Introduction to PlainText Software for Research and Writing

Timothy B. Elder

PlainText
Working
Group

91cca23

Acknowledgements

Support for this work was generously provided by the Center for International Social Science Research at the University of Chicago and Jenny Trinitapoli in particular. And thanks to the attendees to the PlainText Working Group.

Contents

List of Figures

1 What? Why? How?

1.1 What

Academic writing in general, and scientific and social scientific writing in particular, requires that we use empirical data to mount an argument. There are many conventions that have become sacrosanct that we both need to use and find quite helpful for fulfilling the task of providing convincing data and a tight argument, grounded in a long line of other worthwhile research. Citations, figures, tables, equations, proofs, diagrams and prose are all key to this endeavor. Most undergraduate and graduate students, postdocs, early, mid and late career faculty do most of their writing in What You See is What You Get style programs like Word or Google Docs. They are easy to use and you already know how ot use them but they were not made with scientists and social scientists in mind. Anyone who has had to format a bibliography, update their in-text citations, re-number figures, or properly format a table in a Word document will quickly realize the limitations of Microsoft's flagship Office program.

In this document, I am going to give you an introduction to some of the most helpful tools for adopting a plaintext tool kit for conducting and writing your research, which was designed with researchers in mind who need to leverage all these conventions in their work. It will require that you actually write code, install programs, manipulate text and images to compose documents and then typeset them. If you are familiar with a programming language like python or R, you will be well positioned to take advantage of these tools. If you are not familiar with how your computer works, you will have to do a little extra work but I will attempt to make the burden of learning how to use these tools a little less onerous. But what is plain text?

A plain text file is a file where the contents are human readable, and do not contain any information other than the alphanumeric characters in the file itself. Plain text is primarily distinguished from a formatted text file, such as a Word Document. For example, if you open a Word Document that has italicized, bolded, underlined, different type faces, and margins, the instructions for how that text is formatted is not immediately rendered to you, just the *formatted text* is. In a plain text file containing the same information, the styling

conventions are readable to you as they are encoded in the text along with the prose. The distinction is not easy to immediately capture but will become clear very quickly, the more important question to address is why would you be interested in learning anything about plain text files and software.

1.2 Why

I have colleagues who really don't understand my obsession with plaintext software. Most think it is the product of a desire for esoteric and complicated things, a severe and sometimes productive neurosis. They think that it is actually just harder to use Markdown, LaTeX, and `pandoc`, that Word is enough, and that when something is poorly formatted or not particularly pretty then the words and argument have to speak for themselves. I used to be willing to concede the point that just using Word is easier, but it actually *isn't* easier. Yes, you already know how to use Word, but Word wasn't designed with scientists and social scientists in mind, people who have to communicate very complicated things in concise and elegant ways, taking advantage of all the contemporary tools we have at our disposal.

It is in fact easier to do what I will teach you in this document. No one wants to emulate a document that looks like it was written in Word. What more, one day the people who read your work and take it seriously might just want to replicate it. Maybe they'll ask you how a particular figure was made, or what process you adopted to generate a particular regression table. If you have a PDF file for the submitted article version called `my_fabulous_paper_vFINAL_2.pdf` and 20 different Word documents all with slightly different versions of the same name you will lament the times you thought writing in Word was "easier".

To give just a brief introduction to why you should make the effort to switch to plaintext tools for research and writing I will say there are three reasons:

1. They Are Free: All the tools I will outline here are free to use, have a wide user base, and keep your data and writing in an accessible and non-proprietary format.

2. They Encourage Good Practices: Working with plaintext tools makes you interact with your data and writing files in a way that encourages good practices at the expense of bad ones. I will say more about these shortly so you will have to have sufficient faith in the project to get through chapters 1 and 2 to be truly convinced but convinced you certainly shall be.

3. They Make Your Work Accessible: You'll make documents that are reproducible and allow you to distribute the means to replicate your work with ease.

These tools are not just for quantitative scientists and social scientists. They were originally written for scientists who needed to use a lot of math, but they are now quite useful to scientists and social scientists of any methodological commitment, particularly considering the other important feature of a plain text workflow, its synergy with reproducibility and openness.

You might be surprised to learn that I am actually a qualitative sociologist who specializes in interviews and ethnographic data collection.

Several other social scientists have noted the usefulness and superiority of plaintext tools for research and writing. My extended introduction is hardly a substitute for some canonical texts on the subject. I myself became an acolyte when I found Kieran Healy's *The Plain Person's Guide to Plain Text Social Science* (2020) in my second year of graduate school. I recommend reading it as it outlines in a far more parsimonious and convincing way the *why* of why make the switch to a set of software that is a little more onerous at first to use but rewarding once it is mastered. The document you are looking at here is superior in only one regard: it takes you step by step through some of the more dizzying steps in the process of switching to this tool-kit, and is geared toward the graduate student who knows how to use their computer but hasn't made a lot of effort to learn the more archaic and powerful tools already available to them.

1.3 How

For complete beginners, people who are proficient at using their computer but have never opened up a terminal, review Chapter 2 and 3, before going to Chapter 4. If you are familiar with how to use the terminal then you can briefly checkout Chapter 3 before going to Chapter 4 to install the software that we are going to be using. Installing the software is the first big jump to adopting plaintext software for your research and writing and it is a little complicated.

Once you have the software installed you will be ready to start learning how to actually use it in Chapter 5, where I will show you all the most important conventions in LaTeX and Markdown before showing you how to typeset your first document. After that you can proceed to Chapter 6 to learn how to use dynamics documents and RMarkdown in particular. At that point you will be ready to learn how to keep a record of everything you've done and make it available to others using `git` and Github in Chapter 7.

2 The Terminal

Switching to a plain text workflow requires that you adjust the way you interact with your computer. Rather than pointing and clicking and navigating through different windows we will be using what is a primitive technology in computing: the Terminal. It was once the case that all interactions with a computer were done with nothing more than a keyboard, not even a mouse. Of course computing has come a long way and most of your interactions with your computer take place with a mouse, touch pad, or touch screen, and it can be intimidating to approach the terminal.

Using the terminal has some distinct advantages over navigating through the file explorer that comes with your operating system and will help to get you familiar with how to control the software we will be working with. For one, the terminal is actually quite easy to use and straight forward after learning just a few basic principles. Further, navigating your computer with the terminal helps to instruct you in the file structure upon which the programs you interact with relies. Understand this file structure will then help you to both organize your project and typeset documents. Here I will provide a quick tutorial on how to use the terminal with some of the most basic and essential commands.

Windows and macOS use different kinds of terminals, and even in macOS there are slightly different versions of the same terminal. This means that you will have to learn commands for your operating system. Also, they are called slightly different things. On Windows it is called "Command Prompt" and there is a specific program called PowerShell. Don't use PowerShell, use Command Prompt, sometimes referred to as CMD which can be found in the "Accessories" part of the Start Menu. On macOS you will find the terminal under "Other" in your Launchpad. Here is a table with commands across operating systems:

Table 2.1: Equivalent commands for the terminal in Windows and macOS.

Windows CMD	Task	macOS Terminal
`dir`	List files and folders	`ls`

Windows CMD	Task	macOS Terminal
`cd`	Full path of current folder/directory	`pwd`
`cd <path to directory>`	Change folder/directory	`cd <path to directory>`
`cd ..`	One directory up in directory tree	`cd ..`
`mkdir newFolder`	Create new directory in current directory	`mkdir myFolder`
`rmdir myFolder`	Remove a directory*	`rmdir myFolder`
`ren oldFolderName newFolderName`	Rename a directory	`mv oldFolderName newFolderName`
`robocopy myFolder <path to destination directory>`	Copy a directory	`cp -r myFolder <path to destination directory>`
`move myFolder <path to destination directory>`	Move a directory	`mv myFolder <path to destination directory>`
`ren oldFileName newFileName`	Rename a file	`mv oldFileName newFileName`
`copy myFile <path to destination directory>`	Copy a file	`cp myFile <path to destination directory>`
`move myFile <path to destination directory>`	Move a file	`mv myFile <path to destination directory>`
`cls`	Clear the terminal screen	`clear`

2.1 A Short Tutorial

When working in the terminal you will be dealing with *Files* and *Directories*. A directory is what is usually called a "folder", a container for files. A file is the basic unit for holding data in your computer. A file is the thing that you typically open with your cursor (like a text file, Word document or an image file) by double clicking on it. We will be dealing with programs on the terminal but not in the way you are familiar. We will get there eventually, but just remember files go in directories and directories can themselves have sub-directories with files in them. This is the hierarchical folder structure that is

Figure 2.1: The Terminal in macOS

ubiquitous in computing.

With the terminal open go ahead and type in `ls` and hit `Enter`. In macOS you should see something like this print out:

```
(base) MacBook-Pro:~ timothyelder$ ls
Applications
Desktop
Documents
Downloads
Library
Movies
Music
Pictures
Public
```

The `ls` or "list' command in macOS lists the contents of the current directory you are in. To determine which directory you are in type in `pwd` and hit `Enter`.

```
(base) MacBook-Pro:~ timothyelder$ pwd
/Users/timothyelder
```

`pwd` stands for "present working directory" and prints out the path to the directory you are in. The "working directory" just means whatever directory your terminal is open in. On macOS, whenever you open a terminal it automatically opens in what is known as your

"Home" directory which has the files that appear on your Desktop, in your Documents directory and other directories associated with Videos, Music, etc.. A path is the generic way of referring to the address of a directory or file on your computer. Let's start to navigate your computer and manipulate directories and files from the terminal.

To navigate to another directory from your working directory use the `cd` or "change directory" command and specify which other directory you want to change to. Let's change to the Documents directory by typing in `cd /Users/timothyelder/Documents`. This is the path *for me*, as my user name on my machine is "timothyelder", so you'll have to use *your* username or whatever is in the path when you use the `pwd` command. After using the `cd` command to get to the Documents directory let's create a sub-directory there to store the files related to the PlainText Working Group. To do this you will use the `mkdir` or make directory command. Type `mkdir plaintext` into the terminal and hit `Enter`. Check to see that the new directory has been made by typing in `ls` again and see that the directory has been made.

Now, when using the `cd` command (or any other command that takes in a path as an argument) you can use either *absolute* or *relative* paths to specify where you want to go or what file or directory you are specifying. An absolute path uses the full amount of information to describe the address of the file or directory you are referring to (think of them as latitude and longitude), such as `/Users/timothyelder/Documents/plaintext`. That is the absolute path of the `plaintext` directory. Using the absolute path makes everything explicit, but takes up a lot of time when you have to type it in over and over again into the terminal. To save yourself time you can use a relative path which is relative to wherever your terminal is open on your computer (think of these as generic indexical directions, "around the corner", "take a left at the light", or "across from the 7/11"). For instance, if you did the last set of instructions correctly, you created a directory called `plaintext` in the `Documents` directory and we noted the absolute path above. The `plaintext` directory is immediately accessible to the `Documents` directory because the former is a sub-directory of the latter, so simply typing in `cd plaintext` will move your terminal into the `plaintext` directory.

Go ahead and `cd` into the `plaintext` directory and type `ls` again. As you'll see, nothing is printed out from the list command because it is a brand new directory with no files or subdirectories. Next thing to do is to create an example text file. To do this we are going to use a built-in text editor to create a new file using the `vim` command. `vim` is an ancient text editor that is pretty much built into all machines that are based on UNIX which includes macOS and Linux. Into the terminal type `vim my_text_file.txt`. The command `vim` is used to open a text editor in your terminal and you have just used it to open a file called `my_text_file.txt`, and because the file doesn't yet exist, you are creating it at the same time.

This can be very confusing because it looks like an empty terminal window, as can be seen in Figure 2.2.

Figure 2.2: An instance of an empty file opened with `vim`

The terminal is now open in an empty text file, and if you start tapping away at your keyboard nothing will happen, which is also pretty mysterious behavior. To edit the file and add content you need to press the `i` key on your keyboard. This activates "insert" mode in the `vim` text editor meaning you can actually type in the window and put content into the file. This will look like Figure 2.3. Type in "Hello World!" then hit the `esc` or escape key on your keyboard and you will exit the insert mode, then type in `:wq` (that is hit `Shift` - `;` and then type `wq`). Typing in `wq` means "write-quit" which is "write the file contents to memory and exit the editor". To exit without saving use `:q!` instead of `:wq`. Once back to the normal terminal type in `ls` to check that the file is there, and then type in `cat my_text_file.txt` and the file contents will print out. The `cat` command (besides being a cute reminder of our Feline friends) stands for "concatenate and print file contents" and allows you inspect plain text files from the command line.

Though we are not going to be using `vim` extensively it is good to know how to use it, particularly considering how disorienting it can be when a program pops you into a `vim` terminal and you've never seen one before. All digital writing was once conducted in things like `vim`, and other text editors, a class of programs that allows the user to create and edit plain text data. You could do nearly everything we are going to do in the working group with `vim` or an equivalent terminal based text editor. You could write a whole book

Figure 2.3: Insert mode in `vim`

in it if you wanted, or the documentation that you are looking at now (see Figure 2.4)

Lastly, the terminal lets us take a look at *hidden* files in a directory. Do the exact same thing as you did above (where you created a text file called `my_text_file.txt` with "Hello World!" inside it) but this time when you first type in the `vim` command, instead of `my_text_file.txt`, type `.hidden_file`. Make the file contents the "Hello World!" phrase, same as before and write quit out of the file. Back at the normal terminal type `ls` again to make sure the file you just created is there. Curiously, you will not see a file called `.hidden_file` but the `my_text_file.txt` will be there! You can even check in a normal Finder window or File Explorer and the file will not be there.

```
(base) MacBook-Pro:plaintext timothyelder$ ls
my_text_file.txt
```

This is because files that begin with a period are hidden and do not appear without using a special *flag* or *option* for the `ls` command. Typing in `ls -a` will printout *all* the files in the directory, even hidden ones.

```
(base) MacBook-Pro:plaintext timothyelder$ ls -a
.               .hidden_file
..              my_text_file.txt
```

There is nothing special about any given directory that you can navigate to on your com-

Figure 2.4: Composing this documentation in `vim`

puter. They are all generic containers that store generic files and so you can take what you have applied here and move up and down the directory tree, listing out the files and creating files as you please.

2.2 A Few Helpful Hints and A Warning

When using the terminal if you ever need help with a command you can look up what's called a `man` page, or manual page simply by typing in `man <command of interest>`.[1] So if you want to read about everything the `ls` or `cd` commands can do simply type in `man ls` or `man cd` and the terminal prints out information that you can navigate through with the directional keys. If you need to exit a `man` page hit the `q` key on your keyboard.

Also, for the `cd` command, you can navigate into the parent directory of your working directory by typing in `cd ..`. For example, above we created a sub-directory called `plaintext` in our `Documents` directory with two files in it. If you were in the `plaintext` directory and typed `cd ..` that would take you one level up to the `Documents` directory. Doing the `cd ..` command one more time takes you up another level into your home directory where we started out.

[1] **Windows Users**: Use `help` instead of `man`.

Lastly, the terminal is intimidating but hopefully some of its mystery has been resolved now that you can navigate around it, list files out and make them all from the terminal. *But*, the terminal was made by computer scientists and engineers who were very technically capable and knew what they were doing, so when they typed in a command they knew what it meant and what it would do. Sometimes we can get ourselves into trouble on the terminal because we are not computer scientists and engineers and we don't always know what we are doing. For example, the `rm` and `del` commands (in macOS and Windows respectively) delete files, and when you run them they don't ask you to confirm that you really want to delete the file `only_copy_of_my_thesis_do_not_delete_no_backups.tex` and it doesn't go to the Trash folder for you to restore it later. It just gets deleted. So use caution on the terminal but for the most part you can't get into too much trouble.

3 Project Organization

Working with the terminal requires and instructs you about the file structure of your computer, knowledge of which is important for keeping things organized while you do your research and writing. Project organization refers to two separate and sometimes competing tasks: one is organizing the material you need to learn about the world in a way that is conceptually coherent and helpful to you. The second task is organizing actual computer files in such a way that they are accessible to you and to the software you are working with. These are distinct tasks and sometimes they compete with one another, as the first is meant to help you *think better* and the second is meant to help you *do things faster*. For example, it makes sense to keep distinct activities related to your research in different directories (sort of like keeping one notebook of notes for one class, and another notebook for another class). This organization is in part a matter of taste but there are certain organizing principles you should almost certainly avoid. For example, you could just have one directory that has all the files for your project in a flat structure, which makes remembering the paths to each file pretty simple (it would just be `/Users/timothyelder/Documents/project-dir/FILENAMEHERE.txt`). That makes everything accessible but doesn't help to keep conceptually or practically separable parts of your project organized. On the other extreme, you could have a byzantine file structure, where edits to documents are organized by activity (such as having folders for `interview_transcripts`, `field_notes`, `field_notes_reflections`, `interview_notes`, `draft_papers`), and then subdirectories organized by the date in which the files in that directory were edited or created. That would certainly heep thins organized but fairly messy when it came time to put them altogether in a write-up.

