**Productivity Tools**

with Prof. Rafael Irizarry

**Jesse’s note**Course Notes and Screenshots

HarvardX

**Jesse’s note**Course Notes and Screenshots

HarvardX

**Table of Contents**

[with Prof. Rafael Irizarry 0](file:///C:\Users\Jesse's%20Beast\Documents\HarvardX%20classes\Productivity%20Tools.docx#_Toc24885269)

[1. Section 1: Installing Software 4](#_Toc24885270)

[1.1. Keeping Organized with RStudio Projects 4](#_Toc24885271)

[1.2. Introduction to Git and GitHub 7](#_Toc24885272)

[1.2.1. Installing Git Introduction 7](#_Toc24885273)

[1.2.2. Installing Git 7](#_Toc24885274)

[1.2.3. GitHub 10](#_Toc24885275)

[1.2.4. GitHub Repositories 11](#_Toc24885276)

[1.2.5. RStudio, Git, and GitHub 13](#_Toc24885277)

[2. Basic Unix 18](#_Toc24885278)

[2.1. Introduction to Unix 18](#_Toc24885279)

[2.1.1. The Terminal 18](#_Toc24885280)

[2.1.2. The Filesystem 19](#_Toc24885281)

[2.1.3. Working Directory 21](#_Toc24885282)

[2.2. Working with Unix 22](#_Toc24885283)

[2.2.1. Unix Commands 22](#_Toc24885284)

[2.2.2. mv and rm: Moving and Removing Files 25](#_Toc24885285)

[2.2.3. less: Looking at a File 26](#_Toc24885286)

[2.2.4. Preparing for a Data Science Project 26](#_Toc24885287)

[3. Reproducible Reports 30](#_Toc24885288)

[3.1. Reproducible Reports 30](#_Toc24885289)

[3.1.1. Reproducible Reports with R Markdown 30](#_Toc24885290)

[3.1.2. R Markdown 30](#_Toc24885291)

[3.1.3. Knitr 31](#_Toc24885292)

[4. Section 4: Git and GitHub 33](#_Toc24885293)

[4.1. Git and GitHub 33](#_Toc24885294)

[4.1.1. Git and GitHub 33](#_Toc24885295)

[4.1.2. Using Git at the Command Line 34](#_Toc24885296)

[4.1.3. Creating a GitHub Repository 39](#_Toc24885297)

[5. Section 5: Advanced Unix 41](#_Toc24885298)

[5.1. Advanced Unix Part 1 41](#_Toc24885299)

[5.1.1. Advanced Unix: Arguments 41](#_Toc24885300)

[5.1.2. Advanced Unix: Getting Help and Pipes 41](#_Toc24885301)

[5.1.3. Advanced Unix: Wild cards 42](#_Toc24885302)

[5.2. Advanced Unix Part 2 42](#_Toc24885303)

[5.2.1. Advanced Unix: Environment Variables and Shells 42](#_Toc24885304)

[5.2.2. Advanced Unix: Executables, Permissions, and File Types 43](#_Toc24885305)

[5.2.3. Advanced Unix: Commands You Should Learn 45](#_Toc24885306)

**List of Figures**

[Figure 1. Select File and New project in RStudio to start a new project 4](#_Toc24808337)

[Figure 2. Select New Directory 4](#_Toc24808338)

[Figure 3. Choose a Project Name (that will be the Directory Name) and a location on your computer 5](#_Toc24808339)

[Figure 4. It creates a folder with a file my-first-project.rproj 5](#_Toc24808340)

[Figure 5. The project you working in is displayed at the top in RStudio 6](#_Toc24808341)

[Figure 6. Folder of our project with the code we saved: 6](#_Toc24808342)

[Figure 7. When you double click on the rproj file it opens RStudio with the file we were editing 7](#_Toc24808343)

[Figure 8. Choose default editor used by Git 7](#_Toc24808344)

[Figure 9. Installing git Bash 8](#_Toc24808345)

[Figure 10. RStudio using Git Bash 8](#_Toc24808346)

[Figure 11. Terminal App in Mac OS 9](#_Toc24808347)

[Figure 12. GitHub account creation page 10](#_Toc24808348)

[Figure 13. Global option in Rstudios 10](#_Toc24808349)

[Figure 14. Git/SVN global options 11](#_Toc24808350)

[Figure 15. Selecting new GitHub repositories 12](#_Toc24808351)

[Figure 16. Creating new GitHub repo 12](#_Toc24808352)

[Figure 17. Address to connect to the repo 13](#_Toc24808353)

[Figure 18. New project in RStudio with Git 13](#_Toc24808354)

[Figure 19. Git project in RStudio 14](#_Toc24808355)

[Figure 20. Adding a file to the Git tab: 14](#_Toc24808356)

[Figure 21. Committing window 15](#_Toc24808357)

[Figure 22. Summary of submitted commit 16](#_Toc24808358)

[Figure 23. Pushing the changes to the GitHub repo 16](#_Toc24808359)

[Figure 24. GitHub repo after push 17](#_Toc24808360)

[Figure 25. Git Bash terminal the “|” represent the command line 18](#_Toc24808361)

[Figure 26. Example of file system 19](#_Toc24808362)

[Figure 27. Example of a directory “Docs” with 2 sub directories “reports” and “resume” 19](#_Toc24808363)

[Figure 28. Representation of the full path of home directory from root 20](#_Toc24808364)

[Figure 29. Illustrative file system 22](#_Toc24808365)

[Figure 30. Creating a new project from existing directory 27](#_Toc24808366)

[Figure 31. Enter the path of the directory you just created 28](#_Toc24808367)

[Figure 32. Creation of a R Markdown document 30](#_Toc24808368)

[Figure 33. RMarkdown file in RStudio 31](#_Toc24808369)

[Figure 34. HTML Knitr repot 32](#_Toc24808370)

[Figure 35. github\_document production in RStudio 32](#_Toc24808371)

[Figure 36. GitHub upstream repo 34](#_Toc24808372)

[Figure 37. Git process model 34](#_Toc24808373)

[Figure 38. Model of different git command 38](#_Toc24808374)

[Figure 39. GitHub new repo creation 39](#_Toc24808375)

[Figure 40. Git tab of a R project 40](#_Toc24808376)

[Figure 41. Commit and push project files on GitHub 40](#_Toc24808377)

[Figure 42. permissions of files in the working directory 44](#_Toc24808378)

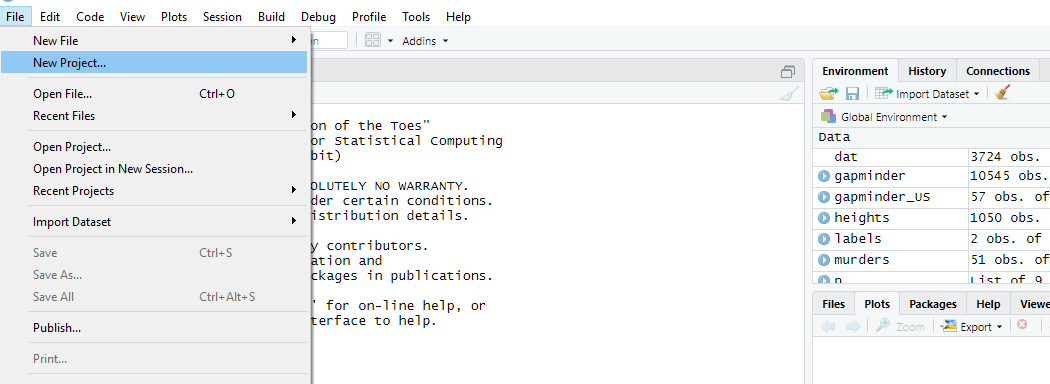
1. Section 1: Installing Software
   1. Keeping Organized with RStudio Projects

A data analysis project is not always a data set and a script. It often involves several scripts. The data may be saved across different files, and it is often convenient to save intermediate files. **RStudio** projects provide a way to keep all this organized in one folder.

To organize yourself on a computer, it will be essential that you understand how your file system is organized. A systematically organized file system can greatly increase your productivity, especially if you work on more than one project at a time. Here we will create a folder in a default location for illustrative purposes. But once you become a regular R user, you will want to think carefully about the best location for the folder in which you will keep a new project.

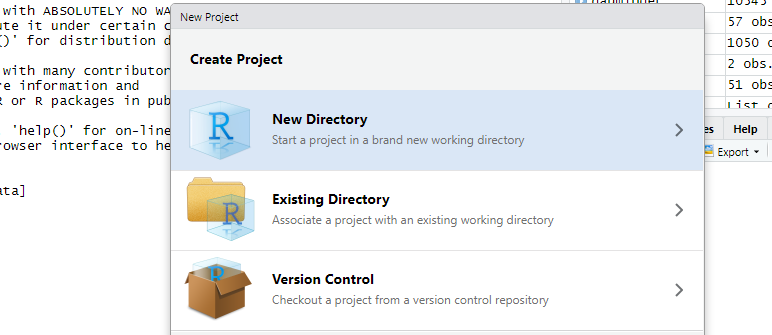
To start a project, you click on File then New project:

1. Select File and New project in RStudio to start a new project

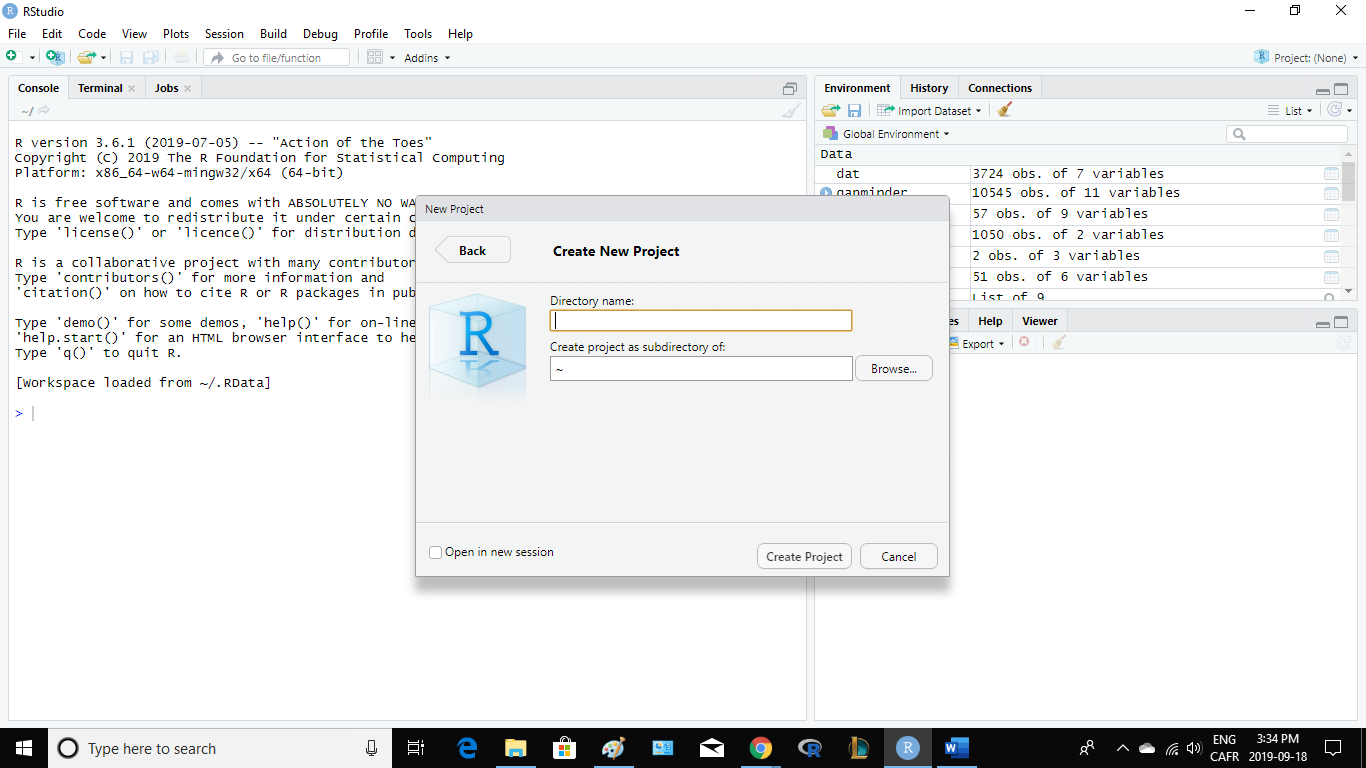


Unless you have a pre-selected folder to save the work, you will select the New Directory option, then for a data analysis project, you usually select the New Project option.

