# **What is Git?**

Git is a version control system that we download onto our computer. It allows us to track the history of a project and collaborate with other people.

## **GIT TRACKS THE HISTORY OF A PROJECT**

To illustrate how Git tracks the history of a project let’s consider the example of writing a book in Example Book Project 1-1.

### **EXAMPLE BOOK PROJECT 1-1**

Let’s assume we are writing a book and we want to use Git to version control all the files in our book project.

Every time we make changes to the book we can save a version of it using Git.

Suppose we make changes to the book on Monday, Wednesday, and Friday, and we save a version on each of those days. This means we have at least three versions of our project. A version of a project in Git is called a commit. In Chapter 2, we will learn more about commits. For now, all we need to know is that in our example we have at least three commits.

These three commits allow us to look at the version of the book that we had at the end of our Monday work session, Wednesday work session, and Friday work session.

Git also allows us to compare any of those saved versions of our book project to one another to check what changed in between the different versions.

As we can see in our example above, using Git is a great way to track the history of our project.

## **GIT IS A TOOL FOR COLLABORATION**

Git also allows us to work on a project together with other people. Let’s go over an example of this in Example Book Project 1-2.

### EXAMPLE BOOK PROJECT 1-2

If a friend wants to help us write a book, Git allows us to work on the same project at the same time and combine our work when we are ready.

For example, we can work on chapter 1 and our friend can work on chapter 2. When we are ready we can combine the work we have done.

If we get an editor to review the book, they can also make edits to all the chapters of the book we have written. And we can later combine the editor’s changes to the main version of the book as well.

An unlimited number of people can work together on a project version controlled by Git.

Therefore, in addition to tracking the history of a project, we can see in the example above that Git is also a useful tool for collaboration.

## **Installing or updating Git**

To work with Git, we must first have it installed on our computer. Even if we already have it installed it is a good idea to update our version of Git to the latest version. This is because Git, just like any modern technology, is under constant development and undergoes updates and changes over time.

Let’s go ahead to Follow Along 1-1 to make sure we have Git installed or updated on our computer.

[ FOLLOW ALONG 1-1 ]

Go to https://git-scm.com/downloads to install or update Git for your computer. The preferred method of installation will depend on your operating system and on the options provided by the Git community for the latest version.

Now that we have installed or updated Git on our computer, let’s look at how we will be interacting with Git.

The graphical user interface and the command line

Throughout this learning journey we will use both the graphical user interface and the command line.

The graphical user interface (GUI) is the icons, buttons, and other graphical representations of objects that allows us to interact with our computer. You can think of it as the point and click interface. For example, the folders on your desktop are part of your computer’s GUI.

The command line (also known as CLI, Terminal, bash, or shell) is a place where we can type text-based commands to interact with our computer.

See Figure 1-1 for a side-by-side comparison of the GUI and the command line.

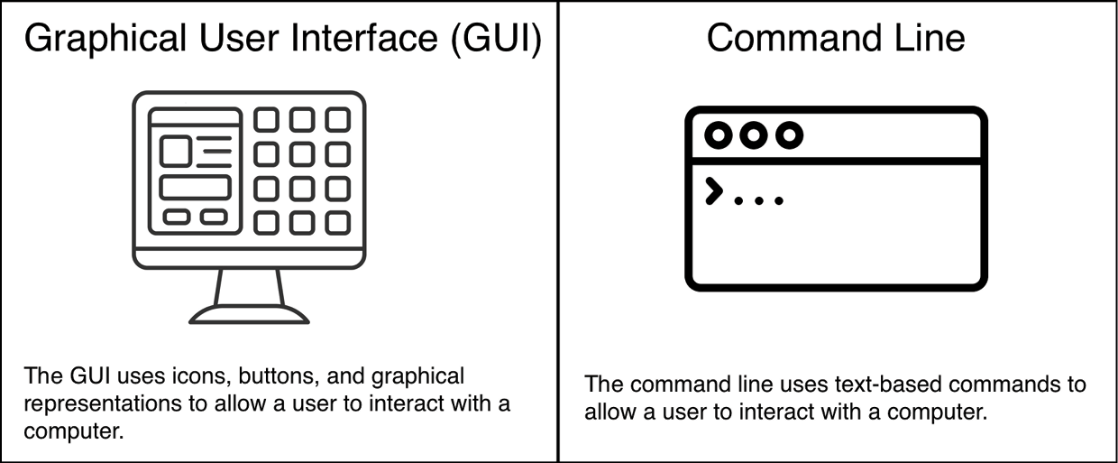


FIGURE 1-1

The graphical user interface (GUI).and the command line are two different ways of interacting with your computer.