Finding a healthy compromise between the conceptual ordering of research material and the organization of computer files will be a matter of personal preference. With that said, I have a few principles that I find helpful and encourage you to emulate:

3.0.1 Each project should have its own directory.

A project can be as small as a side-project that goes nowhere, or your dissertation or even your *magnum opus*. A rule of thumb that I use is that once something takes on a distinct

character, and is relying upon more than 5 or so files to be coherent, it probably needs its own directory.

3.0.2 Each Area of a Project should have its own Sub-Directory.

Again what constitutes an "Area" is ambiguous and you have to use your own judgement and the final rule of thumb is if you are making progress and getting things done. I usually end up having the same directories at the top of my project directory. These include the following:

```
timothyelder@MacBook-Pro project-dir % ls
README.md
research_log.md
analysis
data
drafts
figures
memos
misc
scripts
```

The two files at the top of this list are what are called Markdown files, a simple markup language. The `README.md` file includes an explanation of what the project is, what data it uses, and any required software while the `research_log.md` includes entries about what is going on in the project and my laments about the world and my own research. The rest are sub-directories that contain different parts of the project. In the `data` directory naturally is all my data (both quantitative and qualitative), while the `drafts` directory has all the drafts for the different formal pieces of writing coming from the project. Informal pieces of writing that are likely only to be seen by myself, my advisors, or collaborators are stored in the `memos` directory. `figures` includes all the images that are generated from the data or that might be included in papers, slides, etc.. Whether the project is purely qualitative or not, I inevitably write some scripts in R, python or shell and they go in the `scripts` folder and `misc` is a garbage can of all the things I don't need now but am not confident I will never need again.

3.0.3 Hierarchical is Better than Horizontal

This goes hand in hand with the first two points but I want to emphasize that creating directories and sub-directories is helpful for keeping everything organized and that nesting directories is particularly helpful as it allows for more and more fine grained conceptual categorizations.

3.0.4 Literal is Better than Symbolic

> There are only two hard things in Computer Science: cache invalidation and naming things.
>
> — Phil Karlton

When you are naming directories and files, it is always best to make things explicit. If the directory holds data call it `data`. If the directory contains images of documents from the Florida Office of Insurance Regulation in Tallahassee, call the directory `images_florida_insurance_regulation`. If the file is your dissertation latex file call it `dissertation.tex`. Choosing symbolic names, or anything cryptic makes parsing files difficult, particularly if you take a break from a project for a few weeks and need to get back to it.

IMPORTANT NOTE: It is really important that when you are naming things, including files and directories, that you don't use spaces or slashes in the names. This is because spaces and other characters like slashes are read by your computer as meaningful to how it reads different files. If you use them then errors will be raised and things won't work.

3.1 Some Examples

Here are some examples that exemplify and defy the paradigm I articulated above. All of these are examples from my own computer and time in graduate school. Figure 3.1 is the directory that contains all the files for my Qualifying Paper, my first independently researched and written paper from graduate school. You will note two things: I was doing all my writing with Word and didn't abide by my rules about keeping different directories for different areas of the project. I do this a little bit having a `drafts` directory but I am not certain what the `Print` and `Material` directories are and the `Writing Seminar Submission` directory should probably be in `drafts`. Clearly, I was not being very thoughtful when it comes to how I organized my work, but this is a lot better than the directory in Figure 3.2.

This is a directory for class prep for a sociology course I helped to design along with Profs. Jenny Trinitapoli and John Levi Martin. It is a complete mess with obscure directory names and files that are all at the top of the directory without much organization at all. Files are misspelled and there is a particularly egregious organizational error. I have two files that are indistinguishable in their name except for the suffix ".v2" being included in the file name. A short anecdote will heighten the importance of the naming conventions I articulate here.

Figure 3.1: A poorly organized directory.

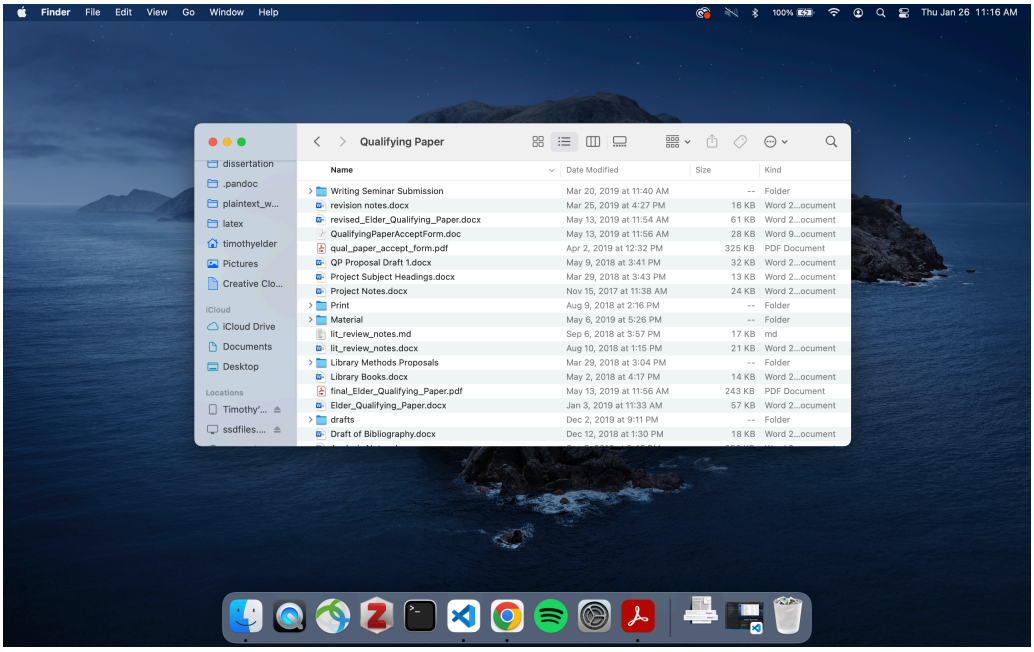
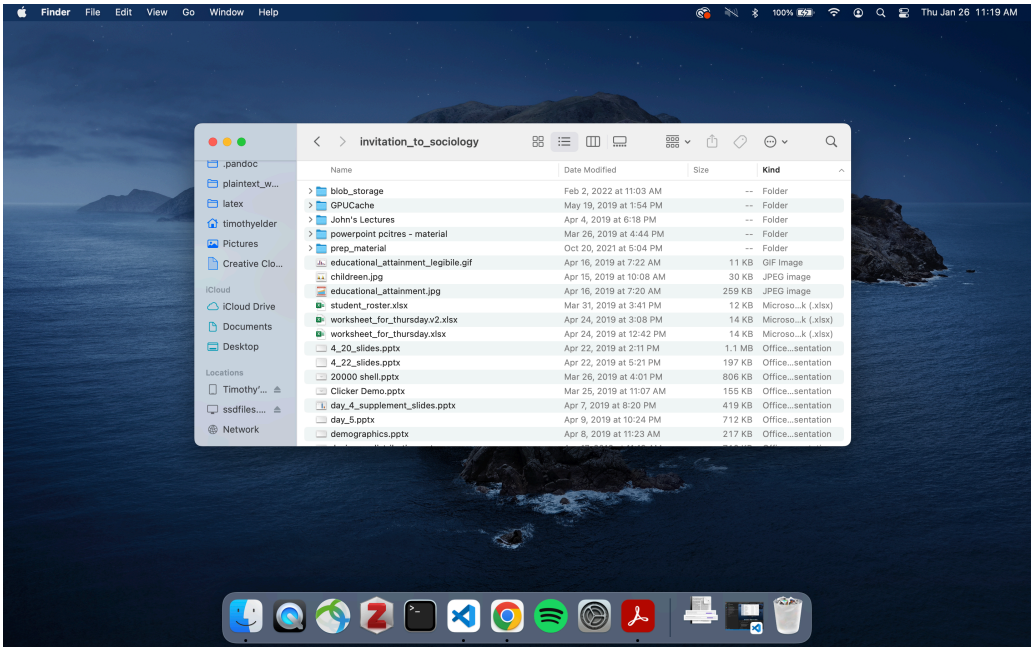


Figure 3.2: A very poorly organized directory.



I have a colleague who has had a paper under review for a couple of years now due to a variety of factors related to the pandemic. On the second R&R, after months of working on other projects and getting ready for job talks, he went back to the project files to address the concerns of a particularly scrupulous Reviewer #3. The reviewer was asking that they re-run some analyses with a different method and so my colleague needed to go back and figure out how a few figures were generated and how the original analyses were specifically conducted. Opening the directory with his code he had endless files all with nearly indistinguishable names like:

```
pol_gss_multiimput.R
pol_gss_multiimput_v2.R
pol_gss_multiimput_v2_1.R
pol_gss_multiimput_v2_2.R
pol_gss_multiimput_v2_2_THIS_ONE.R
pol_gss_multiimput_v2_2_No_REally_THIS_ONE.R
pol_gss_multiimput_v2_2_No_REally_THIS_ONE_final.R
```

He spent weeks and lots of tireless hours figuring out what was what. Instead of being like my friend, be more like me and do something like you see in Figure 3.3. This is a directory for a project that I am working on that includes lots of different scripts and data. Code for processing the data is kept separate from code for doing analyses and files are given explicit and non-redundant names. Certainly, this takes some amount of effort and energy to do and my project directories don't always look like this while working on them, but it is worth cultivating good habits like this to do your research and writing. You will thank yourself later if you are ever in the same position as my poor sweet friend being harassed by pesky Reviewer #3.

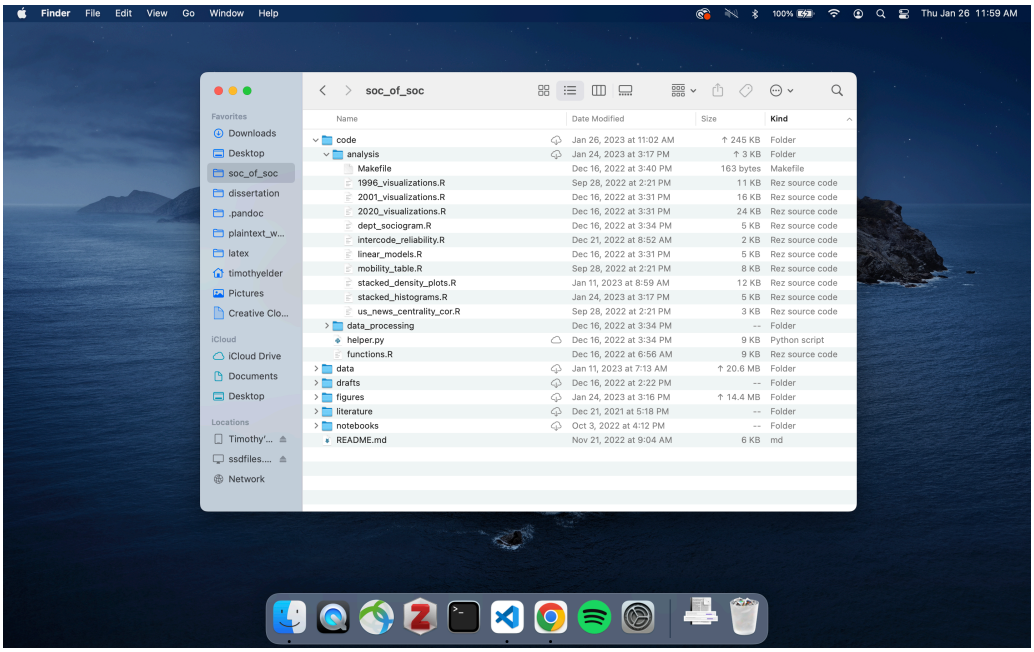
3.2 Summary

The wisdom then for organizing a directory in a way that successfully achieves the two different goals of organization (conceptual clarity and organizational productivity) can be condensed into:

1. Project in Every Directory and a Directory for Every Project
2. A Sub-Directory for every Area of a Project
3. Hierarchical is better than horizontal organization
4. Literal names are always favored over cryptic or symbolic names

This is an area that is going to be much more dependent on your particular disposition to doing work, but at a certain point you will run up against the need to typeset documents

Figure 3.3: A more orderly directory.



and keep conceptually distinct areas actually digitally distinct, so if you don't buy into my dogma that is fine, but adopt some other regular process of your own and stick to it as you do your work.

4 Installation

In this chapter I will guide you through a tedious and error prone step in the process of adopting plaintext software: installing all the bits and pieces that we are going to be using. When you use something like Word to write, all you need is Word, but the workflow we are going to be using relies upon different pieces of software that all work together. The good thing about this process is that you only have to do it once, and you won't have to fiddle around with it endlessly. **NOTE**: For Windows users the process is slightly different for different software. See Chapter 8 at the end of this book for some supplemental instructions.

4.0.1 LaTeX

LaTeX, typically pronounced *lay*-tek or *lah*-tex, is a program for typesetting documents developed in the late 70s. It allows you to create really pretty documents and in particular, documents that include mathematical notation, figures, and tables. It can be used to typeset pretty much anything that needs typesetting.

LaTeX is a pretty big program (around 5GBs) so you will need sufficient space on your hard drive to accommodate it. Depending on your system you will need to navigate to the install page and pick the version that is appropriate for your OS. Follow the installation instructions as you would any other program. This takes a little time due to its size.

4.0.2 pandoc

`pandoc` bills itself as a "universal document converter" and can be used to convert files into different formats. It was primarily developed to take documents written in `Markdown` and convert them into other, prettier formats. When combined with LaTeX, `pandoc` is a pretty powerful tool and we can use it to write in `Markdown` and then simultaneously convert that document into Word, PDF, Tex and HTML. It is this software that allows you to integrate your work with your colleagues that might not want to make the shift to a plain text workflow. Further, you can actually take in documents in a common format like Word and convert them to plain text.

Just like for LaTeX above, navigate to the install page, and download the installer appropriate to your OS. `pandoc` is significantly smaller than LaTeX so you should not have an issue installing it.

4.0.3 `Git` and Github

Git is a version control system which monitors a directory for changes. Think of it as the "Track Changes" option for a Word Document, but it tracks all the changes that occur to a group of files in a directory. Git is a program with a command line user interface so you will need to know your way around a terminal to use it. It also allows you to take advantage of open source software freely available on Github.

GitHub is a website that hosts repositories of software, and if you sign up for a free account, acts like a cloud backup for your projects when they are in plaintext. All the software hosted on GitHub can be installed using `git`, and we will use it to install the templates we are going to be using for our documents. Install `Git` here.

4.1 Cross-Referencing and Text Editors

You will likely be mounting an argument or writing in an area of your discipline that requires you reference past works. So you will need a bibliography, and you will probably want to include tables, figures and diagrams that you will reference in the body of your text. It is a pain having to format a bibliography yourself or to number each table and figure. What more, it is particularly annoying when you decide to rearrange your paper and then have to rearrange the number of all the tables and figures in the text. You wll also want a nice interface to actually write and code in.

4.1.1 pandoc-xnos

`pandoc-xnos` is for cross-referencing figures, tables, equations, sections and pretty much anything else that can be written in a Markdown file. This helps for automatically handling the labelling of different parts of your document, so you can move the position of a figure or table or equations without having to change the reference to it in the body of your text. Install with:

```
pip install pandoc-fignos pandoc-eqnos pandoc-tablenos \
            pandoc-secnos --user
```

If you get an error that `pip` can't be found, make sure you have python installed on your computer, which should be the case for macOS and Linux users. You then use this suite of helper functions in documents with some conventions that we will cover in Chapter 5.

4.1.2 Visual Studio Code

One of the advantages of plain text is that you can open your files on any computer with the native programs installed on it from the factory. Every computer will have a basic text editor (macOS has TextEdit and Windows has Notepad) and you could do everything we are going to do in this working group with the command line and one of these text editors (in fact you could do everything just with the command line). I *highly* recommend you install a more advanced text editor to do your work in.

There are a few excellent options to choose from including Sublime Text, Emacs, Atom, Notepad++ and RStudio. By far the best, and the one I use is Microsoft's Visual Studio Code (or VS Code for short). The advantage to using a text editor regardless of which specific one you choose is that they highlight the syntax of whatever programming language you are writing in (including Markdown and LaTeX) enhancing the readability of your code to help writing and debugging. What more, you can typically run everything directly in the text editor so you never need to navigate away from a single window to get all your work done. You can run `R`, `python`, and keep your LaTeXdocument open all at the same time toggling between the different tasks as you need.