1. Select New Directory

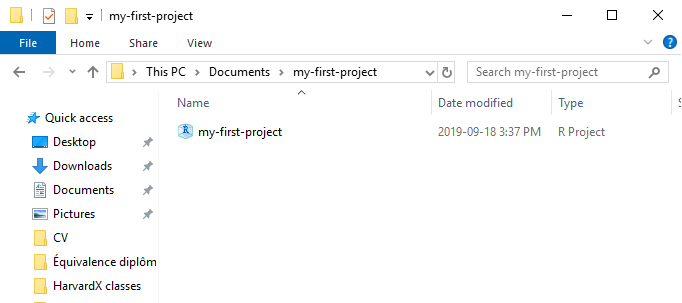


1. Choose a Project Name (that will be the Directory Name) and a location on your computer



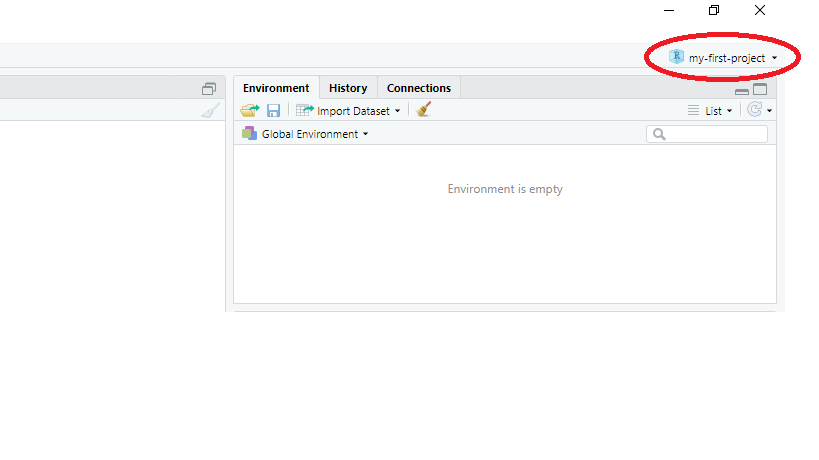
When choosing a folder name, make sure it has a meaningful name that will help you remember what the project is about. As with the files it is recommended to use lower cases letters, no spaces, and use hyphens to separate words. In this example we will call the folder my-first-project. It will create a folder called my-first-project with a file in it called my-first-project.rproj (R project)

1. It creates a folder with a file my-first-project.rproj



Now when you start using RStudio with a project you will see in the upper right corner the project name:

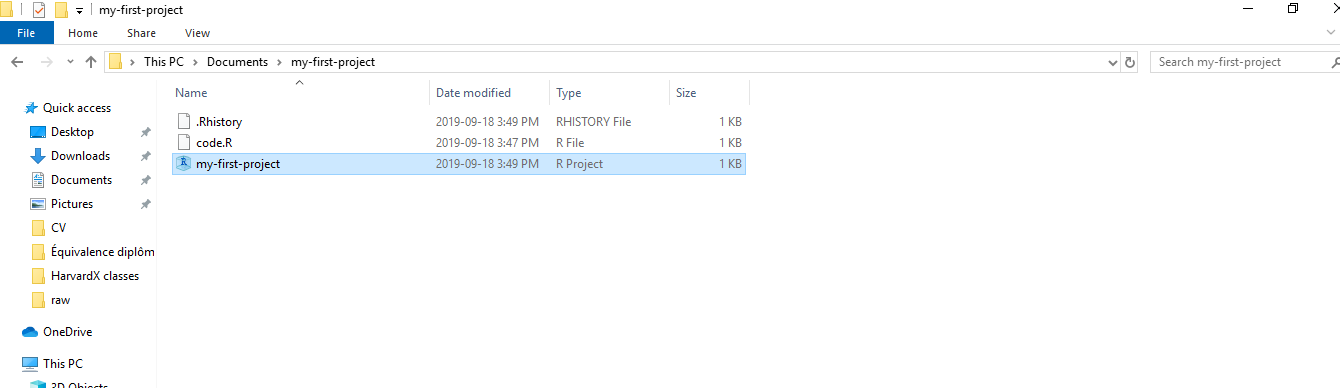
1. The project you working in is displayed at the top in RStudio



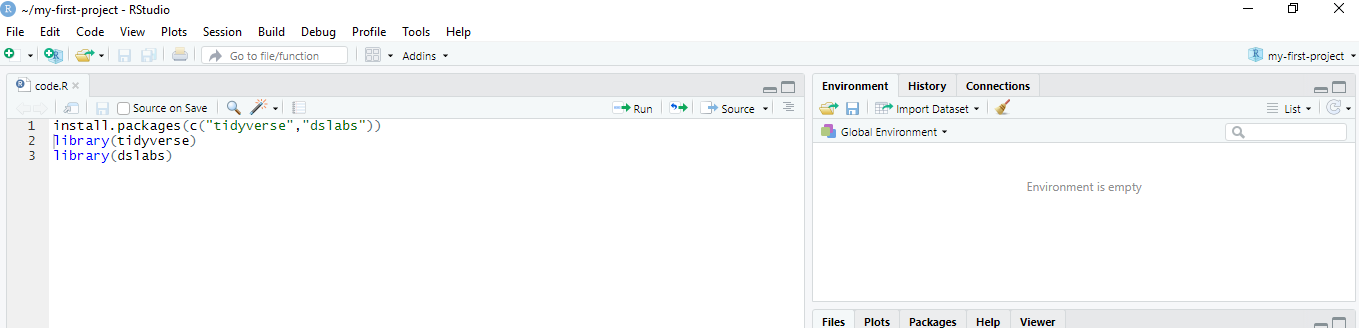
This will remind you what project this particular RStudio belongs to. When you open an RStudio session with no project it will say Project:(None)

When working on a project, all files will be saved and searched for in the folder associated with that project. Here we wrote a script a saved it with the name code.R . Note that if you use meaningful name for the project, we can be a little bit less informative when picking names for the files. Although we did not do it here, note that you can have several scripts open at once. Another advantage of R project is that it saves a file that if you click on it , automatically it will open the project. This file will be in the folder where you started the project. You’ll see file with the suffix Rproj, in this case, the file is called my-first-project.Rproj and you double click on it and it start RStudio, on the project with the file that we were already editing:

1. Folder of our project with the code we saved:



1. When you double click on the rproj file it opens RStudio with the file we were editing



* 1. Introduction to Git and GitHub
     1. Installing Git Introduction

Another great advantage of RStudio projects is that one can share them with collaborators or the public through GitHub. To do this, we will need a piece of software named Git as well as access to a Unix terminal. The installation process for Git is quite different for Mac and Windows. We include both here. Git is what we refer to as a version control system. These are useful for tracking changes to files as well as coordinating the editing of code by multiple collaborators. We will later learn how to use GitHub, which is a hosting system for code. You need Git to interact with GitHub. Having your code, and more generally, data-science projects on GitHub is, among other things, a way to show employers what you can do. Git is most effectively used using Unix, although one can also use it through RStudio.

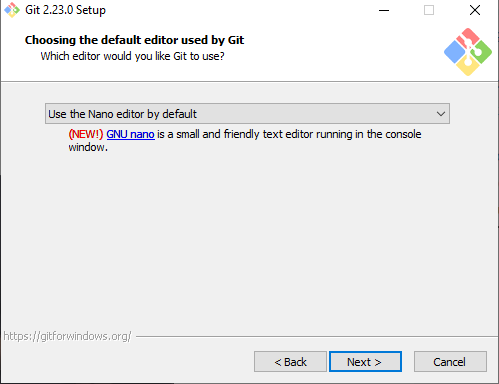
* + 1. Installing Git
       - Installing Git on Windows

We’re going to start by installing Git and Git Bash. There are several pieces of software that permit you to perform Unix commands on Windows. We’ll be using Git Bash as an interface with RStudio, and it is automatically installed when we install Git for Windows. To install Git, start by searching for Git for windows on your browser at this address:

<https://git-scm.com/>

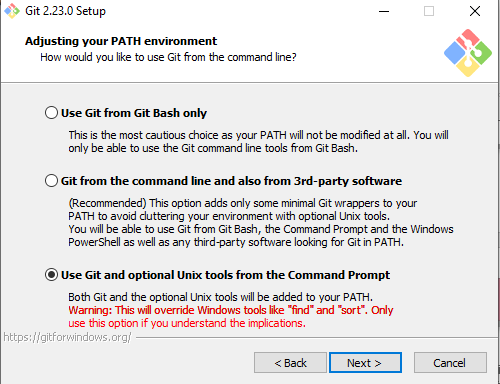
On which you can download the most recent maintained build. Once it finished downloading execute it and select all the default until you get at that page:

1. Choose default editor used by Git



We recommend to use Nano since it’s the easiest to learn. The next step is an important one, it’s the Git-bash installation process, it is recommended to choose “Use Git an optional tools from the Windows command Prompt” option, it will make some windows command line stop working but you will have a Unix equivalent for those.

1. Installing git Bash



You can choose the default for all the other options. Once it finishes installing you will have Git and Git Bash. To check that you are using Git Bash in RStudio you can open a new terminal in RStudio, and it should look something like this:

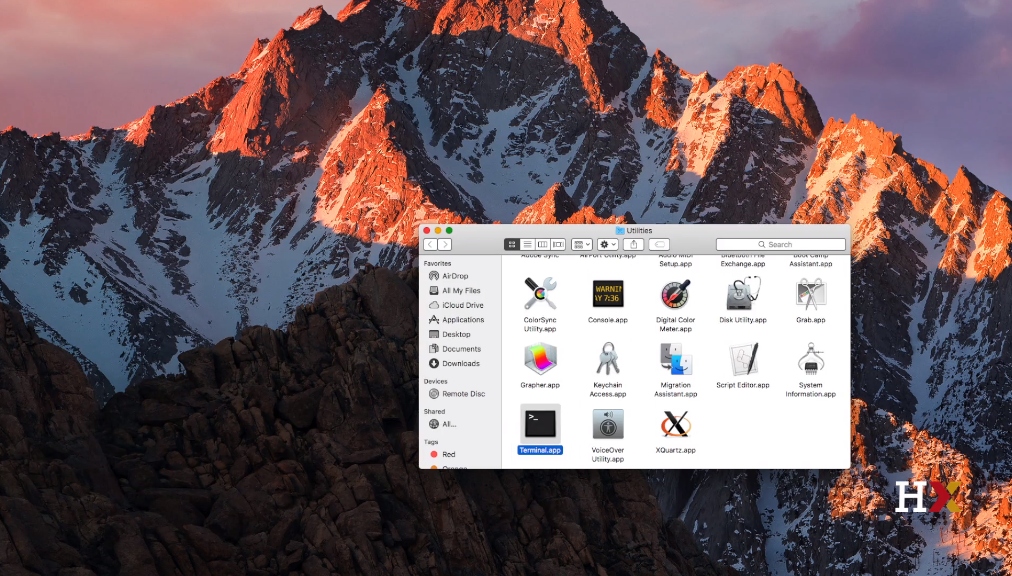
1. RStudio using Git Bash



* + - * Installing Git on Mac

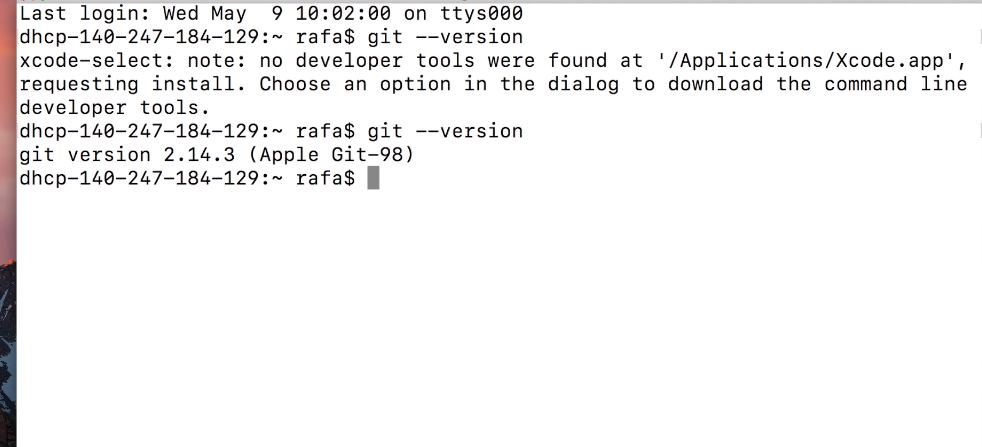
Macs already come with a terminal and it can be used to learn Unix. We can also use it to check if Git is already installed, start the installation process. To open a terminal, you can use Command-Spacebar and then type terminal, or you can fin it in the Utilities folder like this:

1. Terminal App in Mac OS



Once you open it you will have a window like this, if you type

*git –version*



You should have a version number, if not you will be prompted to install git on the terminal. Accept and you have installed Git on your Mac.

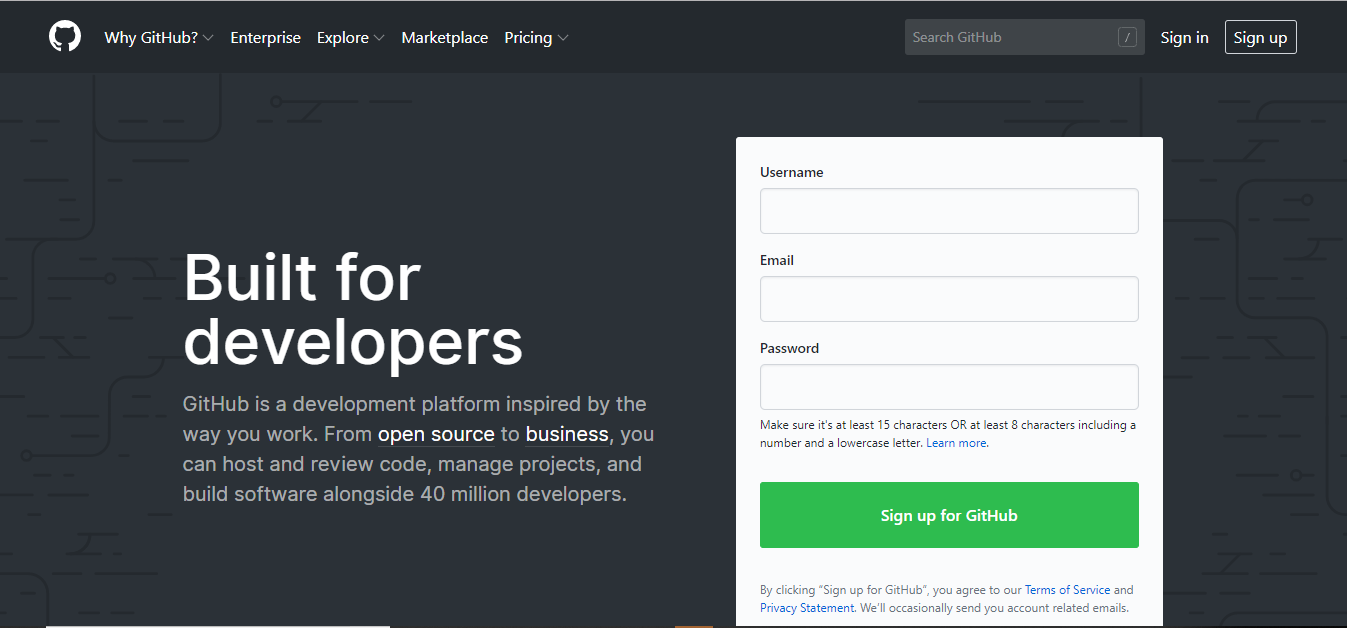
* + 1. GitHub

Basic GitHub accounts are free. The first step is to go to:

<https://github.com/>

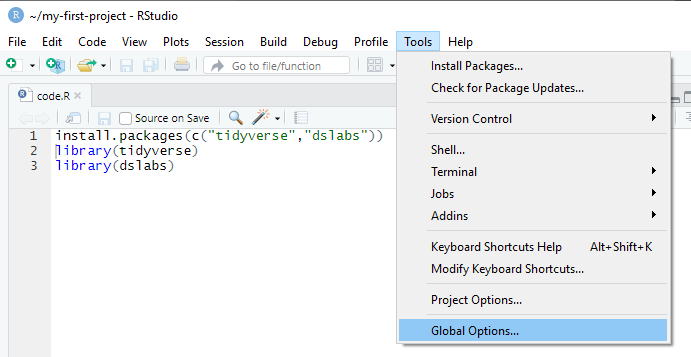
And create an account, you want a name for the account that is short, easy to remember and spell and professional. Since you might be sending potential employers to you GitHub account:

1. GitHub account creation page



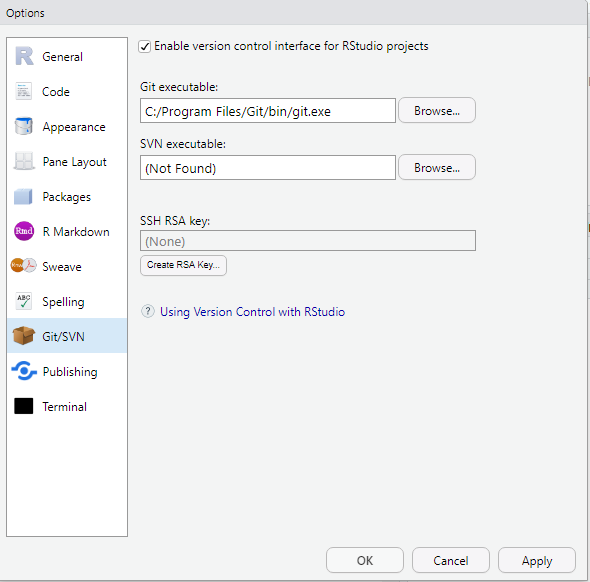
Once you have an account, you’re ready to connect it to RStudio. You start by going to Global option and Gitsvn in RStudio:

1. Global option in Rstudios



You need know to select the path to git executable that we just installed:

1. Git/SVN global options



To avoid to enter the password everytime you connect to RStudio click on create an RSA key.

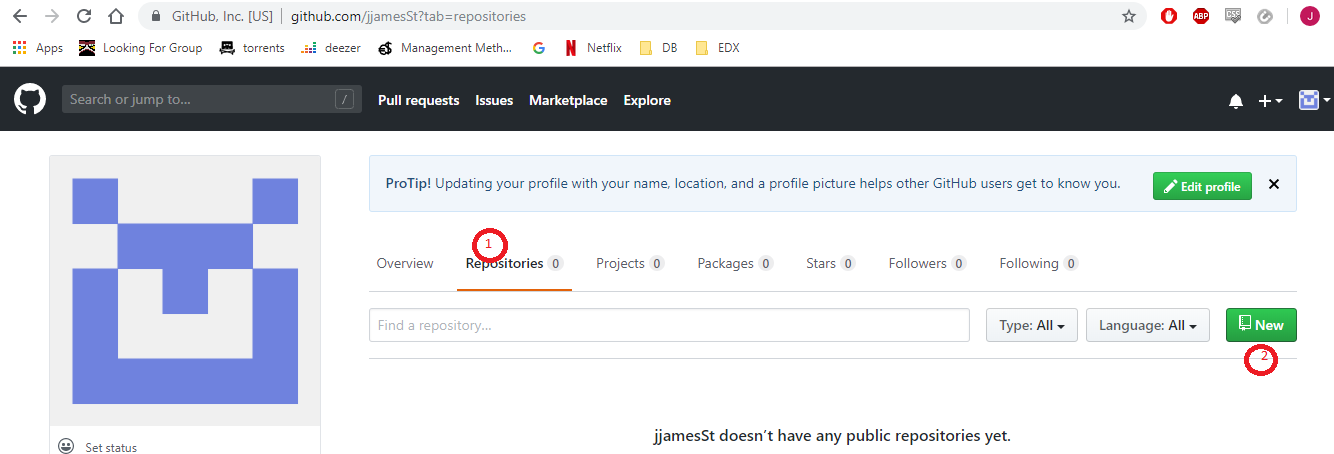
* + 1. GitHub Repositories

The general idea behind GitHub repositories, is to have constantly at least 2 copies of your code on GitHub and on your computer. If you add collaborators to this project, then each will have a copy on their own computer. The GitHub copy is usually considered the master copy that each collaborator syncs to. The first step in creating a repo for your code is to initialize one on GitHub. Go to:

[GitHub.com/yourusername](https://github.com/jjamesSt)

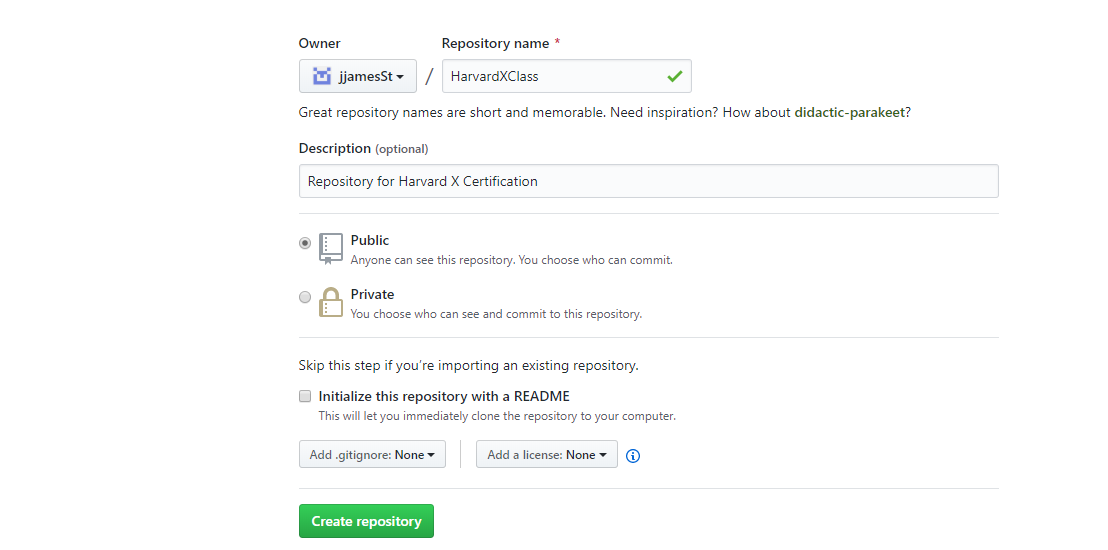
You will get to this page, select Repositories and then new

1. Selecting new GitHub repositories



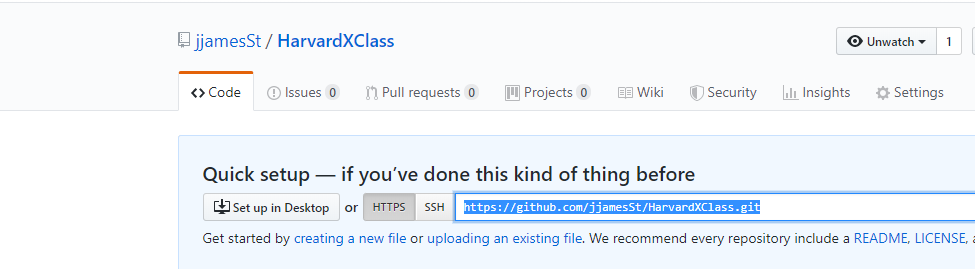
Then you will choose a name for that repo, remember that you will have dozen of repos for different projects, and so choose a name that represent well your project, here we use HarvardXClass for the example:

1. Creating new GitHub repo



If you want to create a private repo it will cost monthly charge. Once the repo is created copy the HTTPS link given to connect to it through RStudio

1. Address to connect to the repo



* + 1. RStudio, Git, and GitHub

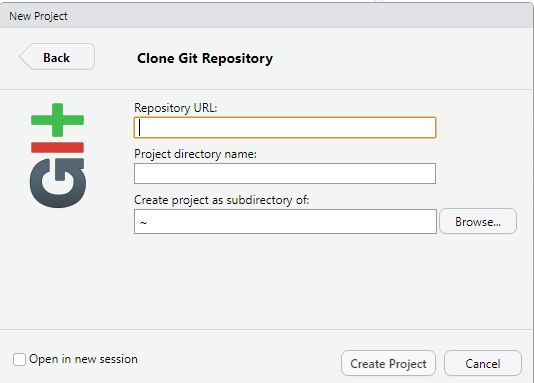
We are now ready to clone our repo, start editing file on your computer, and syncing with GitHub. We will use RStudio to facilitate this, and we’ll use Unix for the first time. Our first step is to let Git know who we are. This will make it easier to connect to GitHub. We start by opening a terminal RStudio and use the git config command to identify ourselves. You type:

*git config --global user.name “Your Name”*

*git config –global user.mail “Email use to create Git Hub account”*

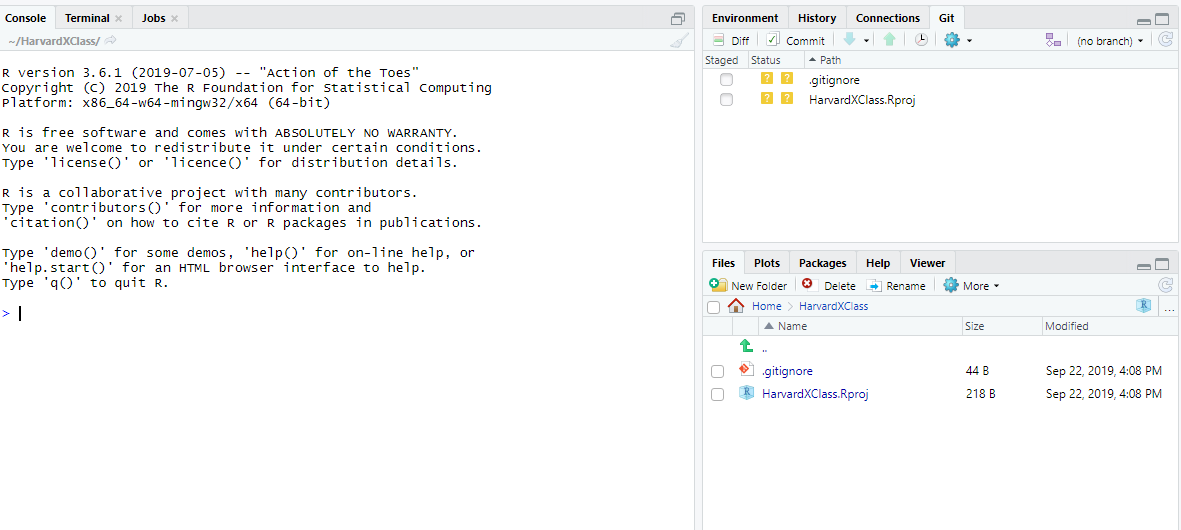
Now we can create a new project with RStudio that use version control and stores the code on GitHub. To do this we start a project, but instead of New directory, we choose Version control, then Git like this:

1. New project in RStudio with Git



In the repository URL you want to copy the link we copied earlier after the creation of the repo. Once you click Create project you will see a new tab in the upper right named Git:

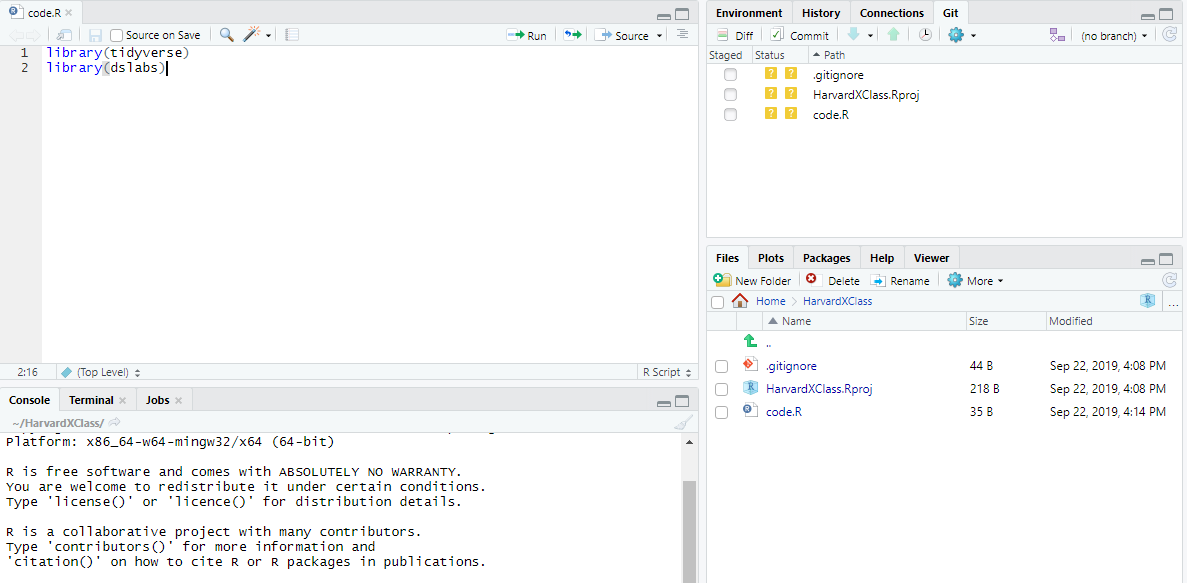
1. Git project in RStudio



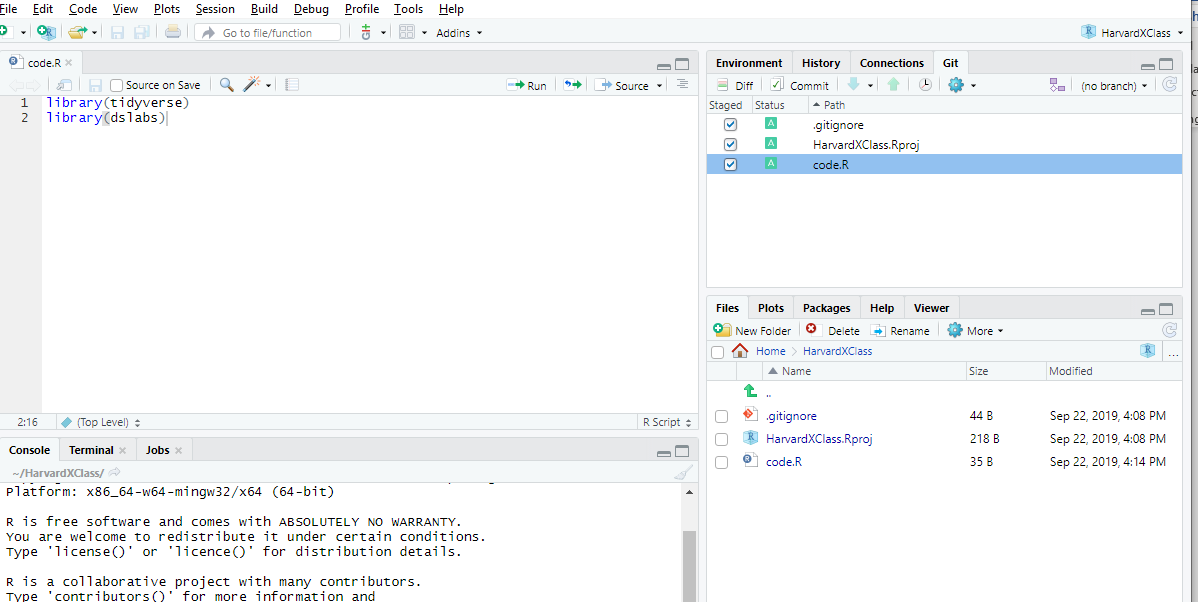
You can see on this tab a number of files with status icon that give you information about their relationship with GitHub. Your local file and GitHub repo will not be synced automatically. The main action of Git is to pull your code from the GitHub repo, then **add** or **modify** a file (stage files), then **commit** your changes on your local folder and finally **push** your modification to the GitHub repo.