Next, let’s take a closer look at the command line.

## **Looking at the command line**

The command line application you will use will depend on the type of operating system you have:

If you have a Mac, your command line application is called Terminal.

If you have a Windows computer, your command line application is called Git Bash.

[ NOTE ]

The Git Bash command line application is only available if you have Git installed on your computer.

When we open the command line window there will be a command prompt. The command prompt in both the Terminal and Git Bash is a short text string that ends in a dollar sign and indicates where we need to enter our commands. The command prompt by default indicates the directory location in the command line and when we open a new command line window it starts off at the current user directory (also known as the home folder) which is represented by the tilde sign “~”. After the command prompt there is a cursor which indicates where we are typing in the command line.

The contents of the command prompt differ depending on your operating system and computer settings, for our purposes the only important part of the command prompt that we have to identify is where it shows the directory location. See Figure #-# for an annotated example of a generic command prompt.

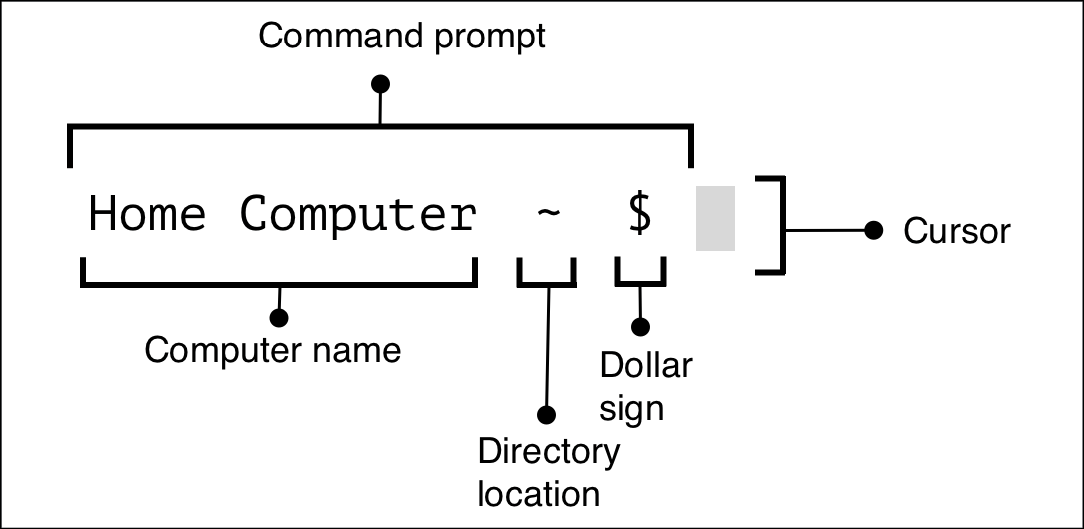


FIGURE 1-2

An example of a command prompt

Now let’s go ahead to Follow Along 1-2 and open the command line application we will be using throughout this book.

[ FOLLOW ALONG 1-2 ]

Find your command line application and open it.

What to notice:

The command prompt indicates the directory location and by default ends in a dollar sign.

Now that we have installed Git on our computer and opened the application that we will be using as our command line, let’s execute our first command in the command line.

## **Executing our first command in the command line**

The command prompt in our command line application ends in a dollar sign ($). In the Terminal by default the dollar sign is on the same line as the command prompt while in Git Bash it is on the line below. Whatever text we type after the dollar sign is what the computer will interpret as a command. To execute the command, we must press the Enter key.

When a step in a Follow Along section in this book includes a command to execute in the command line, the command to execute will be in bold right after the dollar sign ($). If the command produces output, it will be shown below it (not in bold). If the command is supposed to be executed in a directory other than the current user directory, then we will indicate the directory location before the dollar sign.

In Figure #-#, we show an example of what a command in a Follow Along looks like. The example shows that we would execute the pwd command while being in the desktop directory. We will cover what the pwd command does later in this chapter. For now, it is only important to be able to identify where it shows the directory location, the command to execute, and the output.