You should install VS Code as it is what I will be using to demonstrate how to use these plain text tools. You can install it from here. One of the nice things about VS Code is that it is *extensible* and has all sorts of extra features designed and implemented by users to help you get things done. One of the most important extensions is a LaTeX helper that we will be using when we do have to edit or write in LaTeX. After you install VS Code go ahead and install the extension here. There are also other helpful things for writing like a spell checker, a tool for auto-completing citations, a word counter, and support for your favorite (or not so favorite) statistical software. And don't worry, installing these extensions is just a matter of clicking a button.

4.2 Adding to PATH

A lot of the work of typesetting is going to be taking place in the terminal and sometimes you will get an annoying an inexplicable error like "this software is not on the PATH" or "I can't find this software", which means that your computer is searching the location

where executable programs are located (the PATH variable) but it is not finding whatever software you are telling it to run. Why this happens is a mystery. Sometimes the software has installed onto your computer, but the terminal program doesn't know where to look for where the programs are installed. I know this is a little confusing but here is a quick lesson. When you invoke a command on the terminal the first thing the computer does is check an index of the available software, which it does by examining what is called the PATH variable. If it finds whatever you are asking it to use like `pandoc` or LaTeX or R or python or whatever, it runs the command. If it doesn't find what you tell it to look for it returns an error.

Before giving up completely try updating the PATH variable so the terminal knows where the software is. Depending on what specific command line or terminal you are using the instructions are slightly different but they will be pretty portable across platforms, and operating systems. This is one of those things that you really only have to learn once and then you'll be able to improvise a lot better later. To find out where the files for a program are on macOS, run the `which` command followed by what your looking for, such as `which pandoc`, and it will print the path to where the executables of these files are installed. Sometimes the data files for a program get installed onto our computer without the PATH variable getting updated to tell the terminal that the software is installed.

There are two different kinds of terminals on macOS. Run this line of code in your terminal and it will tell you which terminal you are using: `echo $0`. Check below for relevant information about how to add to the path:

4.2.1 `zsh`

To add a directory to your path in zsh: 1. Run `path+=('/path/to/dir')` in an open terminal where the path is to where the executables for the program are. For me `pandoc` is located in `/usr/local/bin/pandoc` and LaTeX is in `/Library/TeX/texbin/latex`. 2. Then run `export PATH` in the same terminal. 3. Restart the terminal and check that it works by typing in `latex --version` or `pandoc --version` or whatever else was giving you trouble.

4.2.2 `bash`

Remember when we learned how to use `vim` from the command line and what a hidden file was? Well now that knowledge will actually come in handy.[1] To add something to the PATH in `bash` you need to edit your `.bash_profile` which is located in your home directory.

[1] If you are unfamiliar with `vim` and hidden files, refer to the session 1 documentation, "Project Organization and the Terminal".

The home directory is where your terminal opens up and is located at `/Users/your-user-name`.

1. Open the `.bash_profile` file in your home directory (for example, `/Users/your-user-name/.bash_profile`) in `vim` by running `vim .bas_profile` in your home directory in a terminal.
2. Add `export PATH="your-dir:$PATH"` to the last line of the file, where *your-dir* is the directory you want to add.
3. Save the `.bash_profile` file by exiting `vim` with `:wq`.
4. Restart your terminal.

4.3 Summary

Congratulations! You have actually just done a lot of the work of getting started using plain text for your research and writing. This is an onerous task and you should be commended for your effort. If you have managed to get this far then you are likely somewhat invested in adopting the plain text paradigm but if this effort has inspired skepticism or reticence about diving deeper I think it is important to note that installing the software is something that you only have to do once on any given computer. So you won't have to endlessly tinker with things. Once these tools are installed they are very dependable

5 Using LaTeX, Markdown, pandoc

Now that you are familiar with the Terminal and have installed the software we are going to be using you are ready to begin addressing the core competencies of working with plain text software for your research and writing. The essential workflow that we are adopting here is writing in Markdown, a simple markup langauge with a few extra bells and whistles, and then using the pandoc program to convert these into HTML, PDF, ad Word files. To establish why this is the appropriate workflow to adopt I will outline what markup langauges are and how they work, as well as the conventions used in LaTeX and then Markdown.

5.1 LaTeX

Though you won't need to write with LaTeX you will need to know what a LaTeX document looks like and how to work with one. Once you have a feel for the LaTeX document setup, you will be able to create your own document templates for pretty much any use. For example, I have LaTeX templates for memos, letters with my contact and university affiliation, article drafts, class handouts, single columned general prose, double column general prose, and even the documentation you are looking at right now was created using this workflow.

LaTeX at bottom is something called a *markup language*, or a means of encoding text where different characters are used to designate structure, formatting and style. In a Word document, all that appears on your screen is the text you are composing, and its style and structure are handled by the program. If you want to change the type face, or the margin, or the line spacing, you go to the top bar and click through menus to change the style. There is a lot of behind the scenes action going in a Word file as the means by which document is styled is hidden from you. In a markup language, all that information is present in the document itself as you explicitly declare the style and structure using the conventions of that particular language. To give a specific example, where in Word you would click the italics button to make text *italics*, in a markup language like LaTeX you wrap the text you want to be italicized with the command `\textit{}`. There is a lot more that goes into it, and diving into practical examples will help to give you an idea of how markup languages

work and LaTeX in particular.

LaTeX is notorious for being unintuitive and hard to pickup and is the bane of any graduate students existence when they take STATS102 for the first time. The thing about LaTeX is that it *is* complicated and unintuitive but there are only a few complicated and unintuitive conventions to understand before LaTeX seems a lot clearer. Below is an example of the simplest LaTeX document you can make, and examining it helps us to begin to understand how markup langauges are different from something like Word and what LaTeX in particular looks like:

```
\documentclass{article}

\begin{document}

  Hello World!

\end{document}
```

In LaTeX, anything that begins with a backlash "\" is a command, and commands usually have brackets at the end to take in some arguments. You can think of working with a LaTeX document like working in narrower and narrower environments. In any file, you first need to declare the kind of document you are creating with `\documentclass` and then declare the beginning of the document with `\begin{document}`. The `\documentclass` command creates the underlying environment with some associated styles in which you will be composing prose, before the `\begin{document}` command creates the environment in which the prose and content of your document go.

The convention of open and closing environments applies to most features of a LaTeX document. We open a document with `\begin{docuemnt}` declare the end of the document with `\end{document}` while figures or images in your document need to be between the commands `\begin{figure}` and `\end{figure}` and the same is true of tables, which go between`\begin{table}` and `\end{table}`. The body of your document can be 2 words or 200 pages worth of content, and I simply use the "Hello World!" document as an example. You could make that document your self in a matter of seconds.

The `\documentclass` command specifies how the document will be formatted and loads a set of default settings. There are a variety of options to choose from depending on your need including `article`, `book`, `report`, `slides`, and `letter`. A very common document class that you'll see is `memoir` which is used for general prose writing. This command can take in arguments to further specify the formatting of your document and usually includes things like the paper size, whether it is one or two sided, font size, line spacing, among many other things. Here is an example where we are setting a document to have 11pt font,

to be one sided and use formatting for an article type document, using all the bells and whistles from the memoir document class:

```
\documentclass[11pt,article,oneside]{memoir}
```

Another convention in LaTeX is to declare metadata and other information in the *preamble* of the document. The preamble is the part of the file that is above the \begin{document} command. In the above example the preamble includes only the \documentclass command but you can do a lot more here including telling LaTeX about all the formatting of your document as well as loading any packages that you will be using to create your document. Also, you can declare metadata in the preamble that then will appear in your document such as the author name, title of the document, abstract, date, and pretty much anything else. Here is another simple example:

```
\documentclass{article}
\title{My First \LaTeX Document}
\date{1/1/1597}
\author{William Shakespeare}

\begin{document}

\maketitle

Hello World!

\end{document}
```

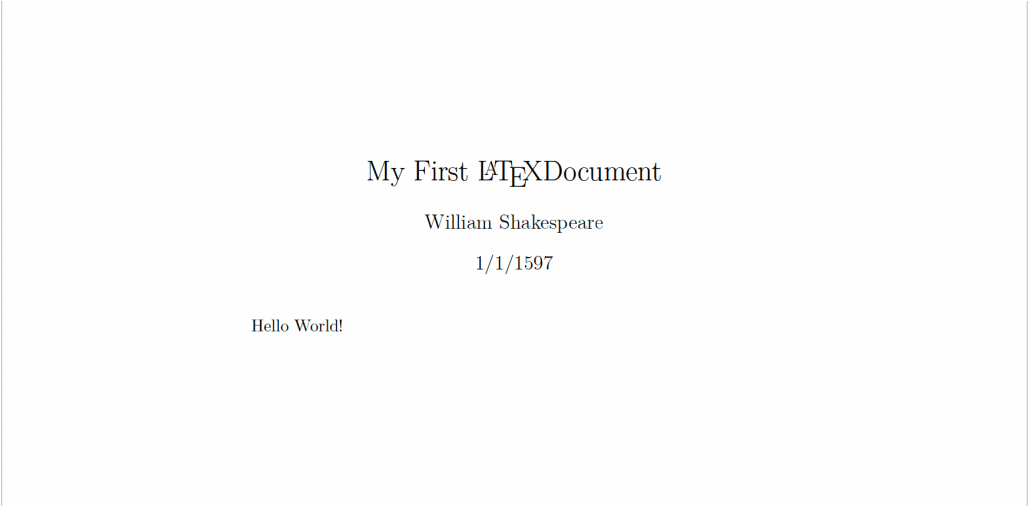
The \title, \date and \author commands are components of the \maketitle command that you can invoke after you begin the document to create a well formatted title block for your document. The above code renders a document that looks like Figure 5.1.

As mentioned above, the body of the document (here it is just "Hello World!") can be as long as you want and encompass all sorts of different things that are particularly helpful to social scientists who write documents that use data to make an argument. For example, and we will cover all of these shortly, you can include tables, figures, citations, theorems, proofs, equations, diagrams, code blocks, and any number of other information that you want typeset in a beautiful way.

To summarize here are the important parts for understanding a LaTeX document and markup languages in general. Remember to:

1. Declare the beginnings and ends of environments in which different formatting or content is located.
2. Include metadata and package calls in the *preamble* of the document.

Figure 5.1: A document made with LaTeX



3. Render formatting using commands like `\textit{}` or `\textbf{}`

Learning the exact conventions of LaTeX is a task that is worth exploring. If you want to be a truly autonomous plain text writer and researcher, you will have to learn more about LaTeX so you can create your own templates. *But*, if you want to stick to thinking and writing then you'll need to know more about the accessible alternative to LaTeX: Markdown.

5.2 Markdown

Markdown is a much easier to learn and intuitive markup language than LaTeX and is becoming more and more ubiquitous. In Markdown, italicized text is created by wrapping the text in **asterisks**, while bold is done with ****two asterisks****. There are certain limitations to Markdown; for instance there is no underlining in Markdown so you have to get by with the other ways of styling text. Here is an example of some text in Markdown before being typeset:

```
# Header 1

You might have some introductory comments

## Sub-Header

Some text with different styling such as *italics* and **bold**.
Remember there is no underlining which is old fashioned anyways, but
you can insert [Hyperlinks](https://en.wikipedia.org/wiki/Hyperlink)
and create tables:

| Movie          | Actor         |
|----------------|---------------|
| Apocalypse Now | Marlon Brando |
| Wizard of Oz   | Judy Garland  |
| Sound of Music | Julie Andrews |

There are also other important aspects of academic writing.^[Like footnotes
     that are rendered at the footer of the page in which they appear.]

![Here is a figure inserted into a Markdown file.](figures/
    healy_plaintext_vis.png)
```

The good thing about Markdown is that it is much easier to use than LaTeX so instead of messing around with a document preamble, document classes, and style files you can get straight to writing your document. I am not going to cover all the different Markdown conventions in this documentation but you can find them here. Markdown is pretty straight forward but you will still want to take advantage of all the fancy features of LaTeX while maintaining the usability and ease of Markdown. Luckily, someone already thought up this idea and created the program pandoc.

5.3 pandoc

pandoc is a command line based program that allows you to convert files between different file formats. It is a pretty powerful program because it will take the markup conventions from one format and translate them into another. So, when you write your next article in Markdown and really emphasize some part ("And to reiterate *again*, it is the *Lumpenproletariat* that ultimately must be motivated by a vanguard party to seize the means of production from the Petty *bourgeoisie*.") pandoc will convert the text surrounded by *asterisks* (which indicates italics) and converts them to an \textit{} command. pandoc is the key link in the workflow that allows us to convert our simple Markdown files into LaTeX files and then into PDFs and how we are able to accommodate the different resources we

need for academic writing, like bibliographies, footnotes, tables, figures, and diagrams.

The basic usage of `pandoc` looks like the following:

```
pandoc my_doc.md -o my_doc.pdf
```

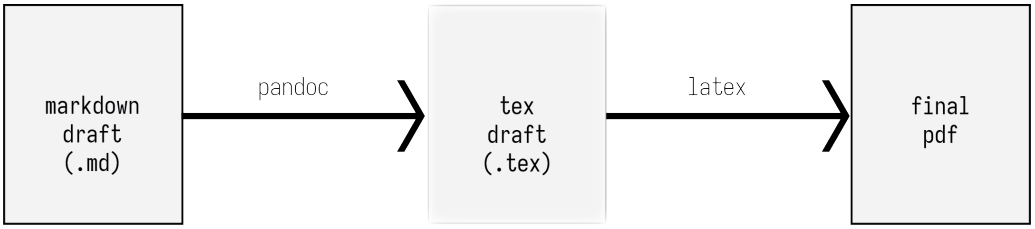
On the left, directly after `pandoc` is the Markdown file that we want to convert to a PDF file. The `-o` flag indicates that the `my_doc.pdf` file is in fact the intended *output* document from the input. That is pretty simple, but so too will be the output PDF. Compare the line above to this one:

```
pandoc my_doc.md -o my_doc.pdf --from=Markdown --template=/Users/
    timothyelder/.pandoc/templates/eisvogel.latex --listings --filter
    pandoc-xnos
```

That is the command for rendering the document your are looking at right now. It does the same thing that we saw above, specifying an input file and then an output file, but it includes all sorts of extra flags that indicate how to handle different attributes of your document and the proper formatting for the output. The `--from=` flag makes it perfectly explicit to the software what the input format is (in this case it is Markdown), while the `--template=` flag tells the software where to look and which template to use when rendering the output. The final two flags (`--listings` and `--filter pandoc-xnos`) indicate the style for code blocks (those shaded areas that display computer code throughout this document) and how to cross reference different parts of the document respectively.

It is important to get clear on what is actually going on when we use the `pandoc` command to create a PDF. What we are doing can be seen in Figure 5.2

Figure 5.2: Writing in Markdown and typesetting into a pdf with a tex intermediate step.

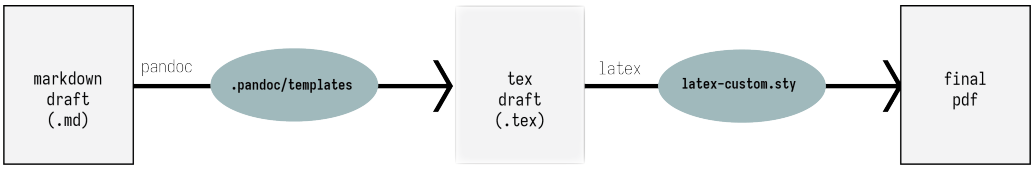


We write our files using the conventions of the Markdown format (indicated by the `.md` file extension) and then use `pandoc` to typeset these into a PDF. `pandoc` is not a typesetting software itself *per se* but relies upon LaTeX to typeset into PDF. When you invoke `pandoc` to create a PDF, `pandoc` performs a quiet intermediate step where it creates a LaTeX copy of the Markdown document (the `.tex` file in Figure 5.2) and typesets that into a PDF by invoking the LaTeX program. `pandoc` is super smart, and understands how to translate the

conventions of one markup language into another, and that is how we are able to write in Markdown (with all its simplicity and speed) while still taking advantage of all the power of LaTeX.

We are able to add a few bells and whistles to pandoc to define the specific formatting of the document. We do this partly by manipulating the characters and conventions in our document directly, but also outside our document, by invoking templates and style files that you can install. This can be seen in Figure 5.3.

Figure 5.3: There are a few extra little dependencies.