As an example let’s add a simple R code to our repo. First, we create a new file with some code, once we save the file it will automatically add it to the Git tab:

1. Adding a file to the Git tab:

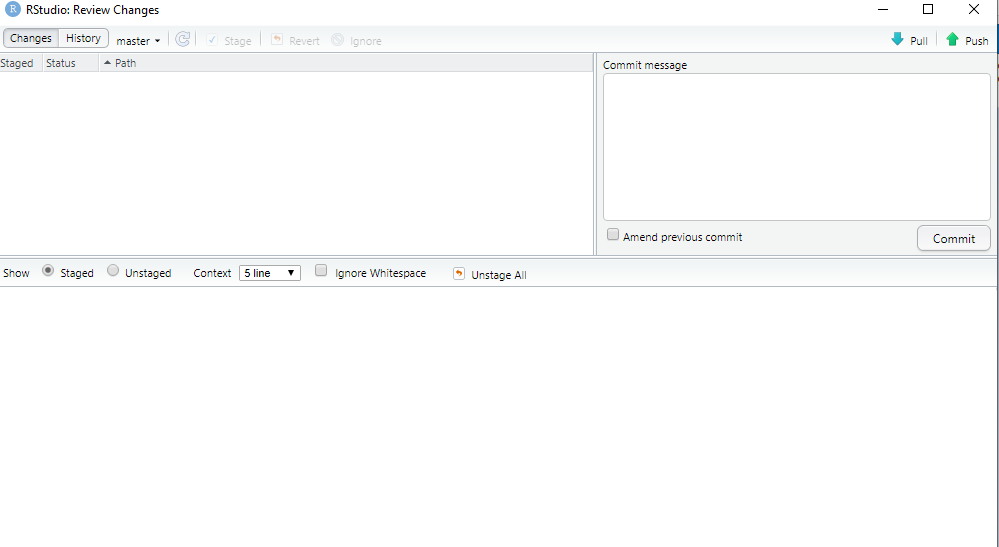


Because this is an empty repo, we don’t need to pull data first. To add the file we need to check the staged checkbox for every file we want to add to the repo:



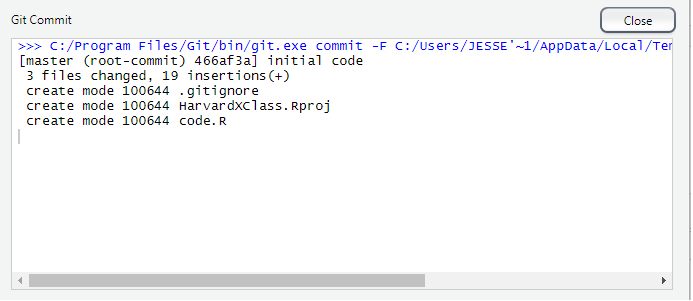
We can now commit our changes, by clicking on the commit button. This will open a new dialog window, with Git you need to add a comment describing the changes you made in the code before committing it, furthermore this window give you a summary of all the changes you are committing.

1. Committing window



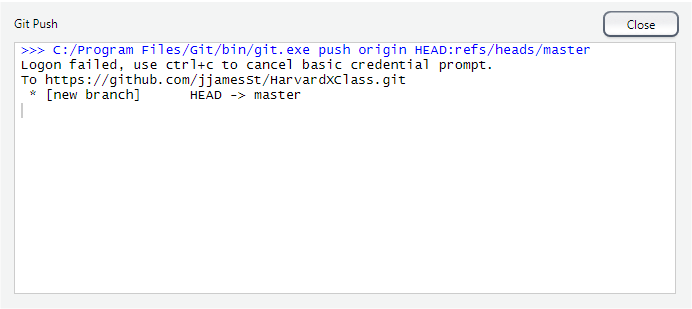
Once a comment is filled and you click the commit button, you’ll have a summary of the changes committed:

1. Summary of submitted commit



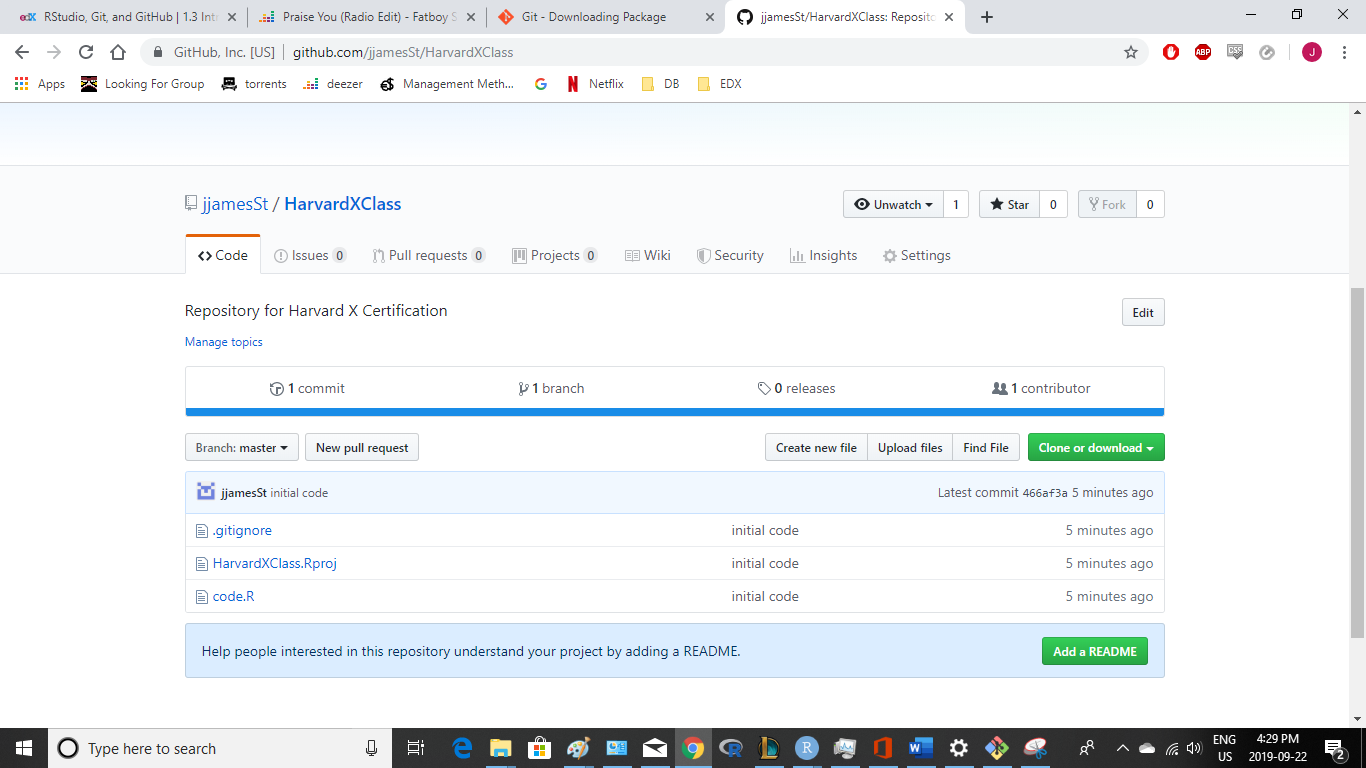
Now we can push those changes in the GitHub repo by pressing the Push button:

1. Pushing the changes to the GitHub repo



If you go back to the URL of your repo you should see the new file:

1. GitHub repo after push



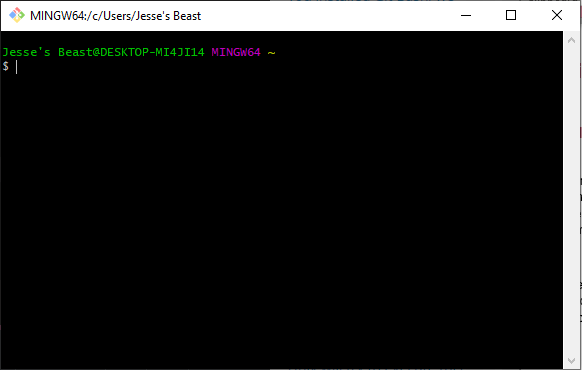
1. Basic Unix
   1. Introduction to Unix
      1. The Terminal

The terminal is our window into the Unix world. Instead of clicking, dragging and dropping to organize our files and folders, we’ll be typing commands into the terminal. We’ll do this in the terminal like how we type commands into the R Console. But now instead of generating plots and statistical summaries, we’ll be organizing files on your system. We have already described how to access the terminal using RStudio namely by going:

*Tools->Terminal->New Terminal*

But often, we want to access the terminal and have no need for RStudio. On windows, assuming you installed Git Bash, we can also access a terminal without RStudio by running the Git Bash program like this. Once you have a terminal open, you can start typing commands. You should see a blinking cursor at the spot where what you type will show up. This is called **the command line**:

1. Git Bash terminal the “|” represent the command line



If you want to try out an example, type into the command line:

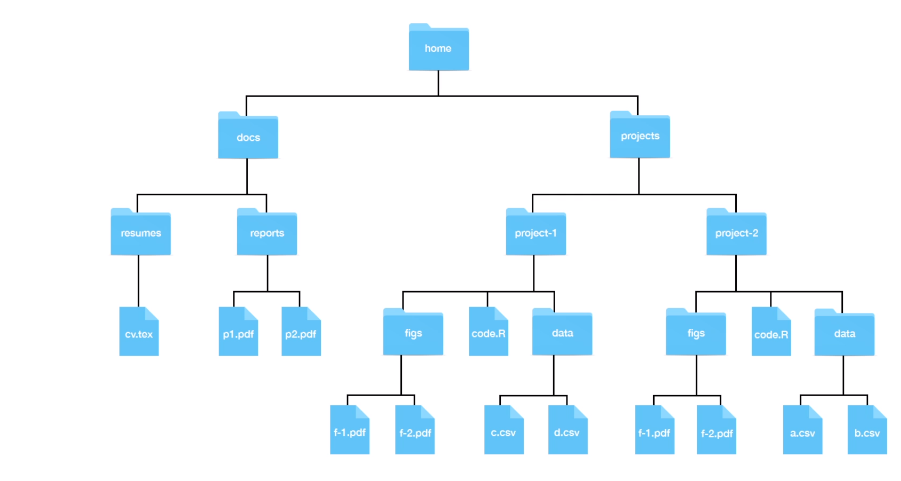
*echo “Hello world”*

The **echo** function is similar to cat in R and executing this line should print out “Hello world”. To go back to a command you previously typed, you can use the up arrow

* + 1. The Filesystem

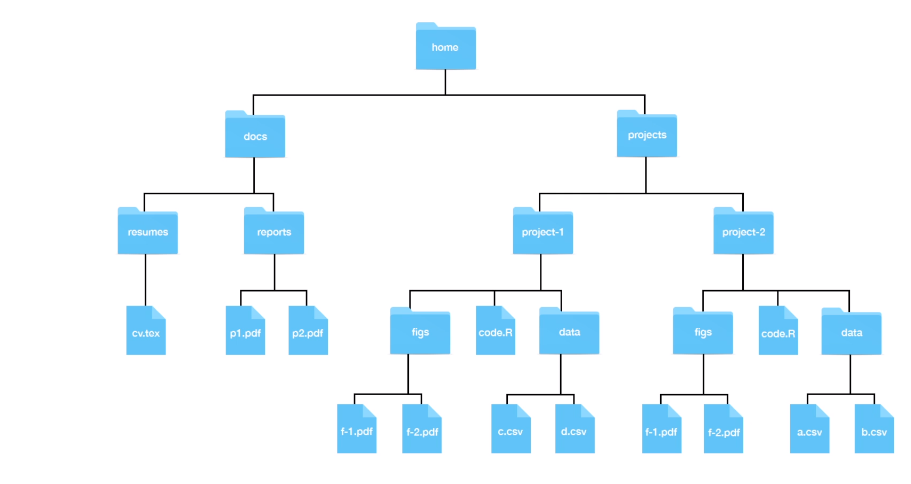
All the files, folders, and programs on your computer are referred as **filesystem**. We’ll focus on files and folders for now and discuss programs, or executables as we call them in Unix, later. The first concept you need to understand, is how your file system is organized. You should think of it as a series of nested folders each containing files, folders, and executables. Here’s a visual:

1. Example of file system



In Unix, we refer to folders as directories. Directories that are inside other directories are often referred to as subdirectories. So for example here :

1. Example of a directory “Docs” with 2 sub directories “reports” and “resume”



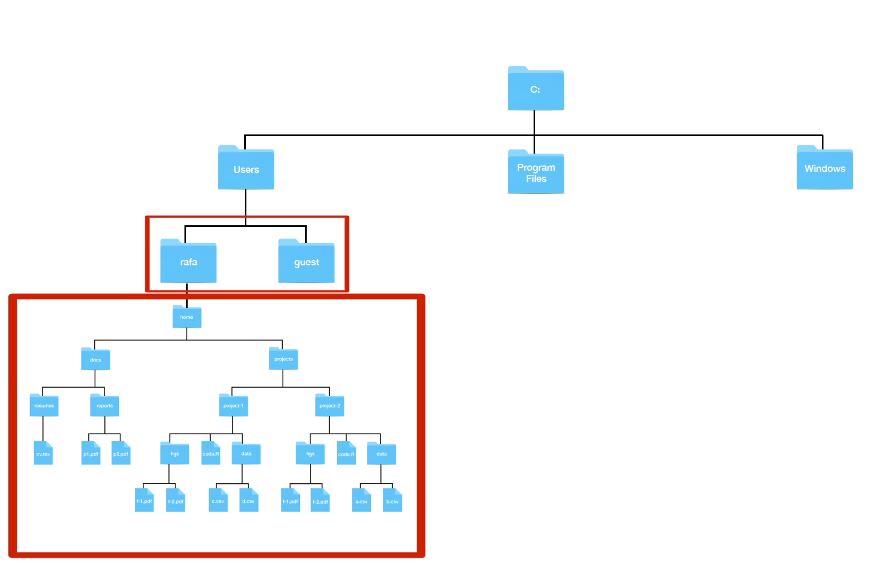
The “Home” directory is where all your stuff is kept. In the figure above, the directory called “Home” represents your home directory, but that is rarely a name that is used. On your system, the name of your home directory is likely the same as your username on that system.