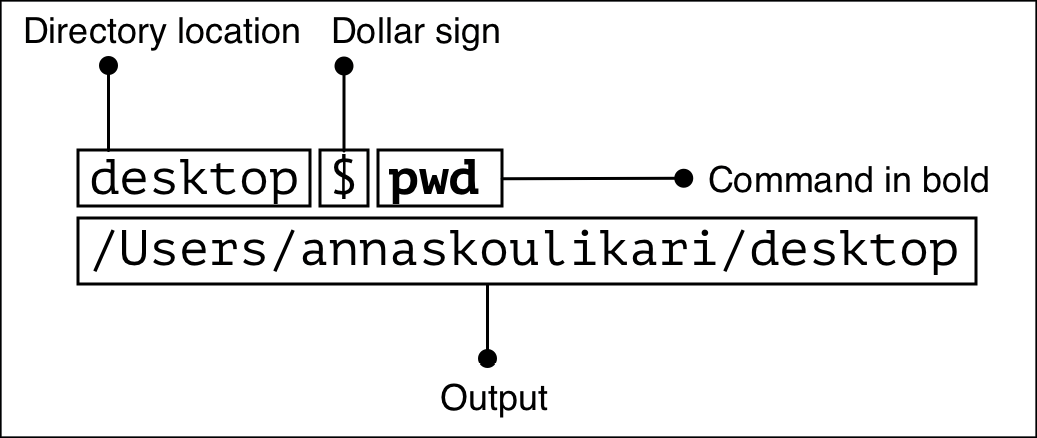


FIGURE 1-3

How to execute command included in a Follow Along.

Keep in mind that some commands produce output and some don’t. When a command does produce output, it may not always be exactly like the output printed in this book. This may be because of one of two main reasons:

1. You have done something differently in your project than what was described in the instructions.
2. Git has changed the output of the command you have executed. This may happen because, as we mentioned earlier in this chapter, Git is a technology that undergoes updates and is constantly being improved.

It may sometimes be important to read and understand the output of a command to know whether you are in situation A or B.

## **Command options and arguments**

Sometimes we will use commands with options and/or arguments. Options are like settings that change the behavior of a command. An option follows a single dash “-” or a double dash “- -”.

Arguments are values that provide information to the command. They will be denoted by angle brackets (< >). We will have to pass in a value for the argument without including the angle brackets.

An example of a command with an option and an argument that we will use is git commit -m “<message>”. In this example, -m is the option and <message> is the argument as shown in Figure #-#. We will cover what this command does in Chapter 3.

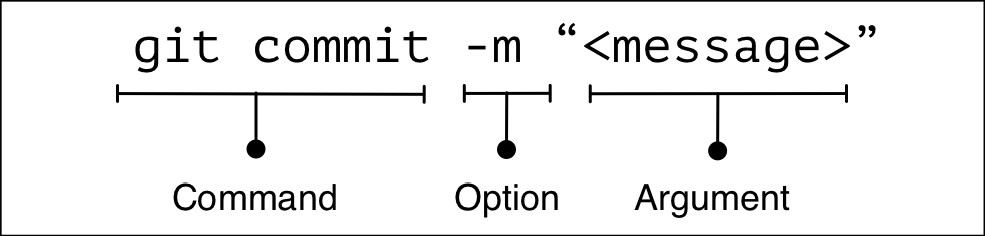


FIGURE 1-4

An example of a command with an option and an argument

From now on, whenever we encounter a command in a Follow Along exercise, we will enter the command in bold in the command line and execute it. Our first example of this is in step 1 of Follow Along 1-3.

[ FOLLOW ALONG 1-3 ]

$ git --version

git version 2.25.1

What to notice:

In the output of the git --version command, we see what version of Git is installed on our computer.

[ NOTE ]

As we mentioned previously, Git is constantly being updated therefore your version of Git may be different than the one displayed in this book.

If the git --version command returns that you don’t have Git installed, return to Follow Along #.# and follow the steps to install Git.

We have now covered how to enter commands in the command line, the next thing to learn is how to clear them from the command line as well.

## **Clearing the command line**

Every time we enter a command in the command line window, it will be listed right under the previous command we had entered.

After entering a lot of commands, the command line window may get quite cluttered.

To clear the command line window we use the clear command.