5.4 Document Composition with Markdown

In this section, I want to explain how you put together the various pieces that we have up to this point only been reading about. We are going to be covering:

1. Inserting figures
2. Creating tables.
3. Including math, both numbered equations and inline math

5.4.1 Writing the Document

This part is pretty self explanatory. You type characters, which form words, words create phrases and sentences, and you keep doing that until you have something that looks like a defensible argument. Once you have that, you put in all the nifty evidence that you have discovered to support your argument and there are all sorts of things that go into that.

5.4.2 Headers and Metadata

Where in a LaTeX document you put metadata such as the author name, date, title, abstract, etc. in the preamble of the document, in Markdown you use what is called a YAML (*yam-mul*) header. Since you don't want the metadata being represented in the document

as it is, you need to contain it in special characters (sort of like how you create an environment in LaTeX with \begin{} and \end{}). Three dashes opens and closes your YAML metadata header like this:

```
---
title: My First Markdown Paper
author: Timothy Elder
date: February 17th, 2023
---
```

The information that the YAML header contains is arbitrary though there are a few standard entries like "author", "title", "date", and "abstract". Because pandoc translates your Markdown document into LaTeX before typesetting, you have the full arsenal of LaTeX options available to you, which means you can change the font, margins, headers and footers among many many others. Too many to cover here but you can review the full set of options in the pandoc documentation here (see the section "Variables for LaTeX").

5.4.3 Figures

A figure or image in a Markdown file can be included with the following notation:

```
![Here is where you put your caption.](/absolute/path/to/image.png){@fig:
    example-crossref-label}
```

The image will render on the page around where you insert the line but you will not have exact control over it unless you go edit the intermediate .tex file. In our templates we specify some options that help images render in an attractive way and LaTeX uses an algorithm for image placements when not manually placed. Instead of manually numbering figures and referencing them in the body of your paper, you assign them a label, which follows the call to the image, wrapped in curly brackets with the @fig: prefix. Rather than having to keep track of how your figures are numbered, when you want to reference the figure in the body of your paper, you do so like this:

```
Here I am explaining the really impressive figure that I made that is going
     to definitively demonstrate some strong association between two
    variables. This can be seen in Figure @fig:example-crossref-label.
```

This means of cross-referencing figures is really convenient especially when you decide to reorder your paper, and suddenly figure #5 is where figure #1 was and all the labels would have to be manually changed if you were using something like Word. **NOTE**: Whenever using pandoc-xnos, include it with `--filter pandoc-xnos` as a flag for the pandoc command in.

5.4.4 Tables

Tables are pretty easy in Markdown and can be styled different ways. This is a pretty bare bones table:

Table 5.1: Some information about Star Wars characters.

Character	Height (cm)	Mass (kg)	Homeworld	Species
Han Solo	180	80	Corellia	Human
Wedge Antilles	170	77	Corellia	Human
Yoda	66	17	NA	Yoda species
Ackbar	180	83	Mon Cala	Mon Calamari
Mace Windu	188	84	Haruun Kal	Human

This table is created with the following code:

```
-------------------------------------------------------------------------
   Character      Height (cm)   Mass (kg)   Homeworld       Species
---------------- ------------- ----------- ------------ ----------------
    Han Solo          180          80        Corellia        Human

 Wedge Antilles       170          77        Corellia        Human

      Yoda            66           17           NA       Yoda species

     Ackbar           180          83        Mon Cala     Mon Calamari

   Mace Windu         188          84       Haruun Kal       Human
-------------------------------------------------------------------------

Table: Some information about Star Wars characters. {#tbl:id}
```

You will be able to output from most statistical software into the Markdown format or LaTeX. And tables in Markdown can also be labeled, just like figures, so you can reference them without keeping track of what order they appear in.

5.4.5 Math

If you are a quantitative scholar and need to describe your modeling strategy you can do that in Markdown. This requires a bit of knowledge of LaTeX: as mentioned above, pandoc

converts your Markdown to LaTeX before typesetting, you can use LaTeX conventions when we really need to. This is such an instance. So if you want to explain to someone the standard deviation of a population you would write the formula using LaTeX math commands (\sigma, \sum, \frac, \bar, exponents and subscripts, etc.) on its own line and wrap it in double dollar signs ($$) like this:

```
$$ \sigma = \sqrt{\frac{ \sum (x_i - \bar{x})^2 } {N - 1}} $$
```

which renders your math beautifully:

$$\sigma = \sqrt{\frac{\sum (x_i - \bar{x})^2}{N - 1}}$$

You can also insert math inline with your text $\mu = \frac{\sum_{i=1}^{n} x_i}{n}$ using single dollar signs in line with the rest of your text, like:

```
When $a \ne 0$, there are two solutions to $(ax^2 + bx + c = 0)$ and they
    are $x = {-b \pm \sqrt{b^2-4ac} \over 2a}$.
```

5.5 Make

It can be very annoying to continually run the same lines of code over and over again, when you are nearing the completion of working on some document, particularly if you are obsessive and neurotic and messing with some minor part of a paper that you want to get perfect. A very helpful resource that computer scientists thought up was to create a system for handling running code called Make. Make automates the process of, for lack of a better word, *making* things. This is somewhat advanced when it comes to typesetting documents so if you want to skip this part or skim that is totally fine. I will note that this tool will be particularly helpful if you are a quantitative scholar and have different scripts that handle the creation of data, figures, or tables etc. etc.

Make is particularly powerful because it has a system for checking whether or not a given output file needs to be changed given some change in the input file. For example, say you have a recipe (recipes in Make are the different commands that you want it to run) that creates your document that looks like this:

```
## Location of Pandoc support files.
PREFIX = /Users/timothyelder/.pandoc

## Location of your working bibliography file
BIB = /Users/timothyelder/Documents/bibs/socbib-pandoc.bib

## CSL stylesheet (located in the csl folder of the PREFIX directory).
CSL = asa

dissertation.pdf: dissertation.md
    pandoc -f --pdf-engine=xelatex --template=$(PREFIX)/templates/ucetd.
        template --top-level-division=chapter --filter pandoc-xnos --
        citeproc --bibliography=$(BIB) dissertation.md -o  dissertation.pdf
```

To typeset the `dissertation.pdf` file given the recipe in the Makefile, you would simply type into the terminal with the Makefile, `make dissertation.pdf` and the recipe runs. Make will check to see if any changes have been made in the name of the prerequisite file (the `dissertation.md` file) which is used to make the PDF. If no change has been made to the Markdown file, then Make will let you know and say `make: Nothing to be done for 'dissertation.pdf'`.

When you run "pdf", a couple of python scripts get run in a particular order, then an R script, and then a `pandoc` line runs and then a shell command. That would be a lot of keystrokes on your own but make does it auto-magically. This is actually a recipe that I have because I have a project where we generate all the data from optical character recognition of JPEGs, and then carving up big plaintext files with regular expressions. Sometimes the underlying data changes because we notice there is some error in the plaintext file and we have to regenerate a whole bunch of CSVs from the plaintext files. What make does is it checks the various pieces of data that you are asking it to use and that you want it to make, and checks whether or not they have changed at all since you last ran Make, and it only runs precesses to update pieces of data as needed. So if you keep running Make without changing anything it will tell you "nothing to update".

5.6 An Example Project

We are going to be using an example project that you should have already installed onto your machine when you installed all of the software but in case you forgot run the following line of code to get it:

```
git clone https://github.com/TimothyElder/pwg-template
```

In the `pwg-template` directory there should be a sub-directory called `article-markdown`.

Open that up and take a look at the file called `article-template.md`. It is an example article about the War of the Roses with some examples of the most commonly used conventions in Markdown. The `article-markdown` directory should have several files in it:

```
Makefile
article-template.md
/figures
war-of-roses.bib
```

Go ahead and take a look at the file called `article-template.md`. It should contain a header that looks like this:

```
---
title: The Implications of the War of the Roses on Anglo-Franco Relations
author:
- name: Timothy Elder
  affiliation: University of Chicago
  email: timothyelder@uchicago.edu
- name: Joe Bloggs
  affiliation: Northwestern University
  email: joebloggs@northwestern.edu
date: January 2023
thanks: "Thank you to the everyone that supported this work."
abstract: The War of the Roses was a significant event in English history
    that had far-reaching implications for the relationships between
    England and France. This paper explores the impact of the conflict on
    the political, economic, and cultural ties between the two countries.
    The war resulted in significant changes to the English monarchy, which
    had repercussions for the English crown's relationships with other
    European powers, including France. This paper provides an in-depth
    analysis of these impacts, and sheds light on the lasting effects of
    this important event in English history.
bibliography: [war-of-roses.bib]
---
```

This is a YAML (Yet Another Markup Language) header and it includes metadata about the paper we are typesetting. In this case it is about the War of the Roses and it lists me as the author. It also includes an acknowledgements, and specifies where the bibliography is located. In this case the bibliography is stored in the same directory as the document so the relative path is just the file's name.[1] You could have something much more bare bones than this if you were just writing a draft of a paper such as this:

[1] You can store your bibliography anywhere on your computer and reference it in your document's metadata. For instance, for most of my work I keep a single `.bib` file (stored at `/Users/timothyelder/Documents/socbib-pandoc.bib`) which has the citation information for everything I am likely to cite and I always reference that `.bib` file in my documents

```
---
title: Origins of the War of the Roses --- Draft
author: Tim Elder
---
```

The paper includes figures, tables, and different styling of the prose. You can use this as a template for other documents so feel free to go ahead and do whatever you want to this file.

When we are ready to typeset our draft document, we can invoke the `pandoc` program via Make in the terminal by simply typing in `make pdf` from the directory with the `article-markdown.md` file in it. You can inspect the Makefile to see how it works and you'll quickly notice there are a few extra bells and whistles to the `pandoc` program.

6 Dynamic Documents with RMarkdown

If the last chapter was the heart of the PlainText Working Group, this is the capillaries flowing through that heart: making sure that your arguments are rendered beautifully in a document that can readily prove to others that you conscientiously executed good social scientific research. We are going to approach two very exciting topics: the first is creating dynamic documents with RMarkdown. The second is creating beautiful and reproducible figures. This processes go hand in hand and we are going to be using the latter as a use case for the former. That is to say we are going to use figure creation to learn why and how to use a dynamic document format like RMarkdown.

To learn the principles of good figure design and how to actually make figures in R with `ggpplot`. You can follow along with the source code that generates a nearly identical document as this in the `plaintext` repo we downloaded earlier, in the `example-rMarkdown` directory. If you follow along, you are going to be interacting with the document and running some code. After running through the document we are going to be rendering to PDF but along the way we are working in what is called the source file, the `.Rmd` file. Before moving on to how to do these things I want to get clear on what precisely we are talking about, and then why you should use these tools. I have spared few opportunities to evangelize for the plain text way of doing things but I do think the logic of using something like RMarkdown does require some further explanation.

6.1 What is RMarkdown

I called RMarkdown a "dynamic document format" which might have been confusing on first pass considering documents aren't static things as we are composing them. That is certainly true, but the information contained in them is meant to remain the same once we are done working on them. The dynamic part comes in the extra bells and whistles that are in the document. RMarkdown and other similar formats (Sweave for LaTeX, jupyter for Python) allow you to embed code along with the prose of your document which means that you are able to keep the means by which you produce your model outputs, tables, figures and diagrams right along with the actual argument. It isn't just that the code is

written in your document (in fact you often don't want that) but that the code is included to load data, manipulate it, create figures and models from it and include those figures and model results directly in your document. That means that when the underlying data changes, you don't have to rerun your code to update the figures and copy and paste it into your document. It auto-magically updates.

The most important thing a document format like this is used for is creating reproducible findings and ensuring that the process you used to get from your data to your insights is reproducible for your friends, colleagues and critics that might have some questions about why your scatter plot looks the way it does, or why a particularly convincing coefficient is both significant and has a large effect size. This has some definitive advantages from the static way of generating documents where you work in some statistical software on the command line or (heaven forbid) a GUI, export figures to an image format like JPEG or PNG and copy and paste your model results into your document. This is bad because copy pasting model findings separates the code that generated the findings from the actual findings, if you get sloppy with your code, it might be hard to figure out how a particularly compelling model result was generated. The same is true for figures.

The dynamic document lets you create the figure or run the model and create the table of results directly inside the document. That way when, after submitting the document to a journal or your PI or advisor, and they come back and ask why a scatter plot looks the way it does, or how a particular coefficient was generated, you can look at the exact model or figure creation procedures used to make that part of the document. That is important for scientific scrutiny but also helpful for you to save time. If you are working on the code at the same time you are writing your manuscript and want to keep figures up to date with the text, you don't have to keep copy pasting the right figures or models to the proper location. They will automatically update when you change the code.

Transparency is important: fraud in science is a problem[1], and we are all upstanding scholars and won't attempt to commit fraud, but there may come a time when someone levels an unwarranted broadside against you regarding a paper or finding you published; this has become something of a trope in the field of psychology today with "methodological terrorists" and "scientific vigilantes", but overall the trend of ensuring that work is conducted in an ethical way is good. Being able to distribute code that actually generated your findings right along with the prose of your document is a helpful way of encouraging transparency. It will actually make you think about your work differently knowing that someone could download the source files of your project and examine the methodological decisions you made.

[1] There have been a few recent and unfortunate examples that are worth reading about including: Singal (2015), Bhattacharjee (2013), Scheiber (2023).

There also may come a time when you feel the need to return to work you did in the past. Maybe you are finally putting together a book project that is going to make a grand theoretical and methodological contribution to the field and it will require you to marshall all your past findings. It will be a big step to have a document that explicitly tells you what you did in the course of publishing your material, and you won't have to spend endless time theorizing about your past intentions examining random source files in an old directory you pulled off a hard drive from an old computer. I have had a fair number of experiences of learning a new trick for making a figure that I then used in a later project. Having the code with the figure makes it easy to identify what code I want to emulate.

6.2 RMarkdown Features

6.2.1 Code Chunks

The code that you include in your documents goes into what are referred to as "code chunks", delimited areas in your document that indicate to your computer that there is code to be run through an interpreter rather than just rendering the alphanumeric characters as prose for your readers to read. The special characters that are used to indicate that you are dealing with a code chunk are three backticks (‘‘‘) followed by curly brackets which contain an `r` and then three more back ticks to end the code chunk. It looks like the following code chunk just not with the hastags at the beginning of the line:

The `r` tells the computer what language you are writing code in. RMarkdown supports both R and python and I am told stata but it should assume that you are running R. Each code chunk has a few basic options you need to decide about including:

- `include = FALSE` prevents code and results from appearing in the finished file. R Markdown still runs the code in the chunk, and the results can be used by other chunks.
- `echo = FALSE` prevents code, but not the results from appearing in the finished file. This is a useful way to embed figures.
- `message = FALSE` prevents messages that are generated by code from appearing in the finished file.
- `warning = FALSE` prevents warnings that are generated by code from appearing in the finished.
- `fig.cap = "..."` adds a caption to graphical results.

As a general rule, particularly when you are dealing with figures you want to set `echo` to false so that your figure appears but that the code that generates it, does not. Fortunately,

Element	Markdown Syntax

Heading	`# H1`
	`## H2`
	`### H3`

Bold	`**bold text**`
Italic	`*italicized text*`
Blockquote	`>blockquote`

Ordered List	`1. First item`
	`2. Second item`
	`3. Third item`

Unordered List	`- First item`
	`- Second item`
	`- Third item`

Code	`` `code` ``
Horizontal Rule	`---`
Link	`[title](https://www.example.com)`
Image	`![alt text](image.jpg)`

we can set the defaults for all our code chunks in a single document within a code chunk at the top of the document (this is separate from your YAML header).

There are a few things to note about this code chunk. For one it is *named*: directly after the language is declared with `r` there is the word `Setup` which is the name of the code chunk. Then there are the actual options for this "Setup" code chunk and then the R argument `opts_chunk$set` which actually sets the default chunk options for the rest of the code chunks, so you don't have to include those arguments later.

6.2.2 Prose

Everything else in your document is what I think is reasonably called "Prose", that is just like you would in a regular Markdown file, you use all the alphanumeric characters to create natural language and an argument that you would in any plain text file. There the Markdown conventions you need to be aware of for representing style and including citations. Here is a table that shows some of the basic Markdown syntax for review.

6.2.3 Citations

Another key part of academic writing is including citations to other works that inform your research. To do this you will need to use a `.bib` file which is generated using a bibliographic software like Zotero or Mendeley. I highly recommend using a `.bib` file because it saves a lot of time formatting and creating the proper list of cited references.