Now if you look at the figure above and you want to remove the file cv.tex . Suppose that on your screen, you can see the home directory. To erase this file , you would double click on the Home, double click on Docs, then double clicks on resumes and then drag cv.tex to the trash. Here you are experiencing the hierarchical nature of the system. Cv.tex is a file inside resumes directory, which is a subdirectory inside the Docs directory, which is a subdirectory of the Home directory

Now suppose you can’t see the Home directory on your screen. You would somehow need to make it appear on your screen. One way to do it is to navigate from what is called the root directory, all the way to your home directory. Any file system will have what is called the **root directory**, which is the directory that contains all directories on your system.

The home directory shown in the figure above will usually be two or more levels from the root:

1. Representation of the full path of home directory from root



* + 1. Working Directory

The concept of a current location is part of the point and click experience. At any given moment, we’re in a folder and see the content of that folder. As you search for a file, you experience the concept of a current location, Once you double click on a directory, you change locations an now are in that folder, as opposed to the folder you were before. In Unix, we don’t have the same visual cues, but the concept of a current location is indispensable. We refer to this as the **working directory**. Each terminal window you have open has a working directory associated with it.

The first command we learn in Unix is **pwd**, which stands for Print Working Directory. You can open a terminal and type *pwd* and you will see your home directory:

*pwd*

*/c/Users/Jesse's Beast*

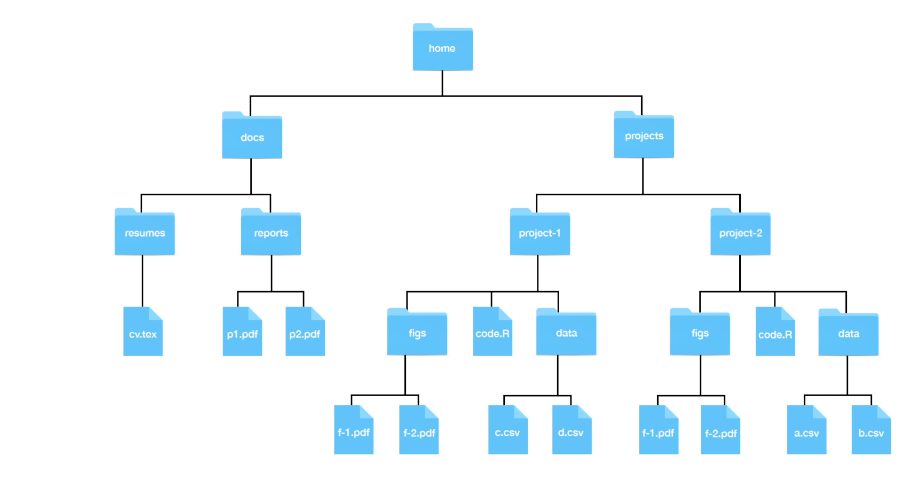
When we first open a terminal, it will start in our home directory. So, in this case, the working directory is the same as a home directory. Note the 4 slashes in the string separate directories, which imply that the root directory is represented here by just a */* .

We refer to string returned by pwd as the full path of the working directory. The name comes from the fact this string spells out the path you need to follow to get to the directory in question from the root directory. Every directory has a full path. Note that in Unix we use the shorthand ~ as a nickname for you home directory. So, for example, if docs is in the path to a directory in your home directory, docs can be written as ~/docs.

* 1. Working with Unix
     1. Unix Commands

We will learn a series of Unix commands that will permit s to prepare a directory for a data science project. However, we’re going to be typing commands that may not work on your system. This is because every system is different, for the code we’re going to be typing we’re going to assume the file system in the following figure:

1. Illustrative file system



The first command we are going to learn is **ls**. In a point and click system, we know what is in a directory because we see it. In the terminal, we don’t see the icons, instead, we use the command ls to list the directory content. To see the content of your home directory, open a terminal an type ls here is an example:

*ls*

*'~$ference and Modeling.docx'*

*'~$obability.docx'*

*'~$oductivity Tools.docx'*

*'~$S Course\_Notes jfs TEMPLATE v1.dotx'*

*'Data Science R Basics.docx'*

*docs/*

*'General Linear Models second edition.pdf'*

*'HarvardX classes.Rproj'*

*'HBS Course\_Notes jfs TEMPLATE v1.dotx'*

*'Inference and Modeling.docx'*

*Probability.docx*

*'Productivity Tools.docx'*

*projects/*

*teachings/*

*Visualization.docx*

Next commands are **mkdir** and **rmdir**, mkdir is used to create directories and rmdir to delete them. Because you will soon be working on several projects, we highly recommend creating a directory called projects in your home directory, where you store all your project related directories. For example:

*mkdir projects docs teachings*

*Jesse's Beast@DESKTOP-MI4JI14 MINGW64 ~/Documents/HarvardX classes*

*$ ls*

*'~$ference and Modeling.docx'*

*'~$obability.docx'*

*'~$oductivity Tools.docx'*

*'~$S Course\_Notes jfs TEMPLATE v1.dotx'*

*'Data Science R Basics.docx'*

*docs/*

*'General Linear Models second edition.pdf'*

*'HarvardX classes.Rproj'*

*'HBS Course\_Notes jfs TEMPLATE v1.dotx'*

*'Inference and Modeling.docx'*

*Probability.docx*

*'Productivity Tools.docx'*

*projects/*

*teachings/*

*Visualization.docx*

The rmdir will remove a directory as long as it is empty, if it’s not, you will get an error message, and the directory will remain untouched.

Next, we want to create directories inside directories we have created, but we want to avoid pointing and clicking our way through the file system. Instead, we’re going to learn how to this in Unix using the command **cd**. The command cd stands for change directory, so we simply type:

cd projects

pwd

/c/Users/Jesse's Beast/Documents/HarvardX classes/projects

And we can see that the working directory is now projects. Before we continue, an important tip, note that in Unix, you can auto complete by hitting Tab. This means that we can type cd, the letter d, and then hit Tab and on Unix, it’ll either autocomplete to docs if it’s the only directory starting with “D” or show you the options.

When using cd you can either use the full path or the relative path of a directory in the example above we used the relative path to move from the working directory to projects and the pwd show us the full path of the projects directory. Unix will assume a relative path if there is no forward slash or a tilde Unix will assume you are using relative path.

Now let’s suppose we want to move back to the directory in which projects is a subdirectory. This is referred to as a parent directory. We could use the full path of the parent directory, but Unix provides a way of doing it more quickly the **dot dot(..)** command so to move back we simply type :

*cd ..*

*pwd*

*/c/Users/Jesse's Beast/Documents/HarvardX classes*

Note that the working directory also has a nickname, and it is a single dot. You can go back to whatever directory you just left by typing:

*cd –*

*pwd*

*/c/Users/Jesse's Beast/Documents/HarvardX classes/projects*

* + 1. mv and rm: Moving and Removing Files

On a point and click system, we move files from one directory to another by dragging and dropping. On Unix, we use the move command, the **mv** command.

**The mv command will not ask you if you are sure. So if your move results in overwriting a file, you lose the file, BE CAREFUL!**

Now that how to use full and relative of paths, using move is relatively straightforward. The general form is :

mv path\_file\_you\_want\_to\_move path\_destination\_directory.

So ,for example , if we want to move the file cv.tex from Resumes to Reports, you could use the full paths like this:

*mv ~/docs/resumes/cv.tex ~/docs/reports/*

You can also use relative paths, so you could do this:

*cd ~/docs/resumes*

*mv cv.tex ../report/*

or do this:

*cd ~/docs/reports*

*mv ../cv.tex/*

Note that in this last one we use the working directory shortcut, dot, to five a relative path as the destination directory. We can also use move to change the name of a file. To do this instead of the second argument being the destination directory, it is also a filename. So for example, to change the name from cv.tex to resume.tex, we simply type:

*cd ~/docs/resumes*

*mv cx.tex resume.tex*

We can combine move and rename, for example :

*cd ~/docs/resumes*

*mv cv.tex ../reports/resume.tex*

We can also move entire directories. So to move resume directory into the reports directory we do this:

*mv ~/docs/resumes ~/docs/reports/*

it is important to add that last “/” to make it clear that you do not want to rename the resumes directory to reports but rather move it into the reports directory. The copy command **cp** behaves similar to the move command, except that instead of moving we copy the file, meaning that the original file system remain untouched. So in all the mv example, we can switch mv to cp, and they will copy instead of move with one exception. We can’t copy entire directories without learning about arguments.

In a point and click systems, we remove files by dragging and dropping them into the trash or using some special click on the mouse. In Unix, we use a command **rm**.

**Unlink throwing files into the trash rm is permanent so be careful**.

The general way it works is like this:

rm filename

We can list files that we want to remove like this:

rm fileone filetwo filethree

You can use full or relative paths. To remove directories, we need to learn about arguments.

* + 1. less: Looking at a File

Often you want to quickly look at the content of a file. If this file is a text file, the quickest way to do this is by using the command less. So for example, to look at the file cv.tex, you should do something like this:

*cd ~/docs/resumes*

*less cv.tex*

This will open up the less viewer. To exit the viewer, you type q. To move up an down the file, you use the arrows. Now there are many other keyboard command you can use with less for example to search or to jump pages. We will learn more about this in a later section when we learn how to learn about commands. Now if you are wondering why the command is called “less”, it is because the original was called “more” as in “show me more of this file”. The second version was called “less” because of the saying “less is more”.

* + 1. Preparing for a Data Science Project

We are now ready to prepare a directory for a project. You should start by creating a directory where you’ll keep all your projects. We recommend a directory called “projects” in your home directory. To do this, you would type:

*cd ~*

*mkdir projects*

The project we are going to work on here as an example, relates to gun violence murders. So we will call the directory “murders\_analysis”, it will be a subdirectory in our projects directories. In the muders\_analysis we’ll create 2 subdirectories to hold the raw data and intermediate data . We’ll call these “data” and “rda”, respectively. RDA stands for R Data. So open a terminal and type:

*cd projects*

*mkdir murders\_analysis*

*$ cd murders\_analysis*

*$ mkdir data rda*

*$ ls*

*data/ rda/*

*pwd*

*/c/Users/Jesse's Beast/Documents/HarvardX classes/projects/murders\_analysis*

Note that the full path of our murders data set is :

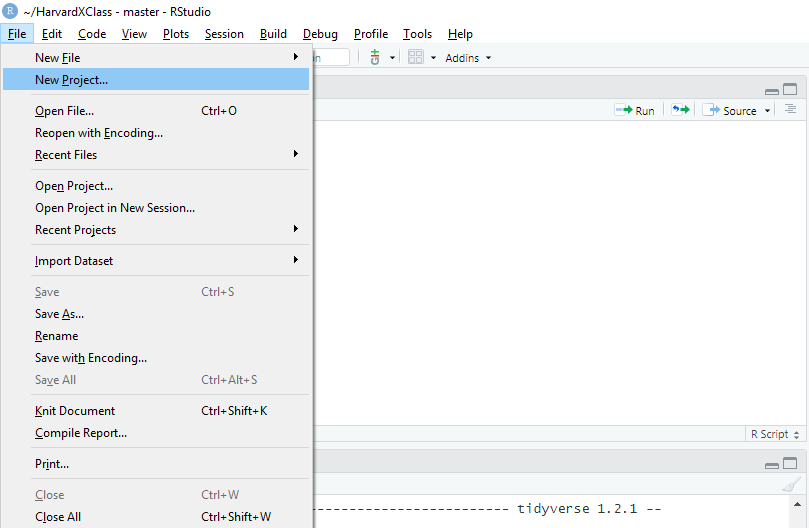
*~/Documents/HarvardX classes/projects/murders\_analysis*

So if we open a new terminal and want to navigate into this directory, we type:

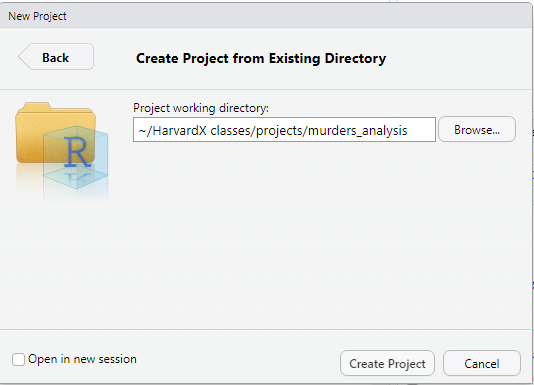
*cd ~/Documents/HarvardX classes/projects/murders\_analysis*

Now in RStudio when you start a new project you can pick existing directory of new directory:

1. Creating a new project from existing directory



1. Enter the path of the directory you just created



Once you do this, you will see that rda and data directories you created are in the files tab.