[ SAVE THE COMMAND ]

Clear

Clear the command line screen

The clear command clears what is in the window however it does not clear the history. If we scroll up in the command line then we will be able to see our previous commands and output. Let’s practice using this command in Follow Along 1-4.

[ FOLLOW ALONG 1-4 ]

$ Clear

We’ve covered entering commands and clearing them from the command line window. Now it’s time to prepare the next tool that will help us in our learning journey, the file system window.

Opening the file system window

The file system window is part of our GUI. Throughout our learning journey we will interact with both our file system window and our command line window which will make our learning experience more tangible. Therefore it is useful to have both a command line window and a file system window open side-by-side on our computer screen as shown in Figure 1-5.

A picture containing rectangle, screenshot, sketch, diagram

Description automatically generated

FIGURE 1-5

And example of a computer with a file system window next to a command line window.

The file system application you will use will depend on your operating system:

Mac: the file system application is Finder.

Windows: the file system application is File Explorer.

Let’s prepare this setup on our computer in Follow-Along 1-5.

[ FOLLOW ALONG 1-5 ]

Find your file system application and open up a file system window next to your command line widow.

We now have a view of our command line window and file system window ready, so let’s continue learning some command line basics.

Identifying our directory ­location in the command line

At any given point in time in a command line window we are in one directory. Remember that the term directory simply means folder. As we mentioned previously, when we first open the command line application (assuming we haven’t changed any default settings) our command line will start off at our current user directory, indicated by the tilde “~” sign.

We can also use the pwd command, which stands for “print working directory”, to see which directory we are in and the path to that directory.

[ SAVE THE COMMAND ]

pwd

Show the path of the current working directory.

[ FOLLOW ALONG 1-6 ]

$ pwd

/users/annaskoulikari

What to notice:

We are in our current user directory.

The output of the pwd command prints the path to the directory we are in. In Follow-Along #-#, /Users/annaskoulikari is an example of a path. My name is Anna Skoulikari and annaskoulikari is my username on my computer. Users and annaskoulikari are two directories. Directories in a path are separated by a slash (/). The annaskoulikari directory is inside the Users directory.

It is important to know our directory location because most commands show us information or affect the directory we are in when we execute them. Also knowing our directory location helps us with navigating through our file system which we will cover later on in this chapter.

[ NOTE ]

If you want the directory, you are at in the file system window you opened in the previous section to be the same directory you are at in the command line window right now, then in the file system window, find the Users directory and then inside that directory find the current user directory which will be your username and open it.

Now that we have covered how to identify which directory we are in, let’s explore how to view the actual contents of a directory.

## Viewing the contents of directories

Before we go over how to view the contents of a directory in the GUI and the command line, we should introduce two types of files that exist in our file system: visible files and hidden files.

Visible files are always visible in our file system.

Hidden files are only visible in our file system if we change our settings to view them. They are often files that store information that we, as users, don’t need to access such as app configurations and various settings.

We do not recommend modifying or deleting hidden files (unless you really know what you’re doing). Once we change our settings to view hidden files, they appear partially transparent (grayed out) and their filename often starts with a dot “.”, as seen in Figure #-#.

A picture containing text, screenshot, font, line

Description automatically generated

In our Git learning journey there are some important hidden files we will want to be aware of so we need to know how to view hidden files both in the GUI and the command line.

In the GUI, to view hidden files in a file system window we have to explicitly make them visible:

On a Mac we have to select Control key + Shift key + dot

On Windows we have to go alter the file settings. For Windows users, a quick search online will give you the exact instructions on how to show hidden files on your computer.

Now, let’s move onto Follow Along 1-7 to carry this out.

[ FOLLOW ALONG 1-7 ]

Make hidden files visible in your system.

In the command line, to view the visible files of the folder we are currently in we can use the ls command (which stands for list). This will show us a list of the visible files in our current folder.

To view both visible and hidden files in our current folder we use the ls command with the -a option, therefore ls -a.

[ SAVE THE COMMAND ]

ls

List visible files

ls -a

List hidden and visible files

Let’s practice using these commands to list different kinds of files in Follow Along 1-8.

What to notice:

The names of many hidden files start with a dot “.”. For example, .gitconfig is a hidden file that may be shown in your ls -a output. We will talk more about this file at the end of this chapter.