To include citations in your `.Rmd` file you will need to specify where your `.bib` file is located by specifying its path in the YAML header at the top of the document. We have an example `.bib` file included in the directory that contains this source file called `plaintext.bib` with just a few sources in it. You then reference an item in that `.bib` file by writing an @ symbol followed by the "citation key" of the item you want to cite, like this `@durkheim_division_2008` which renders a citation like this Durkheim ([1893] 2008). All the citations that you use in your document will appear at the end of your paper, and including a `# References` section at the bottom of your RMarkdown or Markdown document will add a section heading before the bibliography is printed.

There are a variety of ways in which you can customize your bibliography so it matches the proper format of whatever journal you might be submitting to. That will require that you use something called a `CSL` file, which can also be included in the YAML header at the top of your document. There are a lot of CSL files people have created for the various citation conventions, and there is probably one available for the journal you'll be submitting to. You can find some here.

6.2.4 Loading Data

To demonstrate the principles of good figure creation and the actual process by which you do it in `R` using `ggplot` we are going to be looking at some data from the Chicago Transit Authority about ridership on Chicago's "L" Trains.

Much of this document, including this sentence, was written on the Brown Line between Harold Washington Library and Damen Ave.

When working in an RMardkown document, a good practice to follow is to load your data upfront, transform it, then you can use it anywhere else in your document. We aren't going to be doing that here because I want to demonstrate what we are doing as we go, but as a general rule when you work with your documents, include the calls to the libraries you need in a single code chunk, when you load data and create functions when you need and include comments.

```
library(tidyverse)
library(ggpubr)
library(ggrepel)
library(zoo)

# Load CTA Station Data
cta_stations <- read_csv("/Users/timothyelder/Documents/plaintext_workshop/
    data/cta_stations.csv")

cta_stations$north <- as.factor(cta_stations$north)
cta_stations$year  <- as.factor(cta_stations$year)

# Load CTA Ridership Data
cta_rides <- read_csv("/Users/timothyelder/Documents/plaintext_workshop/
    data/cta_rides.csv", show_col_types = FALSE)

# Load Approval Ratings Data
approval <- read_csv("/Users/timothyelder/Documents/plaintext_workshop/data
    /approval_polls.csv", show_col_types = FALSE)

# Load and transform State of the Union Data
sou <- read_csv("/Users/timothyelder/Documents/plaintext_workshop/data/sou-
    length.csv", show_col_types = FALSE)
sou$President <- str_extract(sou$President, "(\\b\\w+\\b)+$")
sou$Date <- as.Date(sou$Date, format = "%m/%d/%Y")
```

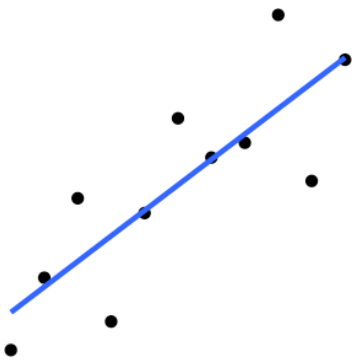


Figure 6.1: Weak Linear Relationship

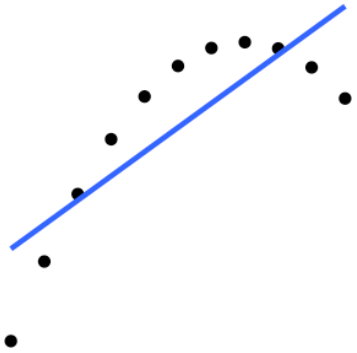


Figure 6.2: Curvilinear Relationship

6.2.5 Anscombe's Quartet

We use visualization for a few different purposes: for one, we use visualizations to quickly make a point about our data. It just so happens that human being's dominant sense is vision, and a visualization is an excellent way of mapping numeric values to visual features that can be quickly processed by your reader. We also use visualizations to learn things about our data as we are analyzing it. And we can learn things from visualizations that you can't learn from basic summary statistics.

A key thing that we use visualizations for is understanding what our data can acutally say, and where it actually *is*. Anscombe's Quartet powerfully demonstrates the need to actually visualize your data so that you know where it is.

Francis Anscombe constructed 4 datasets, each of which are composed of eleven obervations of two variables, and across each of the datasets the variables have the same mean values, standard deviations, and correlation coefficients. Despite these nearly identical summary statistics, each of the datasets are composed of very different sets of data, that are only revealed by visualizing them.

In the first there is a weak linear relationships between the two variables, while in the

second the relationship is curvilinear. In the third the relationship is very linear with a single troubling outlier that might be driving a particularly strong correlation. And in the fourth, there seems to be something a little wrong with the data.

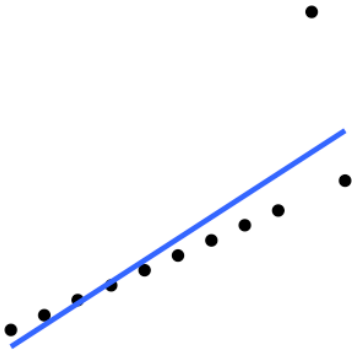


Figure 6.3: Linear relationship with outlier

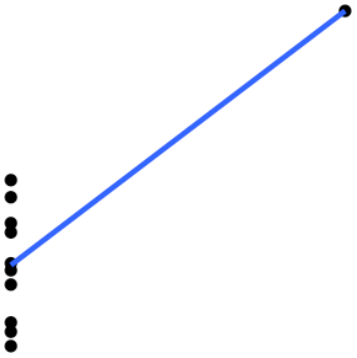


Figure 6.4: Miscoded Variable

6.3 ggplot

ggplot is a library in R that was designed on the basis of a book that was written in the 1990s called *The Grammar of Graphics* (Wilkinson 1999), thus the "gg" in ggplot. It is the * of figure creation in R and is what we are going to be using here. There are also other libraries that have adopted the ggplot conventions and applied them to other kinds of figures not covered in the original package like ggtree for phylogenetic plots, ggnet for network data, and ggridges for ridge plots.

There is a *grammar* to ggplot that once you acquire will allow you to make all sorts of compelling and visually parsimonious plots. The basic setup is thus:

1. Call the ggplot function and feed it data in the data argument, where the "data" in the following example is a data frame with all the data you want to plot.

```
ggplot(data = dataframe)
```

2. Tell ggplot what you want mapped to aesthetic properties of the plot with the aes argument. At a minimum you will likely want to give the aesthetic mapping both an "x" and a "y" value (a column in the data frame) to the respective axis of the plot. This isn't the case if you are making a histogram or density plot but we will cover that shortly. This defines the basic structure of the plot and you can then layer on additional properties.

```
ggplot(data = dataframe, aes(x = Variable_X, y = Variable_Y))
```

3. Call a geom which defines the kind of plot you want to make. For example a line plot is made with geom_line, a histogram with geom_histogram, a scatter plot with geom_point, and a bar plot with geom_bar. You "add" geoms to the base plot that you defined in step two using a literal addition symbol (+) at the end of the line where the base plot is defined.

```
ggplot(data = dataframe, aes(x = Variable_X, y = Variable_Y)) + geom_line()
```

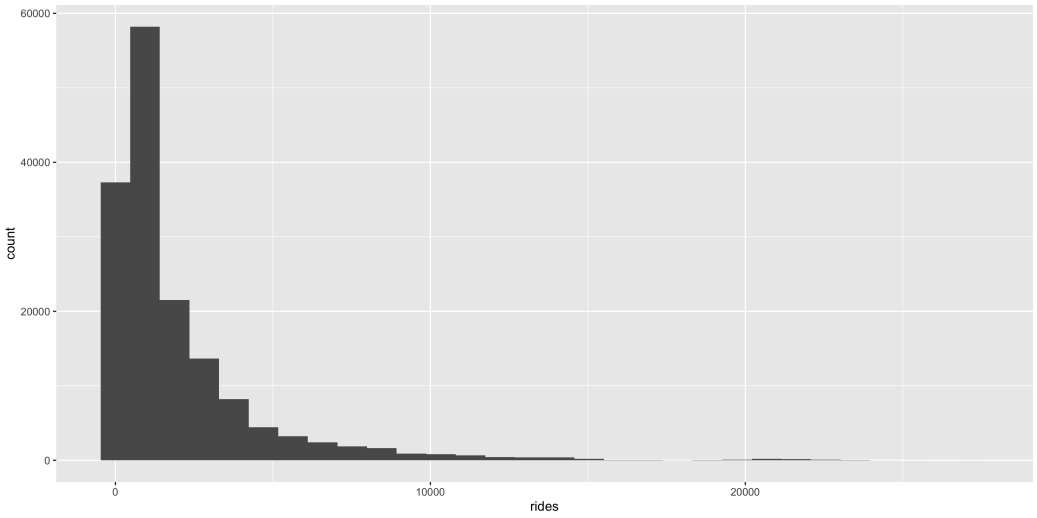
That is the fundamental structure of any plot or figure you can make with ggplot and there is a lot more complexity to add using different geoms and aesthetic mappings that we will cover shortly. And we will start that by actually creating some plots to "learn by doing".

6.3.1 Distributions

When we start to look at our data with visualizations we often want to know the distribution of values that the data can take. We usually do this with histograms, to count the number of times a given value appears. We are going to use some data from the Chicago Transit Authority to demonstrate this. The CTA dataset has information about all the "L" Stations in Chicago and the number of "daily entries", the number of times people swiped their metro cards to board the trains.

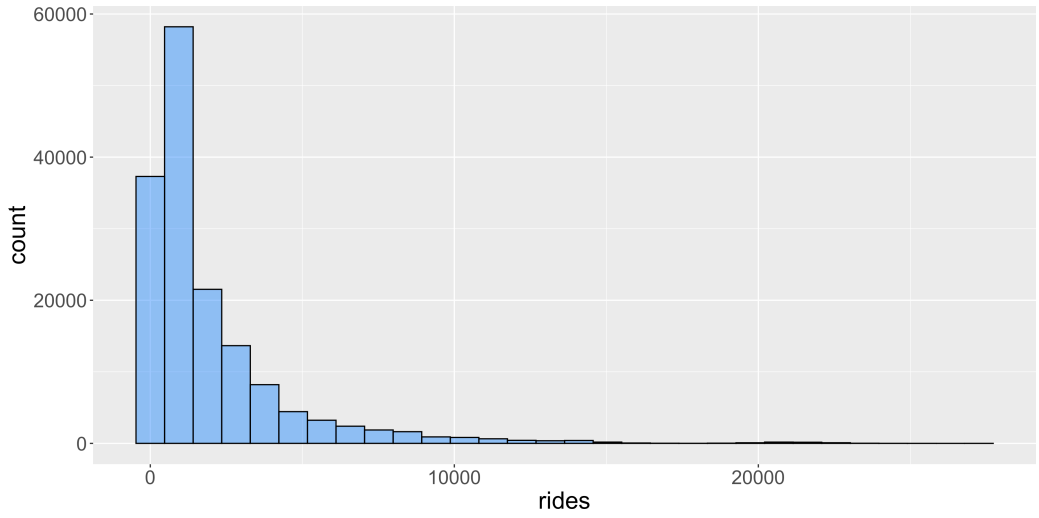
Here we will use the grammar we learned above to start building a plot, by calling `ggplot` telling it what data we want to use, and providing an aesthetic mapping for the X and Y axes and calling a `geom` to create the plot. In this case we are going to be using the `geom_histogram` function to create a histogram. Because a histogram only counts the values in a single variable at a time, when we tell `ggplot` an aesthetic mapping with the `aes` function we only give it an X axis variable.

```
ggplot(data = cta_rides, aes(x = rides)) + geom_histogram()
```



A histogram gives some immediate information about the distribution of daily rides across all the stations in the dataset: in this case it shows that daily ridership across the stations is highly positively skewed, with most stations falling on the left side of the X axis with a very long sparse right tail. But there are some issues with this plot: each bar represents a different "bin" with the height corresponding to the number of times that value appears in the data. The bins are a uniform color so it is hard to tell them apart. Also, the axes are hard to read.

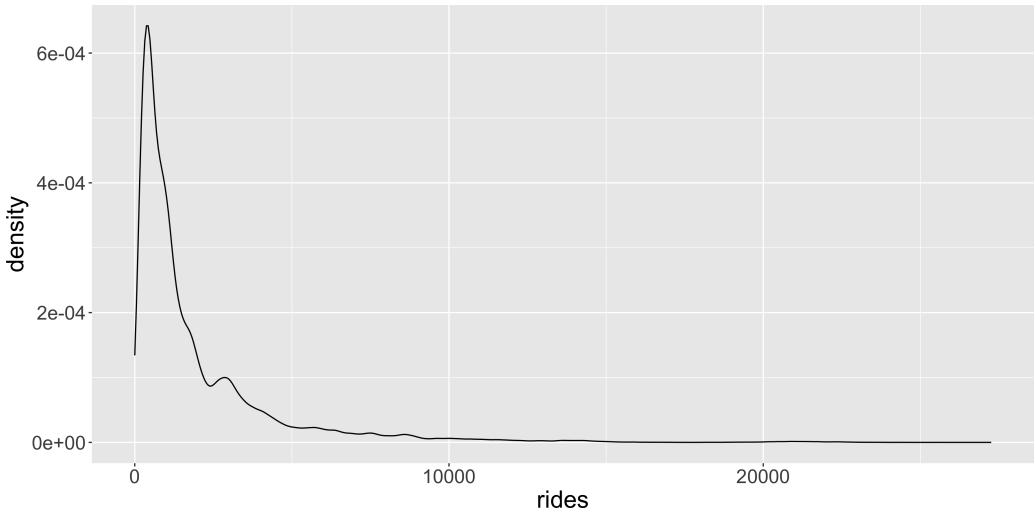
```
ggplot(data = cta_rides, aes(x = rides)) +
  geom_histogram(fill = "dodgerblue", color = "black", alpha = 0.5) +
  theme(text = element_text(size = 20))
```



We can change the colors of the bars and the size of the axis labels by specifying a few more arguments in `ggplot` We can increase the text size with the `theme` function as well as adding color arguments to the `geom_histogram` function. The fill and color arguments refer to the inside color of the bars and the outline of them respectively. The alpha changes the opacity of the bars which also makes it a little nicer to look at.

Another way of visualizing distributions is by using the `geom_density` function which uses a kernel density estimate to produce a smooth curve over the observed frequencies of values in the dataset. When using it the Y axis is a little harder to interpret than in a histogram, where the Y axis is simply the count of the observed value, but it is typically smoother and helps you "see" the distribution better. It is, in part, a matter of taste.

```
ggplot(data = cta_rides, aes(x = rides)) + geom_density() +
  theme(text = element_text(size = 20))
```



6.3.2 Trends Over Time

One of the primary things we want to look at, particularly with longitudinal data, is what happens as time goes by. To demonstrate this we will be looking at ridership by station. Considering that there are more than 100 stations in the dataset so we are going to only look at the top 20 by mean annual ridership. One thing we are going to be looking at is how the pandemic influenced ridership, which as you might imagine, was quite dramatic. To do this we need to do some data transformation.

First we need to find out which stations have the most ridership. To do this we take the dataframe which has a row for every station for every day there is data, and columns for the station name, date, and ridership, we extract the year from the date column, group by the station and the year in which the observation occurs, before calculating the mean and standard deviation for each.

```
summary_df <- cta_rides %>%
  mutate(year = str_extract(date, "\\d{4}")) %>%
  group_by(stationname, year) %>%
  summarise(mean_annual = sum(rides) / 12,
            sd_annual = sd(rides)) %>%
  ungroup()
```

Now we grab the top 20 stations by the number of mean monthly ridership before subsetting the complete dataframe to a smaller one to visualize. We then calculate a rolling mean so it is easier to visualize, otherwise the data would look very chaotic.