An important note, when you are in this project, our default woking directory will be Documents/HarvardX classes/projects/murders\_analysis. You can confirm this by typing in your R Console:

*getwd()*

In your R code, always use relative paths when you’re working on a data science project and relative path to a default working directory. The problem with using full paths, is that your code is unlikely to work on file systems other than your own, as other directory structures change. So this includes using the home directory. You do not want to use ~ in your code.

Let’s create a script to download a file into the data directory. Let’s call this file download-data.R:

*url<-"https://raw.githubusercontent.com/rafalab/dslabs/master/inst/extdata/murders.csv"*

*dest\_file<-"data/murders.csv"*

*download.file(url,destfile = dest\_file)*

running this script will add a file in the data directory. Now we need a script to read that file and prepare a table that we can use for further analysis, we going to call this new script wrangle-data.R:

*library(tidyverse)*

*murders<-read.csv("data/muders.csv")*

*murders<-murders%>%mutate(region=factor(region),rate=total/population\*10^5)*

*save(murders,file="rda/murders.rda")*

In this file we introduce a command we have not yet seen. It’s the command **save()** the save command in R saves objects into what is called an “rda file” RDA stands for R Data.

Although not the case here, this approach is often practical because generating the data object we use for final analysis and plots can be complex and time consuming. So we don’t have to run it every time we start the project. So we run this process one time and save the file. But we still want to be able to generate the entire analysis from the raw data. So we include that script as well.

Now we are ready to create the analysis file We’re going to call this one analysis.R :

*library(tidyverse)*

*load("rda/murders.rda")*

*murders%>%mutate(abb=reorder(abb,rate))%>%*

*ggplot(aes(abb,rate))+*

*geom\_bar(width=0.5,stat="identity",color="black")+*

*coord\_flip()*

If you run this script you’ll see a that it generates a file. Now suppose we want to save the generated file for use in a report or a presentation. We can do this using the command **ggsave**(), this is part of the ggplot package. We should systematically organized so we will save plots to a directory called, “figs”. So we have to create a directory with mkdir in the project and then add the command ggsave in our script :

g*gsave(“figs/barplot.png”)*

We know have a self contained analysis in one directory. One final recommendation is to create a README.txt file describing what each of these files does for the benefit of others reading your code, including your future self. This would not be a script, but just some notes. Note that one of the options for opening a new file in RStudio, is a text file. You can save something like this:

1. Reproducible Reports
   1. Reproducible Reports
      1. Reproducible Reports with R Markdown

The final product of a data analysis project is often a report. Many scientific publications can be thought of as a final report of a data analysis. The same is true for a news article base on data, an analysis report for your company, or lecture notes for a class on how to analyze data. The reports are often on paper or they’re a PDF that include textual descriptions of the findings along with some figures and tables resulting from the analysis.

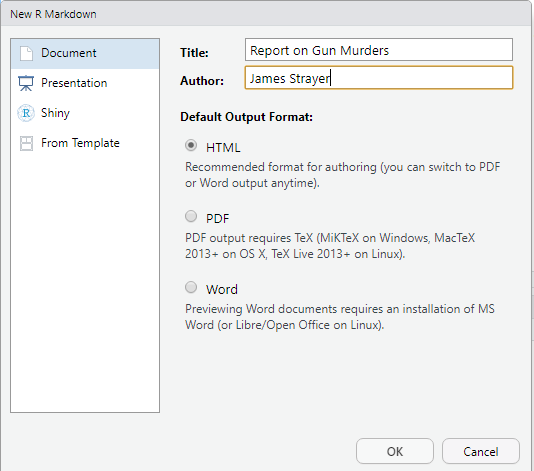
Now imagine ,that you realize that you were given the wrong dataset, and they send you a new one you’re asked to run the same analysis but with this new dataset. Or, what if you realized that a mistake was made an you need to re-examine the code, fix the error and rerun the analysis? Or imagine that someone you’re training wants to see the code you used for report and be able to reproduce the results to learn about the approach. Situations like the ones just described are actually quite common for a data scientist. Here, we describe how to generate reproducible reports with **RMarkdown** and **knitr** in a way that will greatly help with situations such as the ones we described. The main feature is that code and textual descriptions can be combined into the same document. And the figures, and tables produced by the code are automatically added to the document

* + 1. R Markdown

**RMarkdown** is a format for what is called literate programming documents. It is based on Markdown, a markup language that is widely used to generate HTML pages. You can learn more about Markdown at the website provided here. Literate programming weaves instructions, documentation, and detailed comments in between machine-executable code, producing a document that describes the program that is best for human understanding. Unlike a word processor, such as Microsoft Word, where what you see is what you get, with R Markdown, you need to compile the document into the final report. **KnitR** is what compile R Markdown documents. The R Markdown document looks different than the final product. This seems like a disadvantage at first, but it is not at all. Because for example, instead of producing plots and inserting them one by one into the word processing document, the plots are automatically added.

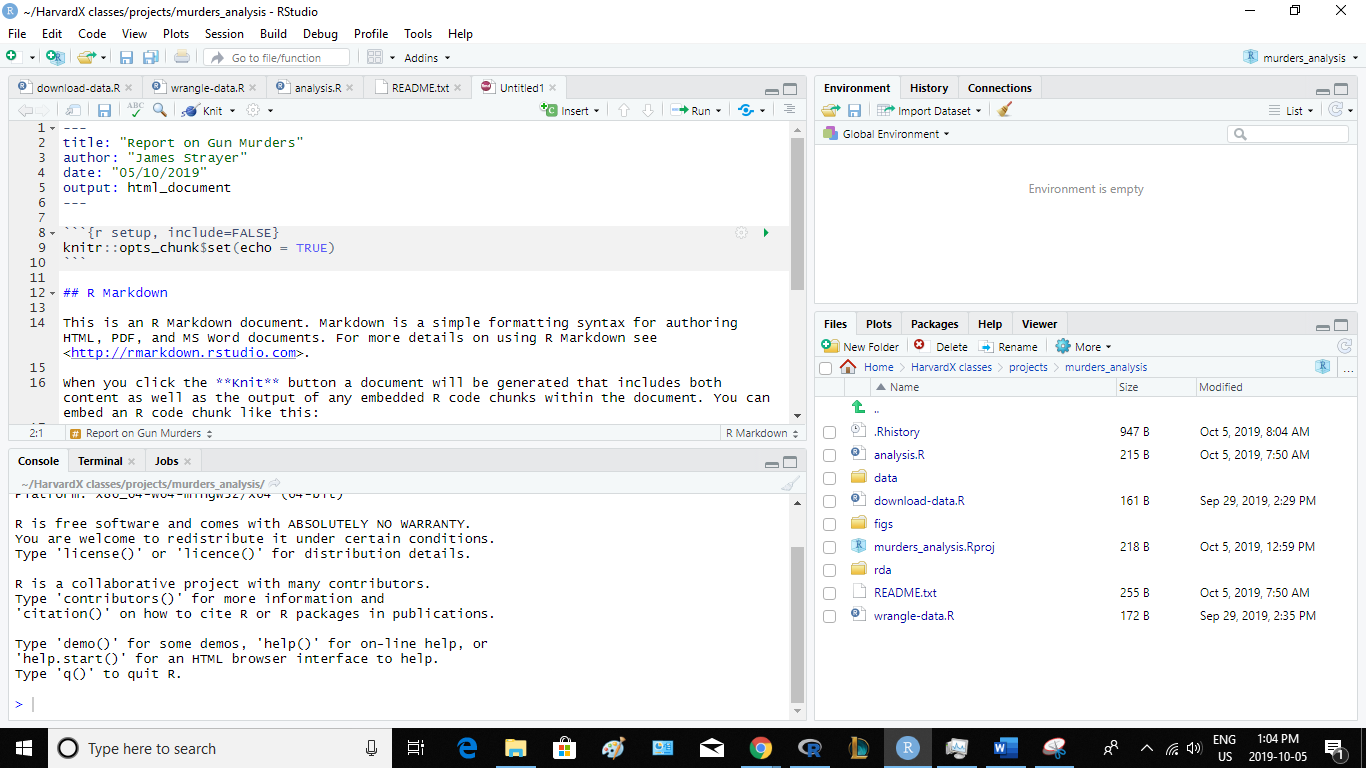
In RStudio, you can start an R Markdown document by clicking on File, then New File, then R Markdown, like this. You will then be asked to enter a title and an author for you document. We’re going to prepare a report on gun murders:

1. Creation of a R Markdown document



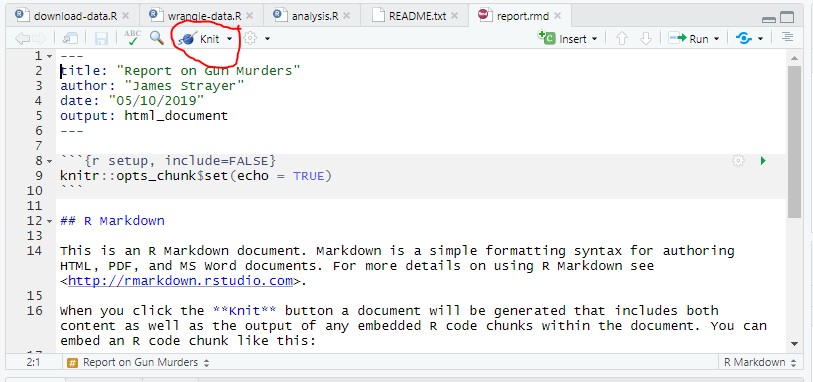
You can also decide what format you would like to file a report to be HTML, PDF or Microsoft Word. We can actually easily change this later, but in this example, we choose HTML as it is the preferred format for debugging purposes.

1. RMarkdown file in RStudio



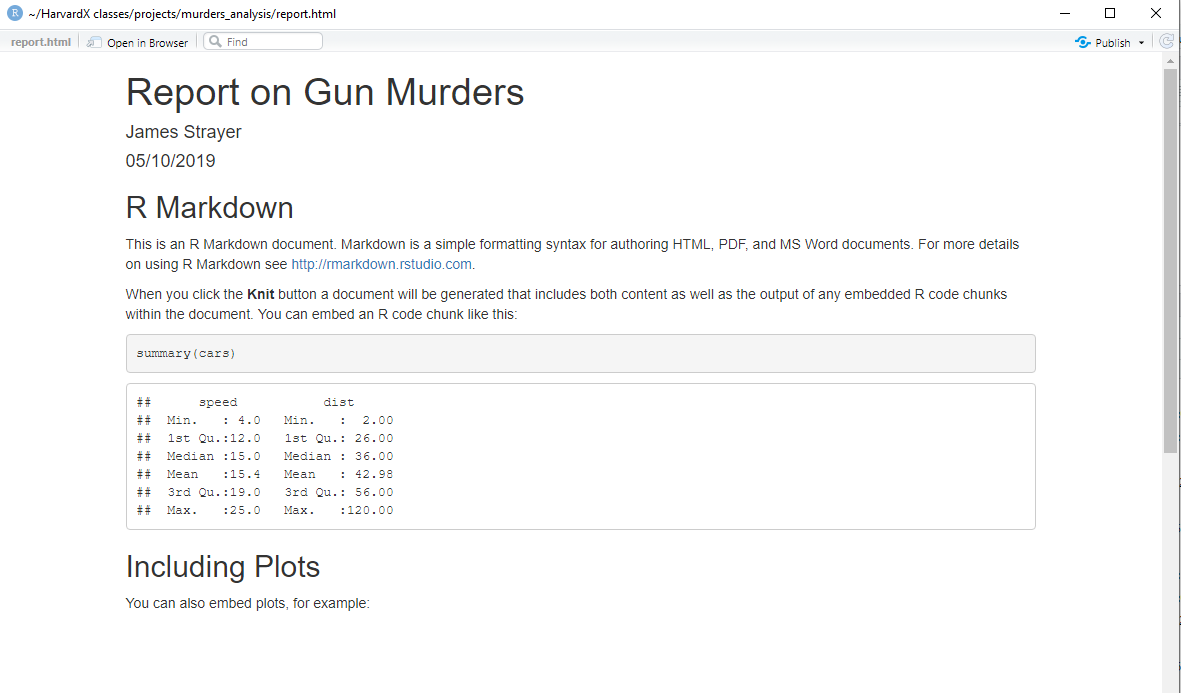
* + 1. Knitr

To convert an R markdown file into a document, we need to use the **knitr** package. The specific function used to compile is the knit function. Now, RStudio provides a button that makes it easier to compile the document.