Now that we have covered how to identify which directory we are in and view the contents of our current directory in the command line, let’s explore how to go into and out of directories.

Navigating into and out of a directory

In the GUI, to go into a directory we can double click on it.

In the command line, to go into a directory, we use the cd command, which stands for “change directory”, and pass in the path to the directory.

[ SAVE THE COMMAND ]

cd <path\_to\_directory>

Change directory

Let’s practice by navigating into our desktop directory in Follow Along 1-9.

[ FOLLOW ALONG 1-9 ]

$ cd desktop

desktop $ pwd

/Users/annaskoulikari/desktop

What to notice:

The cd command does not produce any output.

In step 2, the command prompt and the pwd command output indicate that we are in the desktop directory.

[ NOTE ]

Navigating into and out of directories in the command line does not affect what we are viewing in the file system. For example, navigagting into the desktop directory in the command line will not automatically cause our file system to display the contents of the desktop directory.

[ NOTE ]

Navigating into and out of directories in the command line does not affect what we are viewing in the file system. For example, navigating into the desktop directory in the command line will not automatically cause our file system to display the contents of the desktop directory.

In the GUI, to go back to the parent directory we can select a back button.

In the command line, to go back to the parent directory we can pass in two dots (..) to the cd command. Two dots represent the parent directory of the directory we are currently in. Let’s try this out in Follow Along 1-10.

[ FOLLOW ALONG 1-10 ]

desktop $ cd . .

$ pwd

/Users/annaskoulikari

What to notice:

In step 2, the command prompt and the pwd output indicate that we are back in the current user directory.

Now that we have covered how to go into and out of directories, let’s learn how to make them in the first place.

## **Making a directory**

In the GUI, we can make a directory by right clicking or selecting the relevant option or icon.

In the command line, to make a directory we will use a command called mkdir which stands for “make directory”. The directory will be made inside the directory we are in when we execute the command.

[ SAVE THE COMMAND ]

mkdir <directory\_name>

Make a directory

To keep things simple we will not include spaces in our directory name. If a directory name contains spaces, then we have to make modifications to how we use certain commands in the command line which makes our tasks more complicated.

As we mentioned in the Preface, throughout this book we will work on one project in which we will list the colors of the rainbow as well as some colors that are not part of the rainbow. This is not a very realistic example of a project typically version controlled with Git. However, it is a simplified example that will allow us to focus on learning how Git works.

Since the main objective of our sample project is to list the colors of the rainbow, we will give our project directory the name rainbow.

We will create this project directory in our desktop directory so that we can easily see it from our desktop on our computer screen. Now, let’s move onto Follow Along 1-11 to make this happen.

[ FOLLOW ALONG 1-11 ]

$ cd desktop

desktop $ mkdir rainbow

desktop $ ls

rainbow

What to notice:

In step 3, the ls output shows the rainbow project directory we just created along with any other directory or files we have in our desktop directory.

If we look on the desktop of our computer we should identify the rainbow project directory we just made. We illustrate this in Visualize It 1-1.

[ VISUALIZE IT 1-1 ]

A picture containing screenshot, rectangle, diagram, text

Description automatically generated

We made a project folder called rainbow on our desktop.

We made a project folder called rainbow on our desktop.

Next, we notice that just because we made the rainbow project directory does not mean we navigated into it. We need to explicitly navigate into the directory in the command line, let’s do this in Follow Along 1-12.

[ FOLLOW ALONG 1-12 ]

desktop $ cd rainbow

rainbow $ pwd

/Users/annaskouikari/desktop/rainbow

What to notice:

In step 2, the command prompt and the output from the pwd command indicate we are in the rainbow directory.

We have created and navigated into our rainbow directory. Now, what happens if we close our command line window and open it back up again?

Closing the command line

By default, if we close the command line window and then open it up again, the file location it will open at will be the current user directory. Therefore we will have to navigate to the directory we want to work within again.

Let’s test this out in Follow Along 1-13.

[ FOLLOW ALONG 1-13 ]

rainbow $ pwd

/Users/annaskouikari/desktop/rainbow

Close your command line window and then open a new command line window.

$ pwd

/Users/annaskouikari

$ cd desktop/rainbow

What to notice:

In step 3, the output of the pwd command indicates that we are back in our current user directory.

All right, at this point we have covered several command line basics. The final step to get ready to start working with Git is to set some basic Git configurations.