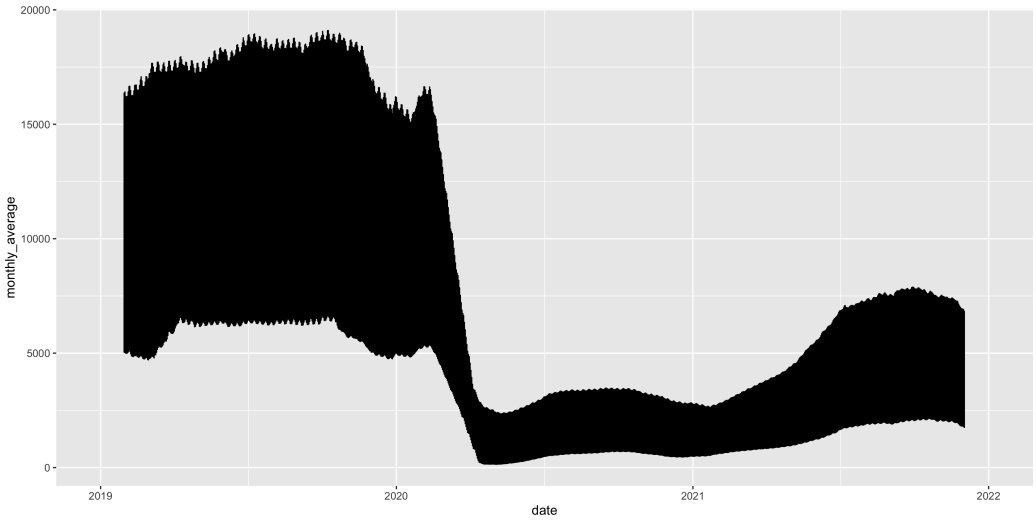
Now we are ready to visualize the daily rate of ridership across the top 20 stations by mean monthly ridership over time. We can do this with a simple line plot, using the `geom_line`

function in `ggplot`. In this first plot we are going to be mapping the rate of organ donation (the column/variable `monthly_average`) to the Y axis and the year in which the observation occurred (`date`) to the X axis to show change in rates over time.

In R you can arbitrarily break up code across lines to keep it "tidy". The style conventions in R is to keep lines under 80 characters long, that way when you view them in a text editor, they are completely visible without scrolling horizontally.

```
ggplot(data = station_roll_stats, aes(x = date, y = monthly_average)) +
  geom_line()
```

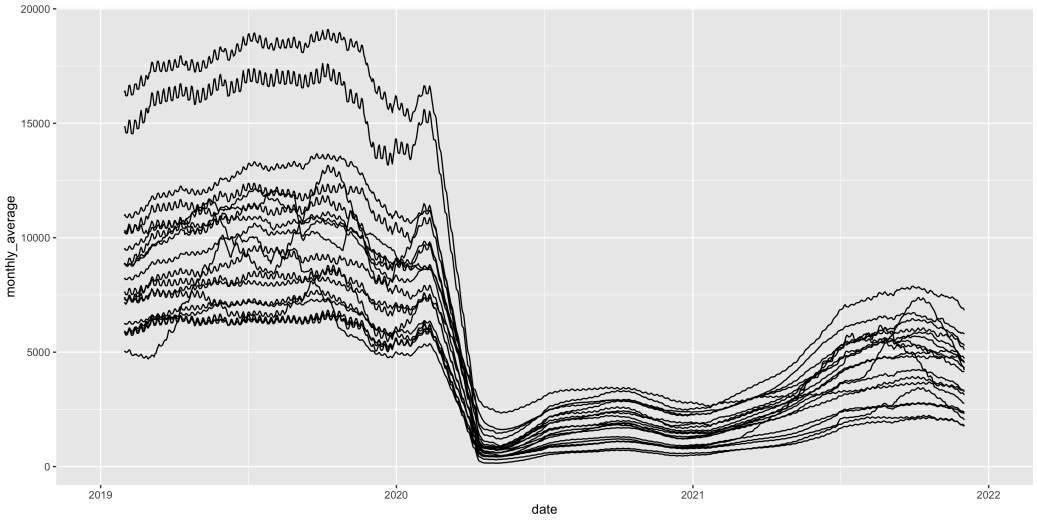
Figure 6.5: Something seems amiss here.



This is not as informative as we would have liked it to be because we are looking at daily ridership for 20 stations and all their value's are getting mapped onto the same points, making the plot uninterpretable. What we need to do is use another aesthetic mapping to tell `ggplot` to group some of the rows of the data frame and plot them across the X axis as single lines. We can do this by adding the `group` argument to the `aes` function inside the call to `ggplot`

```
ggplot(data = station_roll_stats,
       aes(x = date, y = monthly_average, group = stationname)) +
  geom_line()
```

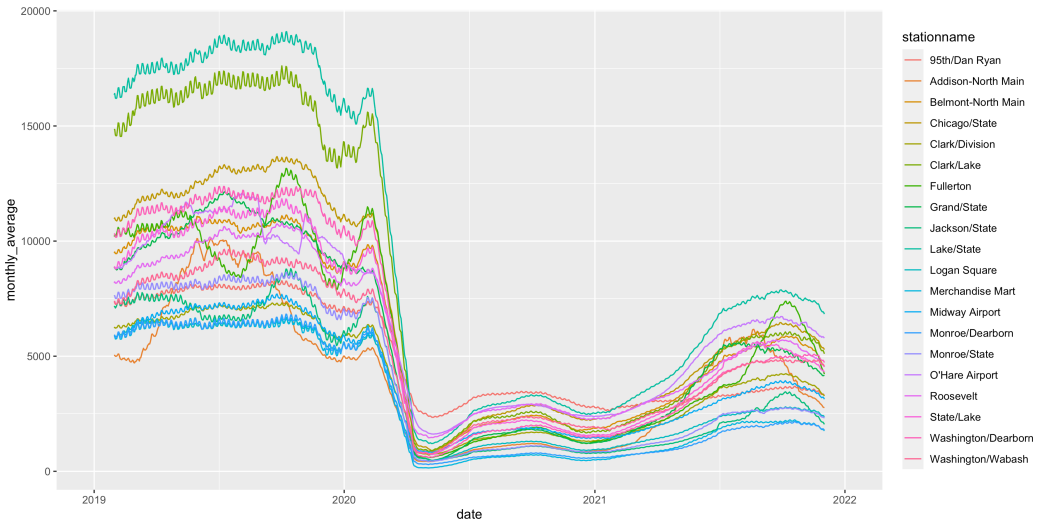
Figure 6.6: This looks better but still needs work



This figure is much better because it actually shows the different stations' change in rate of ridership over time. As you can see, in early 2020 there was a precipitous decline in ridership across the 20 different stations we are looking at but the lines aren't labeled in any way so we don't know which stations are at which line. To do this we can use a the color aesthetic mapping to label the lines

```
ggplot(data = station_roll_stats,
       aes(x = date, y = monthly_average,
           group = stationname, color = stationname)) +
  geom_line()
```

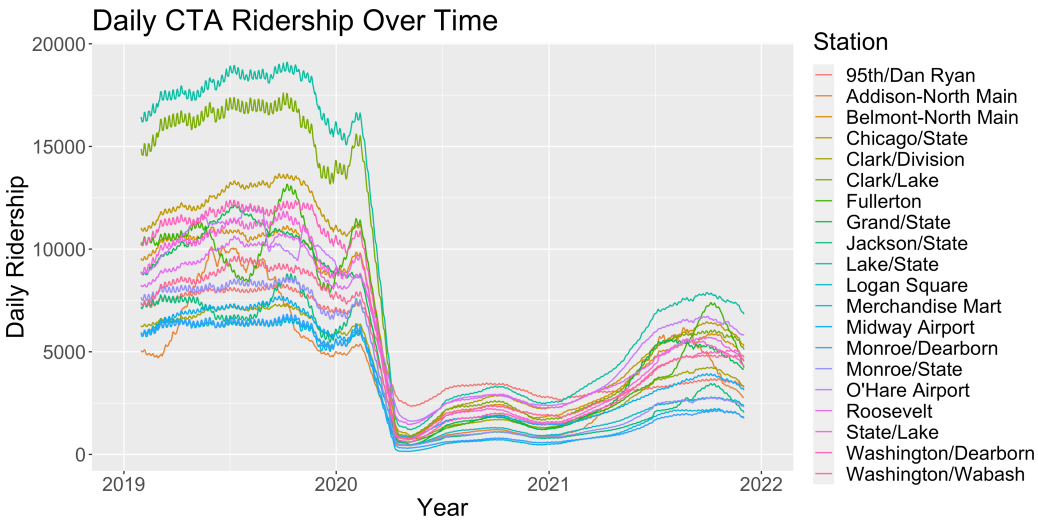
Figure 6.7: Now we can see who is who and what is what. Sort of.



Now we can see what is going on a little bit better but interpreting the plot is still pretty hard. It is pretty clear that Spain is not behaving like the other countries in the dataset and that there looks like there is two other distinct groups of countries, with the United States in one and Sweden in another. To help your reader grasp the plot immediately it is best to provide informative axis labels. The default behavior of `ggplot` is to just use the variable that is mapped to that axis as the label but we can specify what to actually call it with another layer to the plot. This time it isn't a `geom` function we call but a unique function called `labs` which allows you to add X and Y axis labels, a title, subtitle. And again we increase the text size for legiibility.

```
ggplot(data = station_roll_stats,
       aes(x = date, y = monthly_average,
           group = stationname, color = stationname)) +
  geom_line() +
  labs(y = "Daily Ridership",
       x = "Year",
       title = "Daily CTA Ridership Over Time",
       color = "Station") +
  theme(text = element_text(size = 20))
```

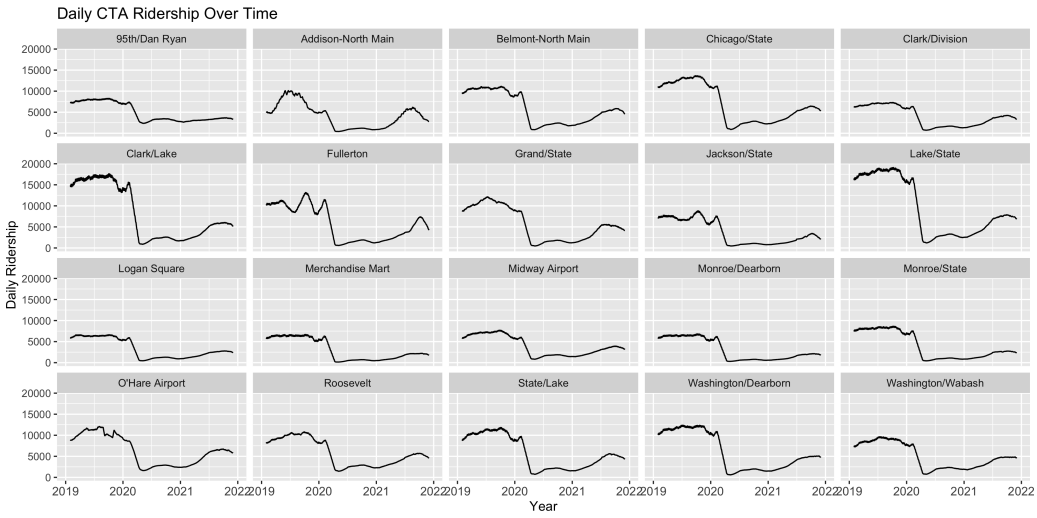
Figure 6.8: Properly labeled axes and a title help to make the figure more interpretable immediately to the reader



That looks more like a real plot but we might want to actually take a look at the individual countries to see what is going on in each of them and see if there are any particulalry surprising changes over time. As the figure stands above, that is somewhat obscured. We could do that by breaking up the data using filters and creating one plot per country but that is tedious. Luckily, `ggplot` has another function that allows you to created a *paneled* figure, so that a single plot is made for each country. We do this by using the `facet_wrap` function.[2]

```
ggplot(data = station_roll_stats, aes(x = date, y = monthly_average)) +
  geom_line() +
  facet_wrap(~stationname) +
  labs(y = "Daily Ridership",
       x = "Year",
       title = "Daily CTA Ridership Over Time") +
  theme(axis.text.x = element_text(size = 10))
```

[2]NOTE: There is a tilde before the variable we are grouping the data by for the panelling. That essentially tells the function to "group by" that variable and plot the X and Y axis.

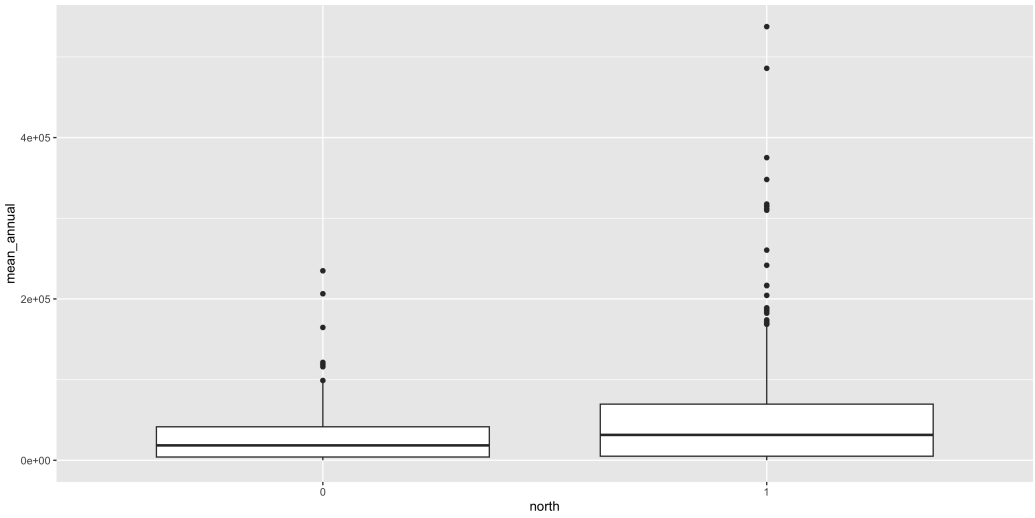


Now that helps to see hwo organ donation rates vary over time by country with a share range of X and Y values so each individual plot is comparable. What more, each plot is labeled with the country it is displaying so each is clearly identifiable.

6.3.3 Variation by Group

Another thing we will want to check out is the variation in the focal variable, the rate of organ donation in each country, by some categorical difference in those countries. One that immediately comes to mind is whether organ donation programs are something citizens opt in to or out of. My own informal hypothesis would be that there would be higher rates of organ donation in countries that include everyone in organ donation and require that they opt out rather than voluntarily opt in. To do this we would want to check out the mean and variation of organ donation across that category. We can use a "box and whisker plot" to plot this. This kind of plot displays a few important statistics: the median, lower and upper quartile (the box), and the minimum and maximum (the whiskers). The `ggplot` does something slightly different in that it shows outliers as individual points so it isn't strictly speaking using the minimum and maximum to the whiskers. We use the `geom_boxplot` function to do this after specifying the categorical variable in the X axis to plot the two groups and the same Y axis mapping as the plots above.

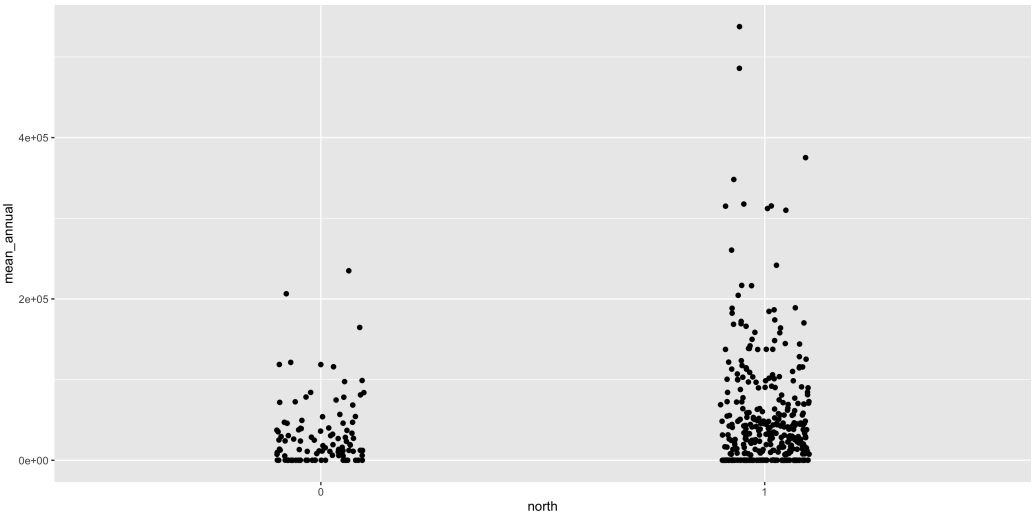
Figure 6.9: There does seem to be a difference but the relationship is obscured.



A few things should jump out to you: there does seem to be a difference between the two focal categories (opting in v. opting out) but the amount of variance in the two is different. What more there is a third inexplicable category. Apparently not every country has data on whether they have an opt in or out policy and so there is missing data. It is also important to note that we are pooling all the data across all the years observations occurred. Whether or not that is a defensible visualization strategy will be a matter of debate.

A principle that we are going to cover is that making a good figure helps you visualize where your data *is* (Martin 2018). The box and whisker plot obscures where each data point actually is. So let's do two things: drop the missing data, and use a different geom that is going to let us see where each data point is. To do that we use a function that drops missing data (`na.omit()`) and a different `geom`. This time we use `geom_scatter` to make a scatter plot.