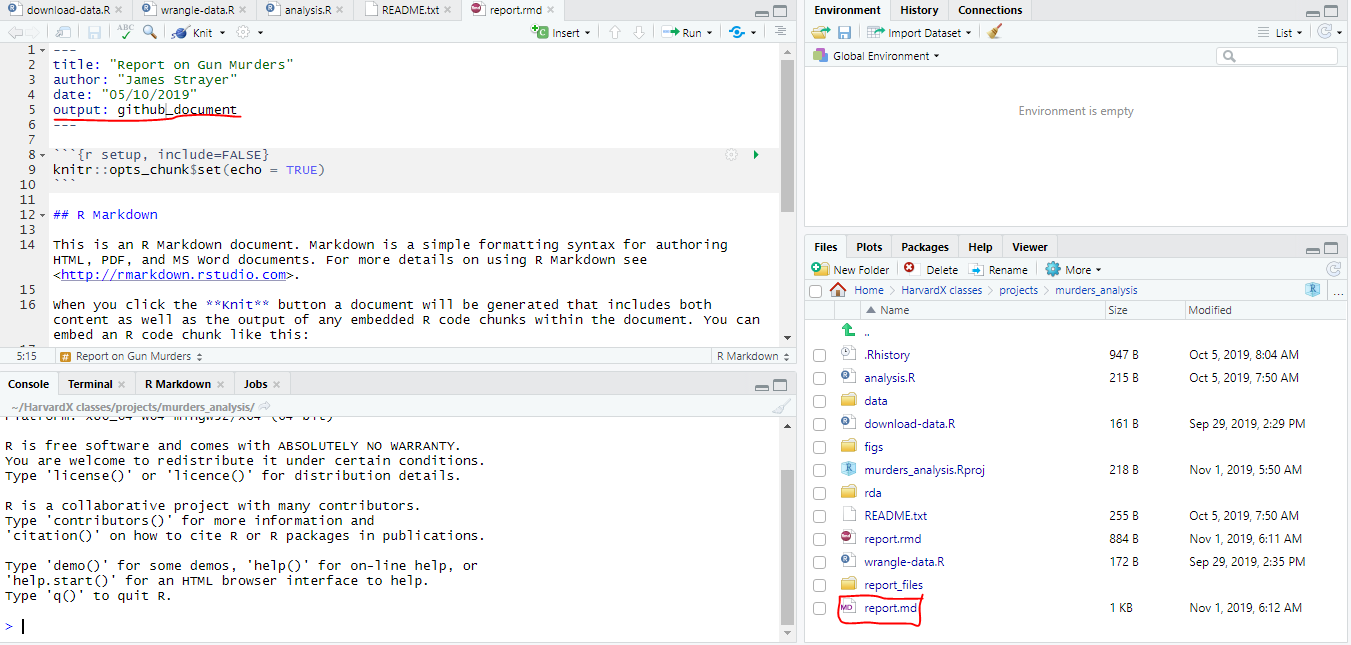
If it’s the first time you use it, you might get a pop up window asking you to install packages needed to run knit, but once that’s installed, then you will automatically knit your document, and you’ll compile your R Markdown file, and a resulting document will pop up. Here’s what it looks like:

1. HTML Knitr repot



Note that this produces an HTML document. You can see it in your working directory. To see it, open a terminal and list the files. You can open the file in a browser and use it to present your analysis. But you can also produce a PDF or a Microsoft document. You can do this by changing the output variable in the header. So instead of output HTML document, you can write PDF document. We can also produce documents that render on GitHub using the output **github\_document.**

1. github\_document production in RStudio



There’s a lot more to learn about R Markdown. We highly recommend you learn as you gain more experience by writing reports in R. There are many free resources on the internet to learn more about R Markdown and knitr like this website:

<Rmarkdown.rstudio.com>

<Yihui.name/knitr/>

1. Section 4: Git and GitHub
   1. Git and GitHub
      1. Git and GitHub

We already learn how to use Git and Github with RStudio. We’re going to learn some more details and how to use the command line. However, we’re only scratching the surface. To learn more about the topic, we highly recommend free sources like those:

[Codecademy](https://www.codecademy.com/)

[GitHub Guides](https://services.github.com/)

[Try Git tutorial](https://try.github.io/)

[Happy Git and GitHub for the useR](https://happygitwithr.com/)

They are 3 mains reasons to use Git and Github:

The first is version control. The version control of Git permit us to keep track of changes we make to our code. We can also revert to previous version of files. Git also permit us to create branches in which we can test out ideas then this decide if we can merge the new branch with the original.

The second reason is collaborating. Once you set up a central repo, you can have multiple people make changes to the code and keep version synced. GitHub provide a free service for centralized repos. GitHub also has a special utility called pull request that can be used by anybody to suggest changes to your code. You can easily either accept or deny these request.

The third reason is to share, and this is the main one we use here. Even if we do not take advantage of the advanced and powerful version control functionality, we can still use Git and GitHub to share our code. We have already shown how we can do this with RStudio. Here we focus on the sharing aspects of Git an GitHub, and we refer to the links we just showed to learn more about this powerful tool. To effectively permit version control and collaboration, in Git files move across four different areas shown here. But how does it all get started? They are 2 ways to do this:

Clone an existing repo

Initialize a new repo

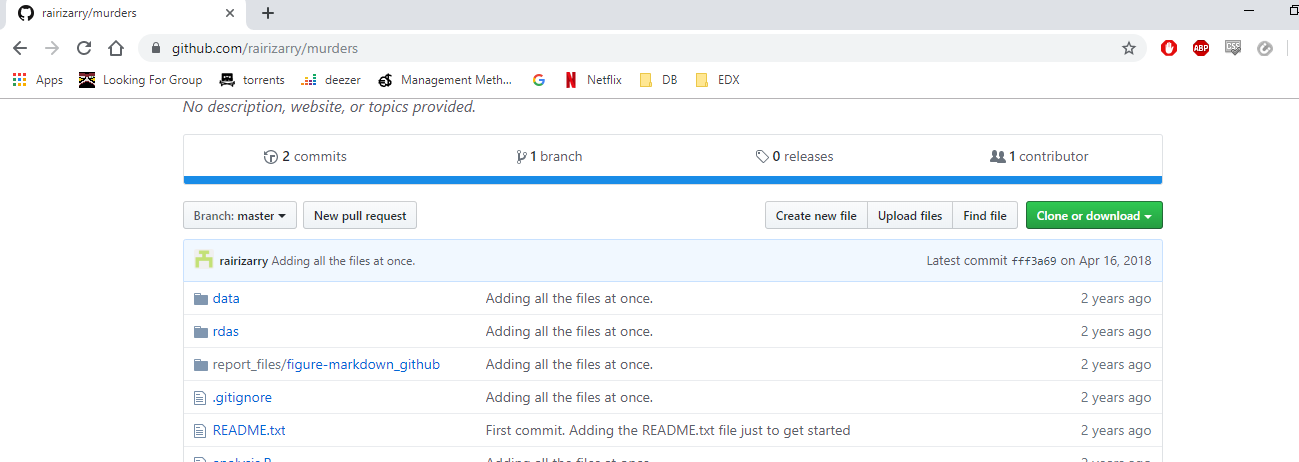
We will try the first approach first.

* + 1. Using Git at the Command Line

We’re going to clone an existing upstream repository. We can see it on GitHub by following this link. By visiting this page:

<https://github.com/rairizarry/murders>

1. GitHub upstream repo



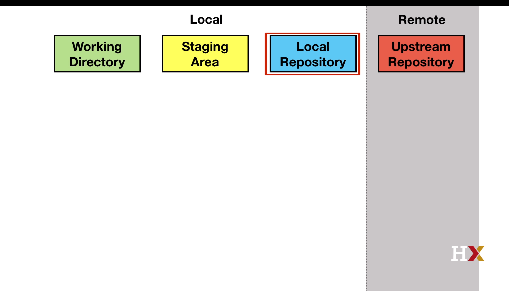
you can see that there’s multiple files and directories. This is called the upstream repository. When you go on GitHub, you will see a green clone button that you can click to copy the repos URL, which is :

<https://github.com/rairizarry/murders.git>

But what does clone mean? Rather than download all these files to your computer, we are going to actually copy the entire git structure, which means we will add the file and directories to each of the three local stages:

Working directory, Staging area, Local repository

1. Git process model



So, when you clone, all three are exactly the same. You can quickly see an example of this by doing the following, open a terminal and type:

*$ mkdir git-example*

*$ cd git-example/*

*$ git clone https://github.com/rairizarry/murders.git*

Now we can cd into the new directory. Which is called murders. You have now cloned a GitHub repo and have working git directory.

If you type ls, you can see all these files and subdirectories. So now, we have done this. Now the working directory is the same as your Unix working directory. When you edit files using an editor such as RStudio, you change the files in this area and only this area. Git can tell you how these files relate to the version of the files in other areas with command git status. If you check the status now, you will see that nothing has changed:

*git status*

*On branch master*

*Your branch is up to date with 'origin/master'.*

*nothing to commit, working tree clean*

So, we are now going to make change that we eventually want to be synced to the upstream repo. But we do not want to do this until we are sure that these are final enough versions to share. However, we can keep track of changes we male on our local directory before pushing these files to the upstream repo. But we also want to avoid keeping track of too many changes in the local version. We don’t want every little change. We really want changes we think are worth tracking. Edits in the staging area are not kept by the version control system. We add a file to the staging area with the **git add** command, as an example:

*$ echo "test" >> new-file.txt*

*$ echo "temporary" >> tmp.txt*

Here we create a file using the echo command, all this does is create a file that has a the word “test” in it. We also created a temporary file. So now, we can stage the file we want in our repository. To do this, we type:

$ *git add new-file.txt*

If you check the status now with git status you get:

*$ git status*

*On branch master*

*Your branch is up to date with 'origin/master'.*

*Changes to be committed:*

*(use "git restore --staged <file>..." to unstage)*

*new file: new-file.txt*

*Untracked files:*

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

*tmp.txt*

Now you get a different message that tells you that there’s a new files stage that has not been committed yet, and there’s another file that is untracked. The temporary file.

Now any changes we make to newfile.txt will get added to the repository next time we commit. We commit using the **git commit** command. Now whenever you commit, you have to add a message stating what you’re doing. In this case, we’re going to use the message “adding a new file name” :

*$ git commit -m "adding a new file"*

*[master 271e5ea] adding a new file*

*1 file changed, 1 insertion(+)*

*create mode 100644 new-file.txt*

We have now changed the local repository. If we type git status, we see this:

*$ git status*

*On branch master*

*Your branch is ahead of 'origin/master' by 1 commit.*

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

*Untracked files:*

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

*tmp.txt*

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

Now remember, if we edit that file again, it only changes in the working directory. To add to the local repo, we need to stage it and the commit the changes, and this will be added to the local repository. So let’s make a little change. We’re adding a new line to that file. We’re going to use the echo command again:

*$ echo "adding a line" >> new-file.txt*

Once we do that, in order to commit, we first stage it, then we commit, and now we can see, by looking at the status, that we have a made a change. This step is often unnecessary in our use of git. We can skip the staging part if we add the file name to the commit command:

*$ git commit -m "minor change to new-file" new-file.txt*

*warning: LF will be replaced by CRLF in new-file.txt.*

*The file will have its original line endings in your working directory*

*warning: LF will be replaced by CRLF in new-file.txt.*

*The file will have its original line endings in your working directory*

*[master 5c106da] minor change to new-file*

*1 file changed, 1 insertion(+)*

Now we can keep track of all the changes we have made by using the git log command. If we do it with the newfile.txt, we can see all the commits that have been done to our local repository. You can see it here:

*$ git log*

*commit 5c106da2fd82a6b91a62291edec7ccbc35a8ca27 (HEAD -> master)*

*Author: James Strayer <strayerjja@gmail.com>*

*Date: Fri Nov 1 07:18:45 2019 -0400*

*minor change to new-file*

*commit 271e5ea11d84974504246600e3e59d6e3a33177d*

*Author: James Strayer <strayerjja@gmail.com>*

*Date: Fri Nov 1 07:12:24 2019 -0400*

*adding a new file*

*commit fff3a6905a7d281ac29650be40674ceeee1fe89d (origin/master, origin/HEAD)*

*Author: Rafael Irizarry <rafa@jimmy.harvard.edu>*

*Date: Mon Apr 16 16:39:48 2018 -0400*

*Adding all the files at once.*

*commit 871af5d3e178403048f905fcc43a89235eb16eea*

*Author: Rafael Irizarry <rafa@jimmy.harvard.edu>*

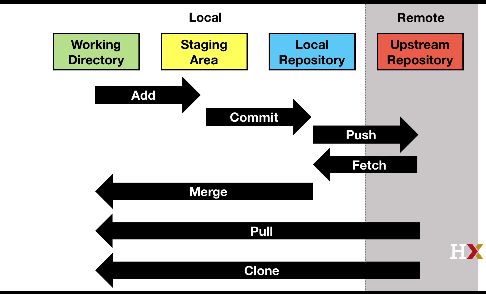
*Date: Mon Apr 16 16:37:10 2018 -0400*

*First commit. Adding the README.txt file just to get started*

Now, we have made changes to the local repository, and now we want to push these changes to the upstream repository the GitHub repository. This is done using the **git push** command. We simply type git push

However, you will not be able to do this because you don’t have permission to edit the upstream repo. You would only be able to do this if it was your repository. Now if this is a collaborative project, the upstream repo may change and become different that our own version. So, to update our local repository to be like the upstream repository we use the command **git fetch** and then to make these copies to the staging and working directories, we use the command **git merge**. That will change the files in our working directory. However, we often just want to change everything in one shot. So, for this, we can use **git pull** that is equivalent to git fetch followed by git merge. Here is the details of what happens:

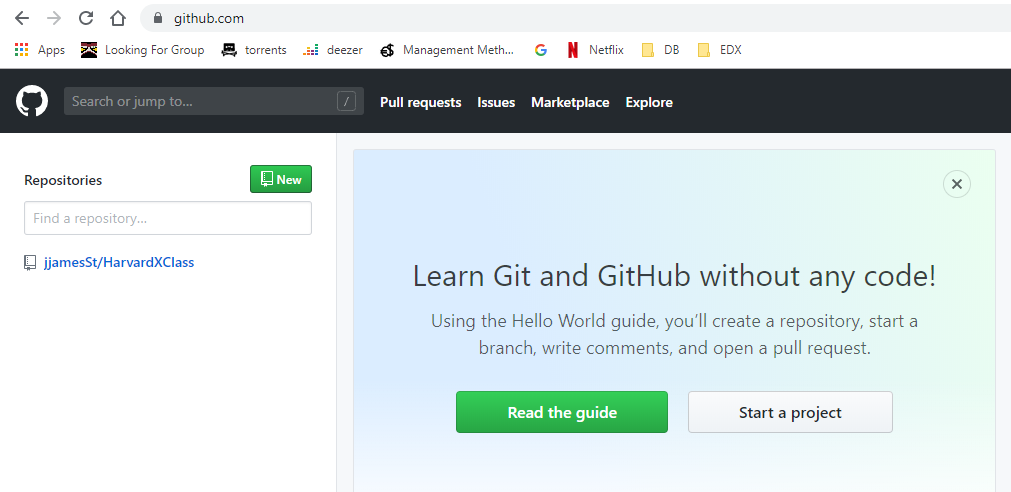
1. Model of different git command



* + 1. Creating a GitHub Repository

We going to learn a second approach we can use to get started. That is by initialising directory on our own computer rather than cloning. We will show how to create the GitHub repository for the gun murders project. We first created a project on our computer independent of Git. So we already had all the files and all of the directories ready, but we did not yet have a Git local repo or a GitHub upstream repo. We start by creating a new repo on our GitHub page. This is the upstream repo. So we go to GitHub, we click on a repository, click New. And we going to call it murders\_analysis to match the name of our project:

