shift+cmd+. to show hidden files. When using the "ls" command, if you want to show hidden files, type "ls -a".

```
[user]
        name = Mark-C-Anderson
        email = anderson.mark.az@gmail.com
[core]
        editor = code -w
[alias]
        hist = log --all --graph --decorate --oneline
[diff]
        tool = vscode
[difftool "vscode"]
        cmd = code --wait --diff $LOCAL $REMOTE
[merge]
        tool = vscode
[mergetool "vscode"]
        cmd = code --wait $MERGED $LOCAL $REMOTE
[difftool]
        prompt = false
[mergetool]
        prompt = false
```

This section will likely be improved on in the future to provide a more detailed review of configuration files.

4.7 Using Git with Other Software

4.7.1 MATLAB

If you have the Git terminal/command line tools installed on your computer, then MATLAB will automatically allow you to interface with Git through the default MATLAB command window. To test this, open MATLAB and type in the command window:

```
!git
```

If Git is installed on your system, MATLAB will automatically display a list of possible Git commands. If you do not have Git installed on your computer, then you will need to install it as described in Section 2.2.2. Once you have Git properly installed, you can type any git command into the MATLAB command window as long as you preface each command with an exclamation mark. For example,

```
!git status
```

will return the current status of your repository, exactly as the “git status” command would if you were using Git in the terminal.

Additionally, if you look at the “Current Folder” window in MATLAB you will see that each file has a colored shape next to it that tells you the current status of that file. Hover over the shape with your mouse to see what each one means.

One neat way to interface Git with your MATLAB is to use MATPack,[40] which is a MATLAB package manager developed by the author to provide a package management and usage system like the Python and Julia programming languages.

[40] https://github.com/AerospaceMark/MATPack

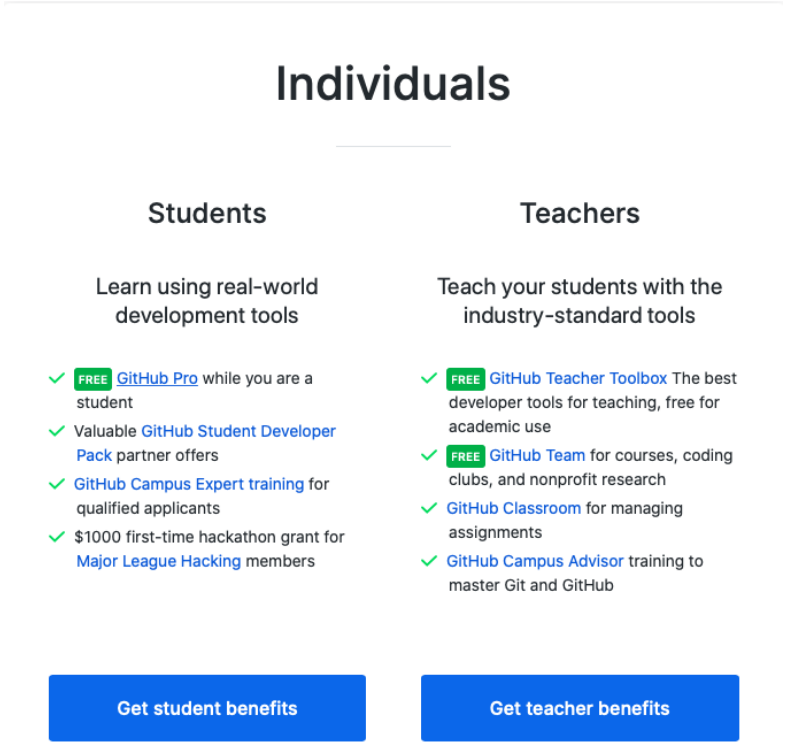
5 Appendix

5.1 Student Upgrade to GitKraken Pro

As a student, you can get a free upgrade to GitKraken Pro by associating your GitHub account with your personal university email address.[41] This gives you access to the GitHub Student Developer Pack.