A scatter plot lets you see every data point but the categorical variable makes all the data points line up and we can't see where every data point is. Considering that the X axis is a categorical (really a nominal) level variable and the specific position along that axis is not really meaningful we can "jitter" the points to let us see where they are along the Y axis. This is another geom that we can add to the plot which will randomly scramble the position along the X axis between a specific range.

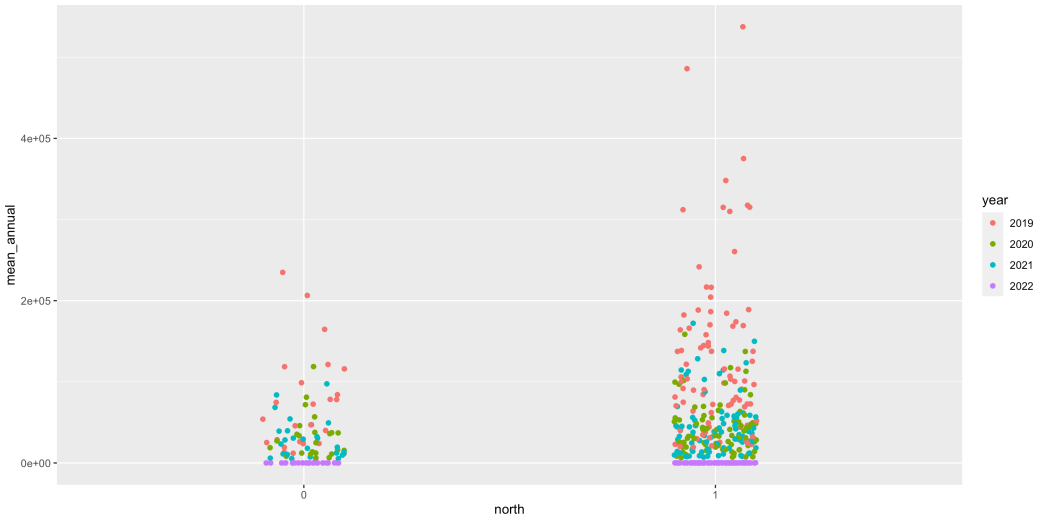


This looks much better and we can see some interesting things between the two countries. There does seem to be a difference between the two groups: the countries that require citizens to opt in to organ donation do tend to have a lower rate than those that require citizens to opt out, and they are much more concentrated around what is probably the mean. *But*, there are only a few observations below the probable mean of the Opt In group. These are what could be driving down the mean value and the difference between the groups.[3]

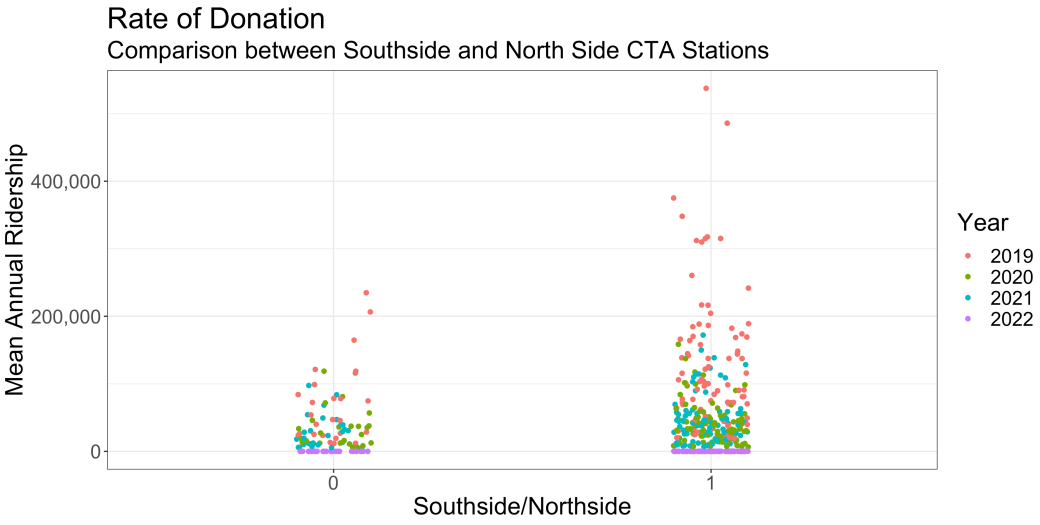
There is another variable in the dataset that we can use to help interrogate the differences between these two groups. The dataset includes a variable (`world`) that captures a typology outlined by Esping-Andersen in *The Three Worlds of Welfare Capitalism* which identifies three types of welfare state regimes by which advanced capitalist democracies can be categorized. These include: liberal, conservative, and social democratic. I'm not very familiar with what these specifically mean but we can add this to our plot to learn another feature of figure design. We can color the individual points by which category they fall in to. To do this we add another argument, `color`, to the aesthetic function in the call to `ggplot`.

[3] I know that I am playing fast and loose here with the comparisons and that a conscientious social scientist, like you, would do a few tests before this visualization and after to interrogate the differences. But we are learning about figure creation so we want to make figures.

Figure 6.10: There are notable differences in terms of the two groups and the different categories of welfare state regimes.



There are some notable and interesting differences on the basis of both the opt in/out comparison and the different welfare state regimes. There are many more Corporatist regimes in the Opt Out category (particularly with high rates of organ donation) compared to the Social Democrats in the same group. Let's clean up the figure to make it more immediately interpretable for a possible reader. We are going to add proper axis labels, a title, legend title and change the plots *theme*. The theme styles the general appearance of the plot, and in this case we are going to change the plot background from the standard `ggplot` grey to a clean white.[4]



[4] I once heard a professor deride the "ggplotification" of social scientific visualizations, and specifically the signature grey background.

6.4 Comments for Code

There is a convention in computer science to include "comments" in your code, prose style notes that annotate your code and make it clearer what you did and why. You indicate a comment in R by including a pound or hashtag (#) at the beginning of the line and the same for Python. It is important to include comments in your code for two reasons:

1. Showing what you did: If you fully adopt an open science posture to your own research and writing you'll probably disseminate a curated dataset and the source files that generate all your findings, including the RMarkdown file of your actual article. Providing informative comments to your code will make it clearer what you did and why to whomever is really interested in understanding what you did.
2. Knowing what you did: The more important reason, or at least the reason that is going to be most useful to you, is that you need to know what *YOU* did. You are going to put the project down and go on vacation or work on other things and you'll need to get back and know what you were doing. Most of the time, really nearly all the time, you are the only one who needs to know what was happening in your document and so write comments knowing what you'll need to know in the future.

6.5 What is Beautiful is Reproducible

Summary of the Chapter.

7 Version Control and Collaboration with `git` and GitHub

A key part of doing research is being able to account for how you got from your data to you findings, and being able to inspect how your project has changed over time is important to that process of accounting. Whatmore, we all want peace of mind as we do our research so that if we make a mistake, we can go back and do it over again. Version Control is a term for the process of developing a set of files and accounting for how they have changed over time. Software engineers need it as they are developing code and releasing it to users, they need to be able to ensure that it is compatible with other software. There are manual ways of doing version control, such as keeping all the old copies of documents in an archived folder, but adopting some software can help automate the process and ensure that there are no hiccups, while also allowing you to keep an elegant organization to your project directory without endless copies of nearly the same documents.

Modern Version Control software also allows you collaborate on files with others and then distribute them as well. In this chapter I will cover how to use this version control software, which runs on the terminal, so be prepared to open a terminal and type in commands. We are also going to try to collaborate on some documents together using GitHub. To do that, and to take full advantage of `git` for version control and backing up your work, you will need to sign up for a GitHub account. If you followed along in Chapter 4 you should have `git` already installed but if you didn't you can install it from here.

GitHub is free to use and allows you to have several private repos for free (we will cover what a repo is shortly, but it is just a directory of files you are tracking). You can pay to get unlimited private repos and some other bells and whistles. You can even host your website on GitHub. If you are worried about the security of your code or data on GitHub you can review their data use policies here. To sign up for a free account follow this link.

7.1 What is Version Control?

Version control refers to software that tracks the changes in groups of files. In academic research i will allow you do a few things: You'll be able to revert files to a previous state. So if you get some feedback from your peers and decide to rework your analyses or the current draft of your next big article and recirculate it and get even more negative feedback, you can immediately go back to the previous version and start over again.

Before going immediately going back to the previous version you can compare different versions of the file as long as you have been tracking it and telling the version control software to keep tracking it. You could even revert your whole project to a previous state if you realized you went down the wrong rabbit hole and need to start again from some more secure footing. even more importantly you can create branches in your project that lets you experiment with something new, all the while the canonical version of your project is safe and sound. And if you're collaborating with someone, you can see who authored particular changes.

Essentially these are also the basic reasons that you should use it:

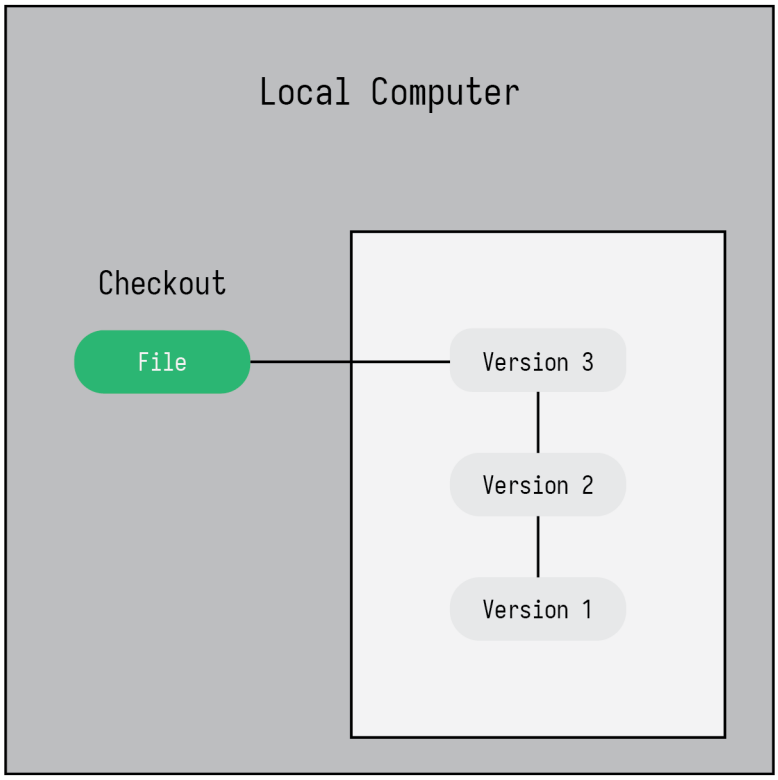
1. It keeps a record of everything, which means that you don't need to keep extra copies of your old versions of files because they are secure in the version history of your project.
2. There is a backup of everything. You can get old versions of files from the version history.
3. You'll be able to distribute the final project code and all directly to your audience by sharing the repository.

With all that said, it is important to note that version control is a *tool* in data management, and not a silver bullet. As is true about pretty much everything else in doing good resaerch, there are no substitutes for conscientiousness.

There are two varieties of version control: local and distributed. You can see an example of local version control in Figure 7.1. Version control keeps a snapshot of your file overtime (Version 1, 2, and 3 in the figure) and you are typically working on the latest version. You decide when to take a snapshot so there is some discretion in the process. It is not like a google doc that is constantly looking at every key stroke and edit you make. Local version control is limited in that if you lose your laptop that has the version history on it, then you lose everything. There would no backup unless you kept backups of your files remotely on something like iCloud or Google Drive. Also, if you are collaborating with some colleagues, you would still have to pass around files via email.

Distributed Version Control is a system for keeping a log of all the changes that have occurred to a set of files, and storing copies on a remote server, as is shown in fig. 7.2.

Figure 7.1: Local version control on a single computer.



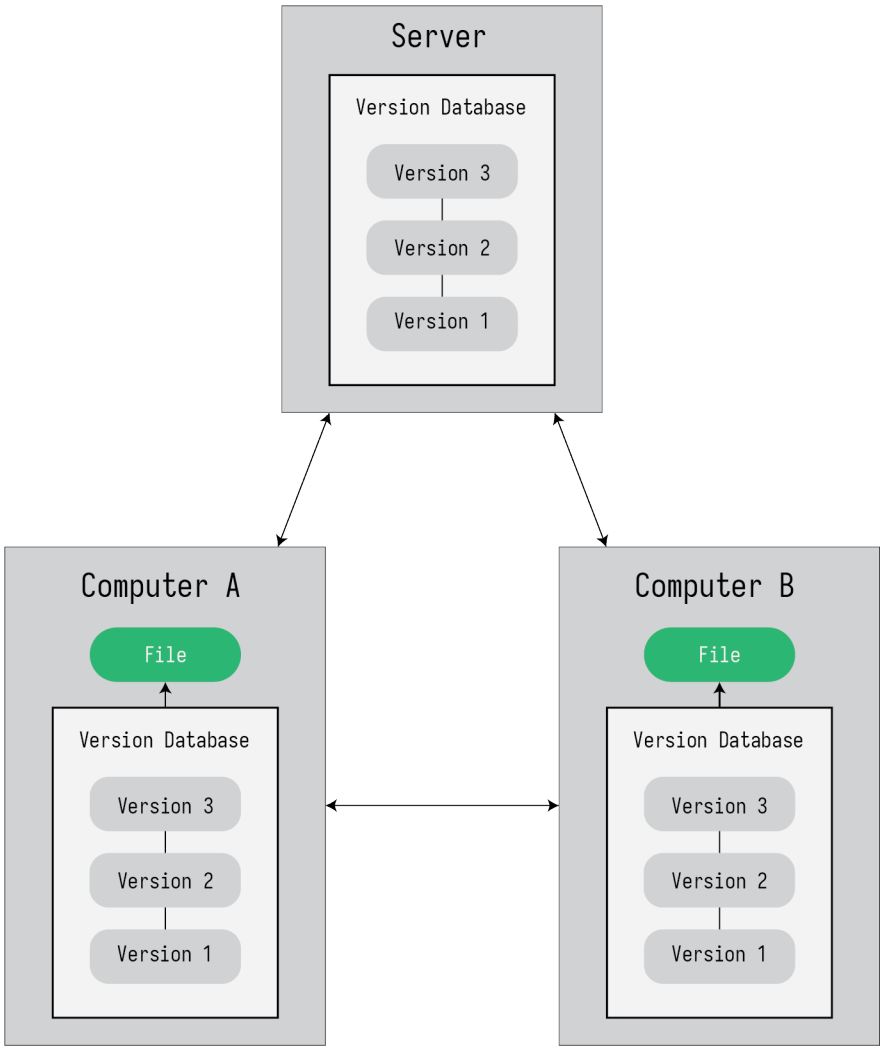
7.2 `git`

`git` is one of the most popular version control softwares out there. It is fairly simple to learn and very powerful but getting started using `git` can be a bit of a hurdle, and I have even known the smartest of graduate students abandon the attempt to learn it after a few tricky mistakes were thrown their way. This is in part due to the fact that there is a little bit of an esoteric vernacular associated with version control that is impenetrable at first. Before diving into how to use it we should get clear on some language first.

repo A repo is shorthand for repository, a repository being the database that stores all information about the version of your files.

add This command adds files that you want to track in the version history. There are a few fancy tricks for adding files. For example if you wanted to add literally everything in your project directory you would use the command `git add .`, if you wanted to

Figure 7.2: Distribted version control with a remote repository.



add all the R files in your `code` directory you would run `git add code/*.R`.

staging area When you add files to be added to the version history of your project, before creating a snapshot they are added to what is referred to as the staging area, which will list the files and their statuses that will be preserved in the version history. You can add or drop things as needed from the staging area.

`commit` This command creates a snapshot of the files that you have added to the staging area essentially creating a version of them in the version history of our project.

`git` stores the complete history of all the files and all the files changes that have occurred in your project and not just the current files you might have.

Now you are ready to start to learn by doing. So follow along with the next few steps on a repository that you can try this out on. One that has a couple of subdirectories of plain text files that you can experiment with, and not one of the directories you have installed from the templates, as those are already `git` repositories.

To get started you first initialize a directory with a `git` repository by opening a terminal, changing the directory to whatever directory you want to track and running `git init .` The terminal should return `Initialized empty Git repository in /Users/timothyelder/Documents/YOUR-DIRECTORY/.git/`. That is the path to the specific `.git` repo directory within your directory that will store all the information about the files you track. If you remember what we learned in Chapter 2, directories and files that begin with a period are hidden from view in a regular Finder or File Explorer windor. You can now run `git status` to ensure that it is running and it should print out all the flles in the directory and its subdirectories. They should all be in red because you haven't added anything yet to the repository.