1. GitHub new repo creation



We need to copy the URL of the repo given to us by GitHub, which represent our upstream repo. Now we must create a local repository. We’re going to do it in the directory that already includes all the files and directories for our murders project. So let’s start by opening a terminal a moving into the directories that contains the project:

*$ cd murders\_analysis*

Now, this is a working directory in our system. Git knows nothing about it. We need to make it a local Git directory. To do this we need to initialize it, by typing:

*$ git init*

We turn the working directory into a Git directory and Git starts tracking everything. Now, the files are there in our working directory, but they’re nowhere else. We haven’t done anything. We haven’t staged, we haven’t committed. So there’s two next steps to do. One is to start moving files into our local repo and another is to connect the local repo to our GitHub repo. We need to let Git know that it is going to be connected to GitHub repository. In a previous example, we had RStudio do this by starting the whole project as a Git project, but now we need to do it ourselves. So what we’re going to do is we’re going to do is to add one of the files just to get started. So we’re going to add the readme.txt file. We’re going to stage it, and then we’re going to do our first commit:

*$ git add README.txt*

*$ git commit -m "First commit. Adding README.txt just to get started."*

Now we have a file in our local repository, just one, just readme.txt. we’re ready to connect our local repo, which now has one file to our upstream repo. We’ve already an URL for the upstream repo, and the command to do this is “get remote”

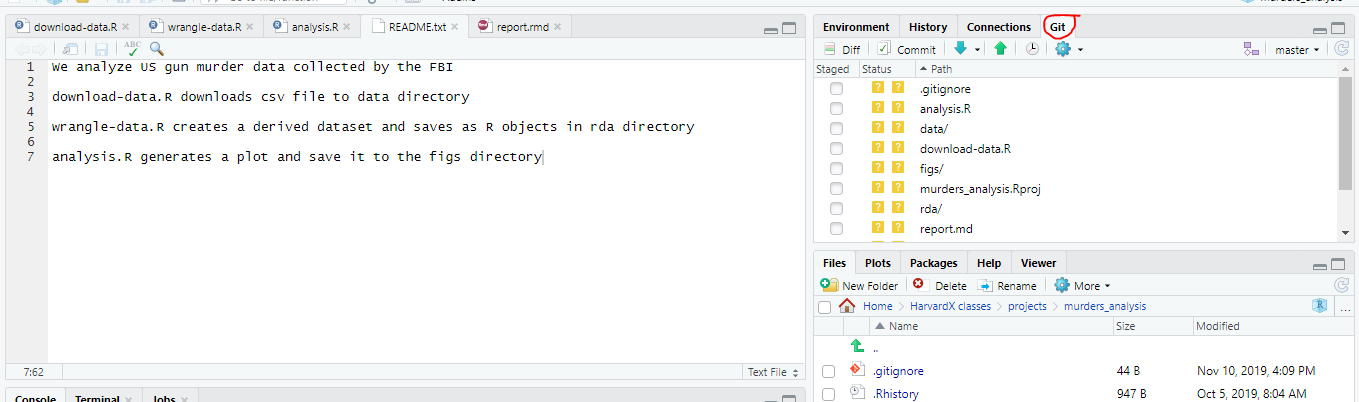
*$ git remote add origin https://github.com/jjamesSt/murders\_analysis.git*

Now we are connected. Now we have some files in our working directory. We have a file in our Git local repository and we also have a connection to our GitHub upstream repository. We’re going to push the file we have up to our GitHub upstream repo. So we type git push and this should do it

*git push*

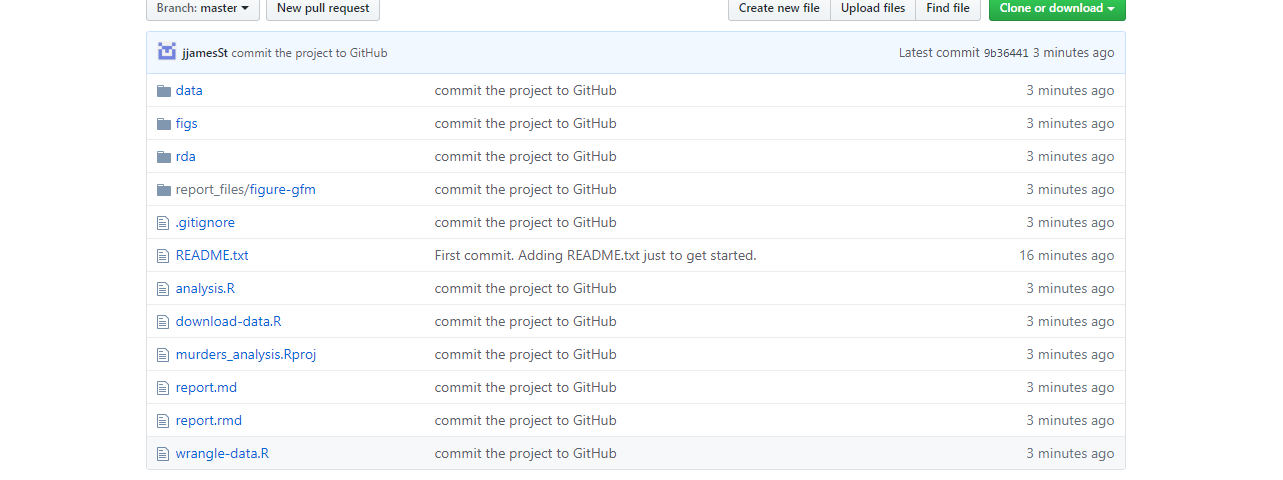
Now, we can continue adding and committing each file one by one. It’s not that hard using the command line, but we’re going to go ahead and do it in RStudio. We find this easier. So to do this, you start the project by opening the R project file. And Then the Git icons should appear, you can see them here:

1. Git tab of a R project



So we just click them all, we hit the commit button and then we hit the push button and they all go up to GitHub and you can confirm this by going to your GitHub web page

1. Commit and push project files on GitHub



1. Section 5: Advanced Unix
   1. Advanced Unix Part 1
      1. Advanced Unix: Arguments

Most of Unix implementations include many powerful tools and utilities. We have just learned the very basics here. Let’s start by describing arguments. Most Unix commands can be run with arguments. Arguments are typically defined by using a dash or two dashes, followed by a letter or a word. And example of an argument is the -r in front or rm. The “r” stands for recursive and the result is that all files and sub directories will be removed without possibility of recovering so:

*rm -r directory-name*

is equivalent to putting the directory in the bin but without being able to recover it.

Often when you’re removing directories, you will encounter files that are protected. To remove protected files, you will have to use the force argument. This is -f, you can combine arguments so for example to remove the directory regardless of protected files, you can type:

*rm -rf directory-name*

**Remember once you remove the directory there is no going back.**

Another command that is often called with arguments is ls. For example, ls -a, the -a stands for all. This argument makes ls show you all files in the directory, including the hidden files. In Unix, all files starting with a dot are hidden. Many applications create hidden files or directories to store important information, but without getting in the way of you work. An example is git. Once you initialize a directory as a git directory a hidden directory called “.git” is created. Another files is the .gitignore file. Now, another example of using arguments with ls is -l. The l stands for “long” and the result is that more information about the file is shown. Yet another example that is often used is the -t argument. This is to put files in chronological order, often we want to reverse the chronological order (to have the last created files for example), so we use the -r argument with ls.

Now we can combine all of that, to show the more information for all the files in reverse chronological order, we can type:

*ls -lart directory-name*

* + 1. Advanced Unix: Getting Help and Pipes

Unix has an extreme version of abbreviation; this makes it very efficient but hard to guess how to call commands. To make up for this weakness, Unix includes complete help files or man pages. Man is short for manual. In most systems, you can type man followed by the command to get help. So for ls, we would type “man ls.” This command is not available in some of the compact implementations of Unix such as Git Bash. An alternative way to get help that works on Git Bash is to type the command followed by –help. So for ls, it would be:

*ls –help*

The help pages are typically long, and if you type man ls, for example, you’ll have to scroll up and down to see all the help file. It would be nice to save the results into a file and then use less to see it. The pipe, | , does something similar. It pipes the result of a command to the command after the pipe. This is similar to the %>% command we learned in R. So to get more help, we can type:

*ls –help | less*

This will pipe the result of help into less and we can scroll a little bit easier. It is also useful when listing files in a directory that has many files, so we can type:

*ls -lart | less*

* + 1. Advanced Unix: Wild cards

Suppose we want to remove all the temporary HTML files produced during troubleshooting for a project. Imagine there’s a dozens of files. It’ll be quite painful to remove them one by one. In Unix, we can write an expression that means “all the files that end in HTML.” It uses the asterisk wild card. So, to list all HTML files, we will type

*ls\*.html*

This will list all the files ending in HTML. To remove all HTML files in a directory, we’ll type:

rm \*.html

The other useful wild card is the question mark symbol. This means “any character”. So if all the files we want to erase have the form file-001.html and we know there’s three characters, then we are going to use

*rm file-???.html*

This will only erase files that have this very specific format. We can combine wild cards too. For example, to remove all files with the format file-001, for example, regardless of the suffix, we could type

*rm file-???.\**

**Warning: Combining rm with the \* wild card can be dangerous. There are combinations of these commands that will erase your entire file system without asking you for confirmation. Make sure you understand how it works before using this wild card with the rm command.**

* 1. Advanced Unix Part 2
     1. Advanced Unix: Environment Variables and Shells

Unix has settings that affect your command line environment, these are called environment variables, the home directory is one of them. We can change some of them. In Unix, variables are distinguished from other entities by adding dollar sign in front. The home directory is stored in $HOME.

So, we can wee our home directory by typing:

*echo $HOME*

You can see all the environmental variables by typing *env*. You can change some of this, but how you change them varies across different shells.

Much of the commands that we’ve been running are part of what is called the Unix shell. There are different shells. If you want to know what your shell is, you can get it by typing:

*echo $SHELL*

it’s an environmental variable. The most common shell is **bash**. Once you know the shell you can change environmental variables. In bash shell, we do this using:

export val value

where val is the variable we want to change and value the new value we want to give it. To change the path you would type:

*export PATH=/usr/bin/*

and then a directory, Now **don’t run this command on your computer**, this is just an example. Now there is a program that is run before each terminal starts, where you can edit variables, so they change whenever you call the terminal. This changes in different implementations. But if you are using bash, you can create a file called :

.bashrc or bash\_profile or bash\_login or .profile

And add commands there that run every time you start a terminal.

* + 1. Advanced Unix: Executables, Permissions, and File Types

Another important thing to know in Unix is that there are executables. In Unix, all programs are files, but the programs the ones that run, are called executables. So ls, mv, git they’re all files. They’re executable files. But where are these program files?

You can find out using the command “which”. If you type

*which git*

it will show you the directory in which this file resides. In most systems, this will be /usr/bin, is probably full of programs. You can look at them by typing :

*ls /usr/bin*

Now the applications directory in the Mac, or the Program Files in Windows, those are also examples of directories that have executables. Now note that when you type ls, Unix somehow knows to go run a program that’s in another directory. How does it know this? This information is included in the environmental variable path. If you type:

*echo $PATH*

you will see a list of directories separated by a colon “:”. The directory /usr/bin is probably one of the directories in that list. Unix looks for program files in those directories in the order they appear in the path variable. Although we don’t teach it here, you can actually create executable files yourself, but if you put in your working directory, and this directory is not on the path, you can’t run it just by typing the command. One way to get around this is by typing the full path. So if your command is called my\_ls, you can type:

*./my-ls*

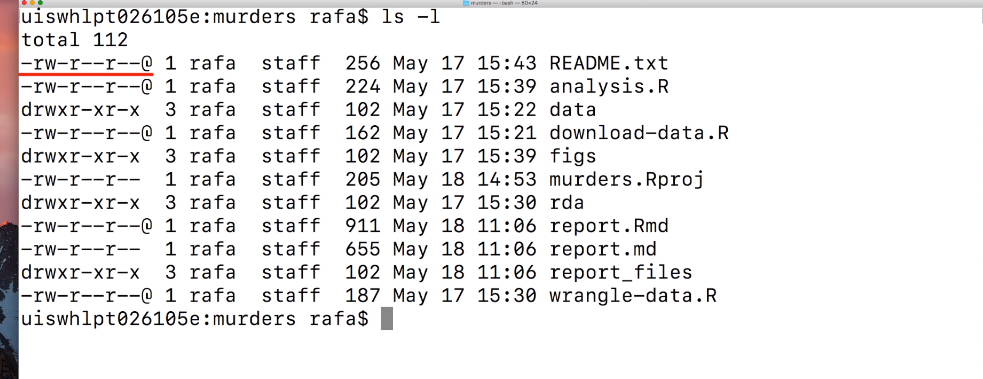
And it’ll learn. Learning how to write your own executable is useful but not shown in this class.

One last thing we want to point out are the permissions and file types. If you type:

*ls -l*

at the beginning, you’ll see a series of symbols like this:

1. permissions of files in the working directory



The string indicates the type of files each file is. So a regular file is a directory or is an executable. The string also indicates the permissions of each file, is it readable? Is it writable? Is it executable? Can other users on the system read the file? Can other users on the system write to the file? Can other users execute the file if it’s an executable? This is more advanced than what we cover here, but you can learn much more in a Unix reference book.

* + 1. Advanced Unix: Commands You Should Learn

This chapter is to talk about Unix command that you should know but we will not go into details.

* Start (open on Mac)

This command try to find the best application for the file you want to open, for example, if you try to open an R or RMD file with start it should open RStudio.

* Nano

This open a bare-bones text editor to read a change your file

* Ln

Creates symbolic links. It is not recommended to use symbolic links, but you should learn what this is anyways

* Tar

Let’s you create or extract archives, which is very useful when you want to put a very complex file structure into a single file

* Ssh

Let’s you connect to other computers

* Grep

Let’s you search for patterns in a file

* Awk and sed

Are commands that permit you to find specific strings and files and change them.