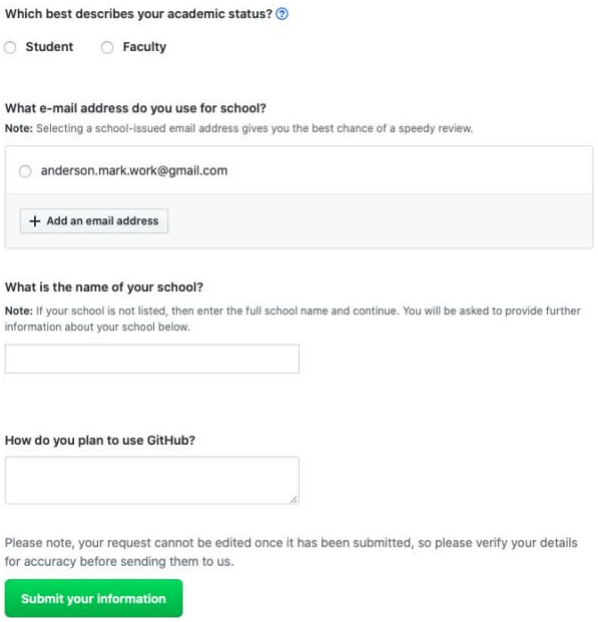
5.1.1 Getting the GitHub Student Developer Pack

Go to https://education.github.com/pack and select the blue "Get the Pack" icon. This will bring up the following page:

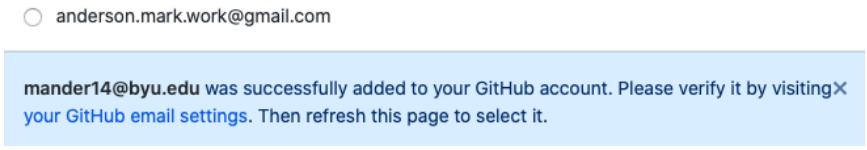


[41] GitHub only allows you to get the student developer pack once, so make sure that you get it attached to the GitHub account that you plan on using for the rest of your time as a student. Even if you make your GitHub account for the sole purpose of getting the student developer pack, just be aware that if you want to access its features in the future it must be through this account.

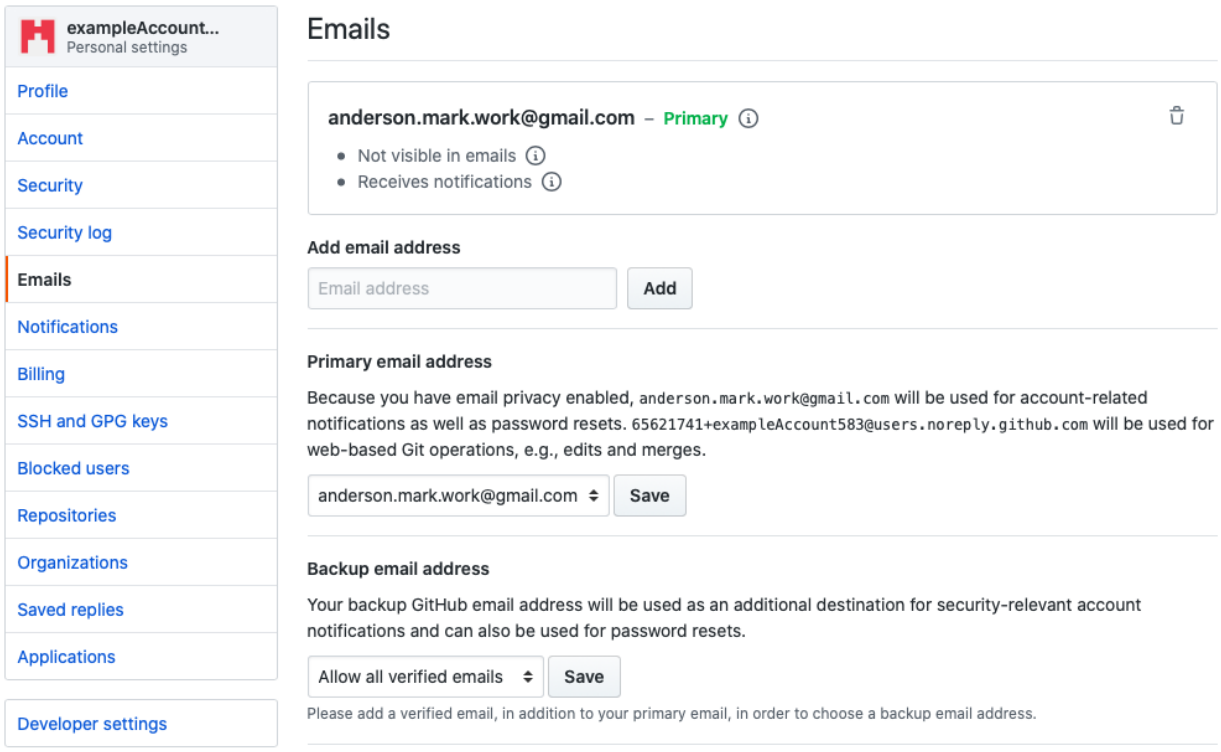
Select “Get student benefits”. You will see the following:



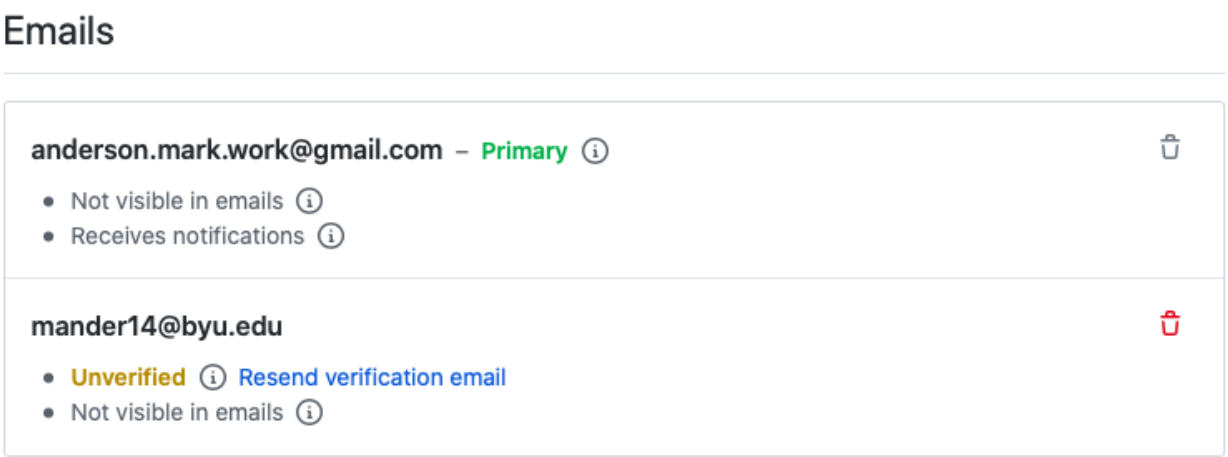
Add an email address, you may see the following after adding it:



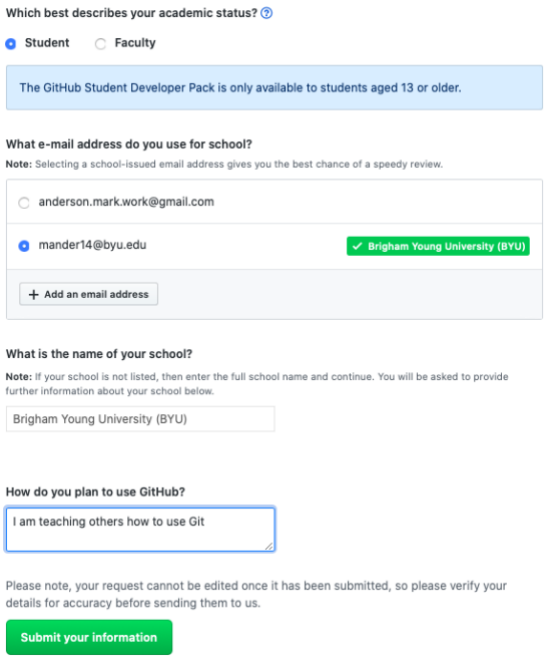
Right-click on the “your GitHub email settings” text and open it in a new tab. You will be brought to the following page:



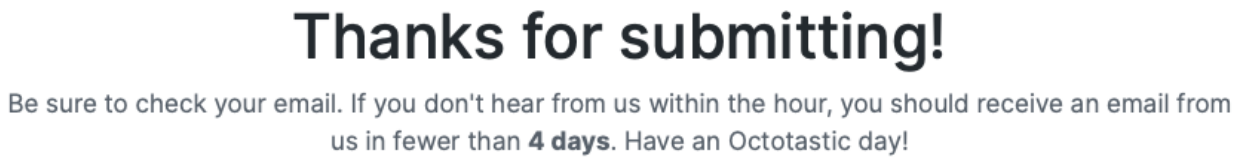
Add your school email address in the “Add email address” field. You will see the following at the top of the page:



GitHub will send a verification email to your school email address. Once you verify the email address, you’re ready to get the student developer pack. Return to the tab for the student developer pack, or just re-navigate to it on https://education.github.com/pack. Once filled out, the application should look something like this:



Click on the green "Submit your information" icon[42] and you will see the following notice on the next page:

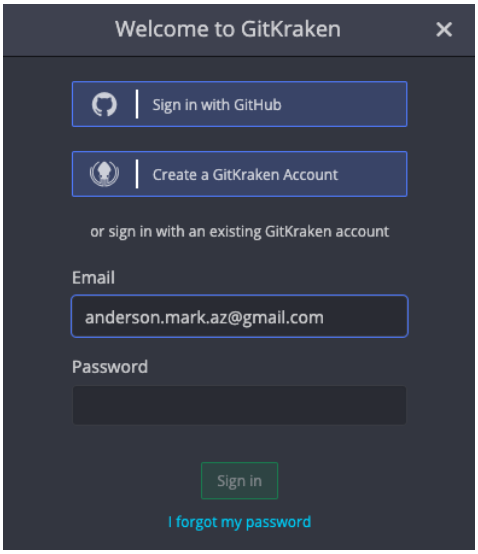


It looks like it can take a while for your request to be accepted, but while making this tutorial the email saying that the offer was accepted was received about a minute after submitting. The email will have lots of information for you to explore. Welcome to the student developer pack.

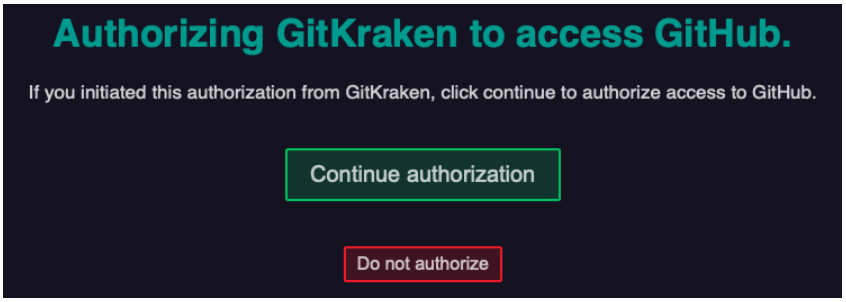
5.1.2 Getting GitKraken Pro Through the Student Developer Pack

First, sign into GitHub on your browser. After this, open GitKraken and go to "File > Sign into a different account". You will see the following popup:

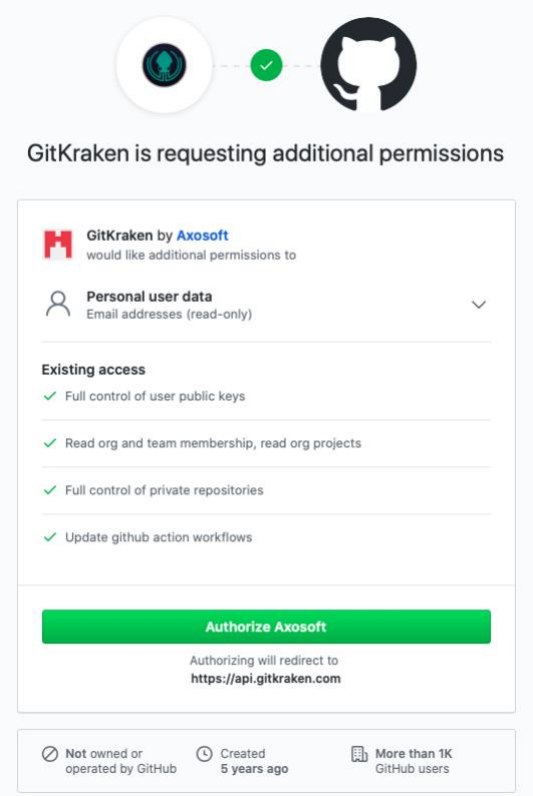
[42] GitHub may also ask for some sort of proof of your academic status. This may involve taking a picture of your student ID, uploading a transcript, or another form of proof.