SETTING GIT CONFIGURATION

Git configurations are Git settings which allow us to customize how Git works. They consist of variables and their values and they are stored in a couple different files.

In our example, we will set variables in our global Git configuration file which is a hidden file called .gitconfig which can usually be found in our current user directory.

To do so, we will use the git config command and set variables for our user settings as well as a default branch name (if you don’t yet know what branches are, don’t worry, we will learn more about them in Chapter 4).

First, let’s view the contents of our global Git configuration file. To do so we pass the --global and the --list options to the git config command.

[ SAVE THE COMMAND ]

git config --global --list

List variables in global Git configuration.

Let’s try this out in Follow Along 1-14.

[ FOLLOW ALONG 1-14 ]

What to notice:

In our example, we are assuming that we have not set our user settings or any other global configuration variables and that is why the git config --global --list output in Follow Along #.# is empty. However, you will most likely have some global variables set therefore your output may be different.

The output from step 1 will be the same as the content in the .gitconfig file in step 2.

rainbow $ git config --global --list

Optionally, open the .gitconfig file in your current user directory.

For our purposes, we are only interested in the user.name, user.email, and init.defaultBranch variables.

First, let’s look at the user.name and user.email variables.

## **User settings**

The user.name and user.email variables are used so that when we are saving versions of our project (or in other words when we are making commits) Git will know who saved that work.

If in the output of Follow Along #-#, these variables are set to the values you want them to be then you may skip Follow Along #.#. If these variables do not appear or are not set to the values you want them to be then carry on to Follow Along #-#.

To set these variables we will pass them as arguments to the git config command and then enter our desired values for them inside quotation marks.

[ SAVE THE COMMAND ]

git config --global user.name “<name>”

Set the name in the global Git configuration

git config --global user.email “<email>”

Set the email in the global Git configuration

Decide what name and email you want to associate with the work you do on your project and continue onto Follow Along 1-15.

[ FOLLOW ALONG 1-15 ]

rainbow $ git config --global user.name “annaskoulikari”

rainbow $ git config --global user.email “annaskoulikari@gmail.com”

rainbow $ git config --global --list

user.name=annaskoulikari

[user.email=annaskoulikari@gmail.com](mailto:user.email=annaskoulikari@gmail.com)

What to notice:

In step 3, in the git config --global --list output, the user.name and user.email variables are set to the values you entered.

[ NOTE ]

If the values for the user.name and user.email variables are not visible in step 3 of Follow Along #.# or they are set to the wrong values, then there may have been an error in steps 1 or 2. You may need to repeat those steps.

## **Default branch configuration**

The next Git configuration variable we want to look at is init.defaultBranch. We will cover what branches in more depth in Chapter 4. For now, all you need to know is that by default Git will create a branch called master when we initialize a new repository. From Git version 2.28 onwards, we can set a different name for the initial branch. We will cover the reason to set a different name in the “A bit of Git history: master and main” section in Chapter 4.

For our project, we are going to set the initial default branch name to main. To do so we will pass the init.defaultBranch variable as an argument to the git config command and the desired branch name in quotation marks.

[ SAVE THE COMMAN D ]

git config --global init.defaultBranch “<branch\_name”>

Set the default branch name in the global Git configuration.

Let’s see this in action in Follow Along 1-16.

[ FOLLOW ALONG 1-16 ]

rainbow $ git config --global init.defaultBranch main

rainbow $ git config --global --list

user.name=annaskoulikari

user.email=annaskoulikari@gmail.com

init.defaultbranch=main

Optionally, open the .gitconfig file in your current user directory.

What to notice:

In step 2, in the git config --global --list output, the init.defaultBranch variable is set to main.

In step 3, the .gitconfig file includes the variables and values that we have set in this section.

SUMMARY

In this chapter, we went over what Git is and discussed how it is a useful tool for tracking the history of a project and collaborating with others. We prepared ourselves for working on a project using Git by installing or updating Git on our computer and later setting some basic Git configuration variables.

We also learnt some command line basics such as how to view the contents of directories, how to navigate into and out of them, and how to make them in the first place. We ended up making the project directory that we will be using throughout this learning journey called rainbow.

Now, we’re ready to move onto Chapter 2 where we will actually turn our rainbow project directory into a Git repository and start learning about the most important areas when working with Git.