Select the “Sign in with GitHub” option. You’ll be brought to the following web page:



Click on the green “Continue authorization” icon. You will be brought to the following page:



Click on the green “Authorize Axosoft” icon (Axosoft is the company that makes GitKraken).

You may have to wait a few minutes for the authorization to be successful, but once it is successful you will see a screen that says it was successful. Afterwards, open GitKraken and in the bottom right corner you should see something like this:

Where the orange “FREE” label has now been replaced with the orange “PRO” label. While making this tutorial it was about thirty minutes before the label changed to “PRO” You may need to restart GitKraken. Congratulations, you now have a Pro version of GitKraken. As of the present writing, this Pro version for students is advertised as a one-year deal.

5.2 Further Learning

5.2.1 Git in General

5.2.1.1 Udemy.com

Udemy is an extremely useful website that provides online video tutorials on a huge array of topics ranging from playing the guitar to in-depth MATLAB tutorials on machine learning. I learned the bulk of how to use Git from a single online class. This class teaches exclusively in the terminal and is about six hours long. The teacher is extremely knowledgeable and teaches in great clarity, explaining every command. By the end of this class you will be able to use Git very comfortably in the terminal. Here is the link to the class:

https://www.udemy.com/share/101tLwAEIccF9TR3wF/

If the price seems high, don't stress. Udemy seems to do deals about every other week where the classes are offered for around $10. Just check back daily until you see that it's on sale.

5.2.1.2 Other

https://www.youtube.com/watch?v=uR6G2v_WsRA – brings up committing almost like saving your place in a video game

https://www.youtube.com/watch?v=1h9_cB9mPT8 – Git for noobs

http://try.github.io – interactive tutorial

http://blog.martinfenner.org/2014/08/25/using-microsoft-word-with-git/ - Using git with word

https://michaelstepner.com/blog/git-vs-dropbox/ - git for researchers

https://neurathsboat.blog/post/git-intro/ - git for scientists

https://milesmcbain.github.io/git_4_sci/index.html - git course for scientists

https://biz30.timedoctor.com/git-mecurial-and-cvs-comparison-of-svn-software/ - comparison between mercurial, SVN, and Git

http://blogs.nature.com/naturejobs/2018/06/11/git-the-reproducibility-tool-scientists-love-to-hate/ - Git's hard at first, but after that it's worth it. Includes links to tutorials

https://codeburst.io/number-one-piece-of-advice-for-new-developers-ddd08abc8bfa (New Developer? Learn Git)

https://gist.github.com/myusuf3/7f645819ded92bda6677 (Removing Submodules)

5.2.2 GitHub

https://www.youtube.com/watch?v=-YVIpI4ucQw
https://www.youtube.com/watch?v=w3jLJU7DT5E
https://www.youtube.com/watch?v=BCQHnlnPusY

5.2.3 GitLab

https://www.youtube.com/watch?v=Jt4Z1vwtXT0

5.2.4 GitKraken

GitKraken has a whole YouTube channel where they both teach you Git and help you learn how to use GitKraken. The videos are very well-made and are found at:

https://www.youtube.com/channel/UCp06FAzrFalo3txskS1gCfA

5.2.5 The Terminal

https://www.youtube.com/watch?v=oK8EvVeVltE&list=PLRqwX-V7Uu6ZF9C0YMKuns9sLDzK6zoiV&t=0s