When you are working on a project with plain text software like LaTeX, they generate lots of extra files that you don't really need but that help the software typeset your documents. Sometimes you also have `.log` files that you don't want to be tracked in your version history. Luckily, `git` has a system for ignoring specific file names and file types. To do that we are going to create another hidden file. You can create this file using a text editor or directly in your terminal with `vim`. We are going to cerate a `.gitignore` file which `git` recognizes and uses to ignore certain files.

Go ahead and create the file using `vim` by typing into the terminal in the directory `vim .gitignore`. Then once you have the file created you can put in the file extensions that you don't want tracked. I typically ignore files that are generated by OS as well as LaTeX log files that get generated while typesetting documents. So a typical `.gitignore` file looks like this for me:

```
# OS generated files #
######################
.DS_Store
**.DS_Store
.DS_Store?
._*
.Spotlight-V100
.Trashes
ehthumbs.db
Thumbs.db
**.icloud

# LaTeX generated files #
#########################
*.aux
*.lof
*.log
*.lot
*.fls
*.out
*.toc
*.fmt
*.fot
*.cb
*.cb2
.*.lb
```

If you have files that end in these extensions then and you run `git status` again, then you'll notice that they don't appear now when you run that command. Now that you have the repo initialized, a `.gitignore` file created, you are ready to add files to be tracked by `git`. Typically you'll want to initialize a `git` repository early on in your project when you don't have to many files in your project yet. To add everything to `git` tracking you simply run `git add .`. The period at the end of that command means "everything", but it won't add things that you are specifically ignoring. Go ahead and run `git add .` and then `git status` again. Now you'll see all the files that were red are now green, meaning that they have been added to the staging area to be added to `git` tracking.

The final step is to create a snapshot of the files that you have added to tracking, what is otherwise known as a `commit`. Every `commit` requires a small message, typically under 80 characters, that describes what is the content of the commit or what changes have been made. You keep these messages short so they can be red in a single line. To make a commit with a message you run `git commit -m ''init commit''` or whatever else you might want between the quotation marks.[1] Now, you have made your first commit and it is stored

[1] If you just run `git commit` without the `-m` flag, your terminal will open up a `vim` Window and you'll be able to leave a larger message. The convention is to keep the top level commit message to less than 80 characters,

locally on your computer.

Thus far we have run the following commands:

```
git init                        # initialize tracking
git status                      # shows the status of tracked files
vim .gitignore                  # hidden ignore file
git add .                       # adds everything to tracking
git commit -m "init "commit    # creates a snapshot of the project
```

Now when you work on your files you should start to get into the habit of making commits on a regular basis. If you are working on your project every day, all day, then one or two commits a day is not a bad idea. If you are only working on one file in your project on any given week and not much else is happening, then a commit a week is just as good. It is up to how conservative you want to be in your versioning. It is free, so you can be as fastidious as you want to be.

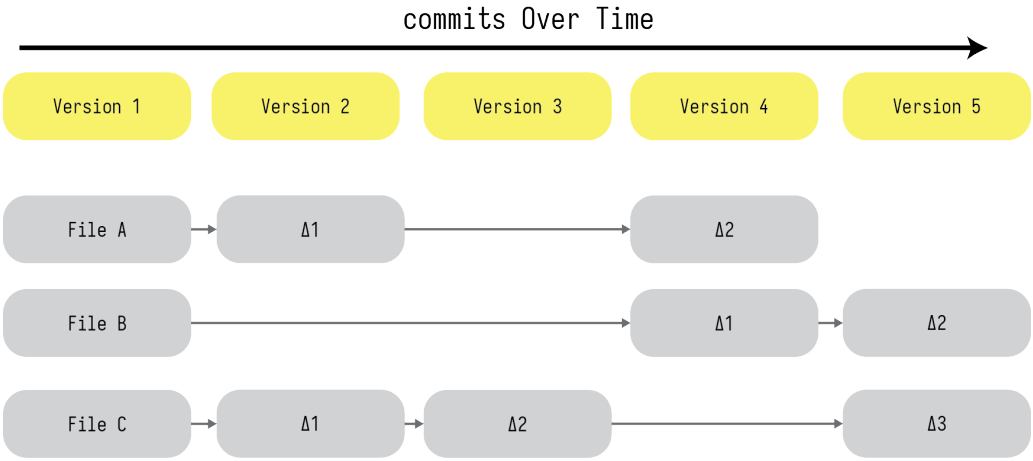
After you have a group of files being tracked and a few commits made you'll have a version history that looks a little like Figure 7.3. In this figure we have a repository with three files and five commits. `git` prizes efficiency so it does two important things: first, it doesn't keep exact copies of the files that you are tracking but a complete log of all the changes, so the full information about the file is conserved while the size is not. Second, it only stores new information when a file is changed. So in Figure 7.3 between the first and second commits, only files A and C had any changes and the delta between the two files are sotred while no new information is lofgged in file B because no changes ocurred. Between commmits 2 and 3 changes only occurred in file C. File B only gets a change at commit 4 and so it is updated from the initial version.

Another way of visualizing this can be found in Figure 7.4, where instead of showing the changes between different commits across files, it shows the snapshots of the repository, as it might appear in your finder, but with the extra time dimension to show changes over the commits. As you can see in each of the commits all the files appear because none of them get deleted. You should note that their version numbers change to reflect the changes that occurred in Figure 7.3. Files A nd C become files A1 and C1 between commit 1 and 2 while file B remains unchanged. File C1 becomes File C2 between commit 2 and 3, and so on and so forth.

This is in part indicative of the underlying strengths of `git`: rather than copying files endlessly, `git` compares files and their changes, storing the deltas between versions so there is a complete history of the files and how they have changed over time. There is some cryptography under the hood that ensures everything is as it should be and generates a

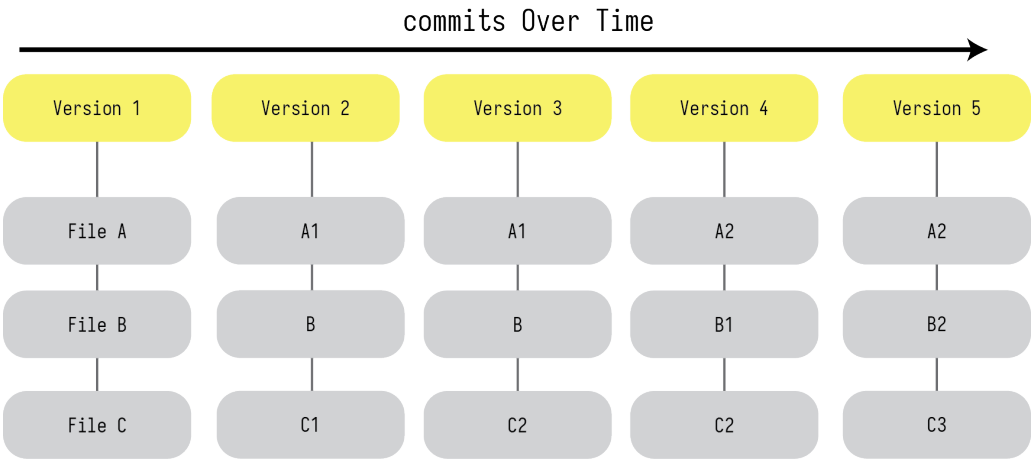
and then include however much test you want with a similar character length below it.

Figure 7.3: How versions are stored over time as you make `commits`



key for each commit and allows you to retrieve files from commits using these keys. It can be a little confusing to deal with this kind of system at first but it quickly becomes second nature once you start using it regulalry.

Figure 7.4: Another way of thinking about how versions are stored over time.

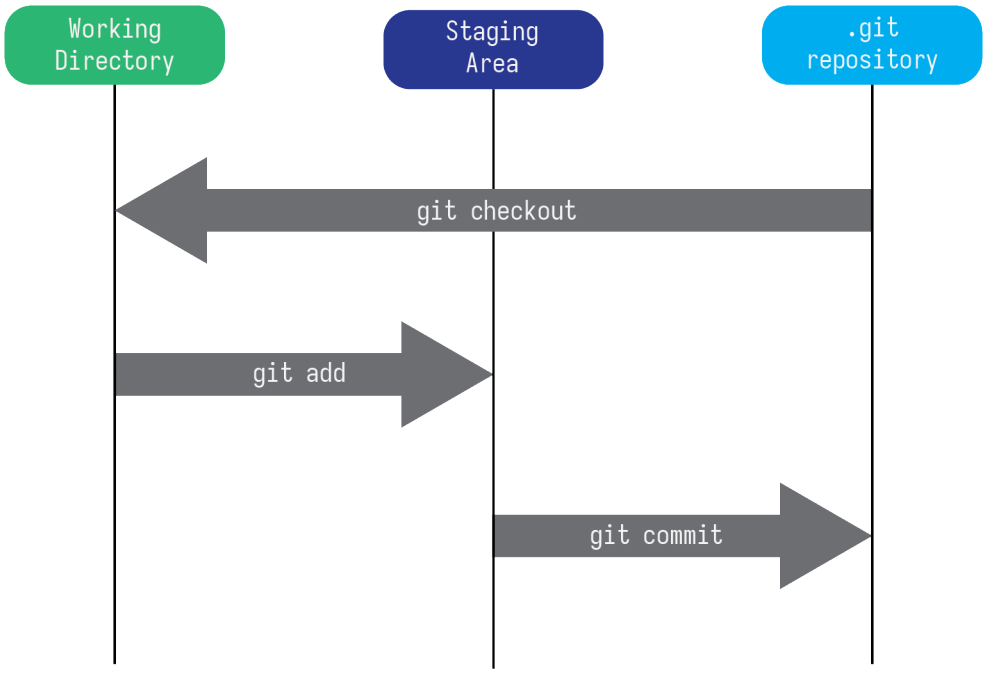


In Figure 7.5 we visualize how the process of working in this work flow looks. You are working in your prject directory (the working directory at the left hand side of the figure) and once you are ready to make a commit, you add the files you want (the `git add` arrow in the center of the figure) to the staging area, where they wait to actually be added to the version data base (the `.git repository` on the right).

There is one addition to this figure that we have not covered which is the `git checkout` command. `git checkout` is a part of the branching capability that `git` has which allows you to create paralell versions of hyour project to experiement with. This is most helpful

to quantitiative people who are going to be having lots of code files that they are working with.

Figure 7.5: `git` workflow with only local repository.

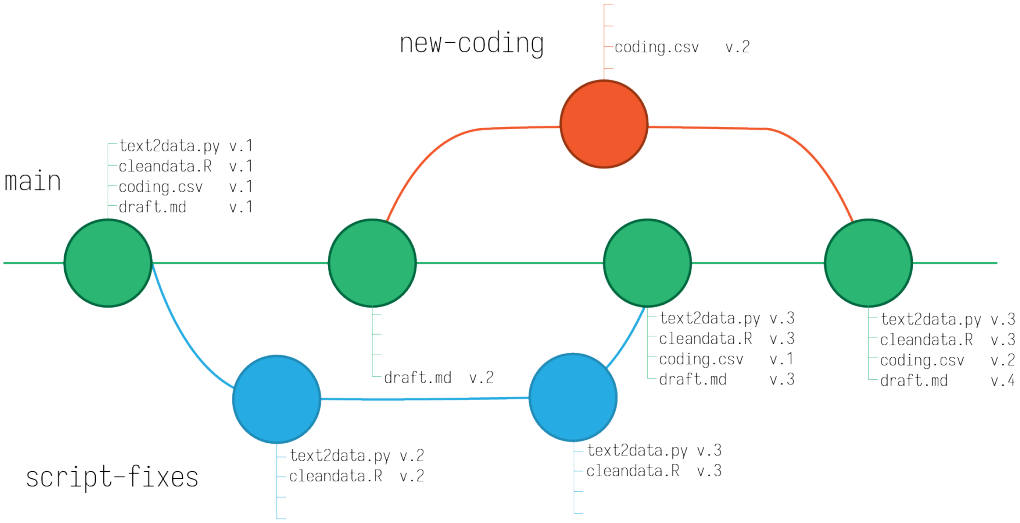


For example, in one of my projects, I had a lot of scripts that handled all sorts of data cleaning and generation tasks, and they were all written iteratively so that they all relied upon one another in a slighlty confusing network of code. I wanted to ensure that the process for data generation was streamlined, but I wanted to be able to ensure that I was able to generate the data the same way I had before, so whatever new procedure I had could be checked against the original. To do that I created a branch of my prokject directory that allowed me to significanctly change the scripts without having any fear that I might lose something, and without having to literally copy the folder and keep extra versions of things on my computer.

7.3 GitHub

GitHub is the cloud-based service that complements `git`. If you sign-up for a free account you can create remote repositories to backup your local repositoeies. This has two helpful use cases: for one you can ensure that no matter what happens to your local files you will still have a complete history of your project stored remotely. Further, you can use it to collaborate with colleagues remotely.

Figure 7.6: Branching versions of a project repository



Now there are a few extra pieces of jargon to learn now:

remote The copy of your repository stored on GitHub.
push Send current local versions to the remote repository.
pull Get any changes from remote repo.
clone Copy a remote repository to your computer.
HEAD The canonical top level commit of a repo.

7.3.1 `git config`

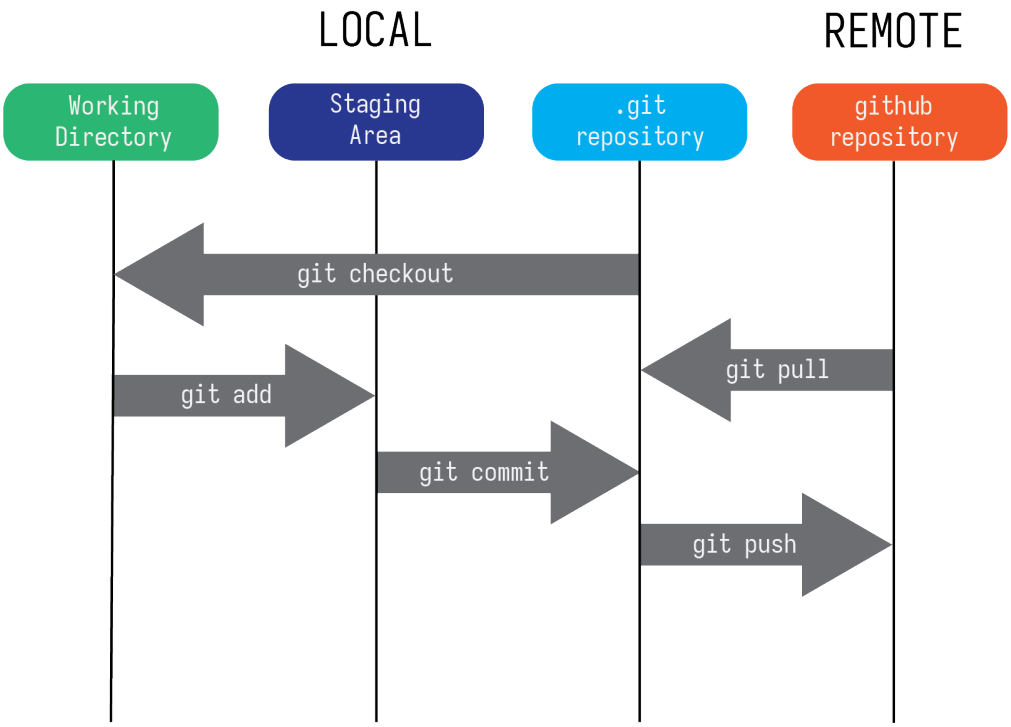
Once you have an account with GitHub you'll need to tell your local installation of `git` what your credentials are. This is referred to as your `git config`. You'll need to do this only once and make sure you know your GitHub username and the email that you use for your account.

You can list your current `config` settings by running `git config --list`. To get specific settings you can run `git config --get user.email`. If you haven't set up these settings they should return nothing. What we want to do is to set your `config` settings so that no matter where you are working on your computer, `git` knows what your confg settings are, these are referred to as your `global` settings.

For me, I set these settings using the following commands.

```
git config --global user.name "TimothyElder"
git config --global user.email "timothyelder@uchicago.edu"
```

Figure 7.7: `git` workflow with local repository and remote repositories.



You will need to substitute my user name and email for *your* user name and email. Go ahead and do that now in a terminal. It won't matter where your terminal is open because the `--global` flag will create a configuration file in a standard directory that is accesible anywhere on your computer.

7.3.2 Personal Access Token

There is one tricky part of using GitHub that reliably trips people up. GitHub is very concerned with ensuring that the security and your code is airtight. This is in part because of how ubiquitous the platform is for hosting and distributing large pieces of software that are used by lots of people.

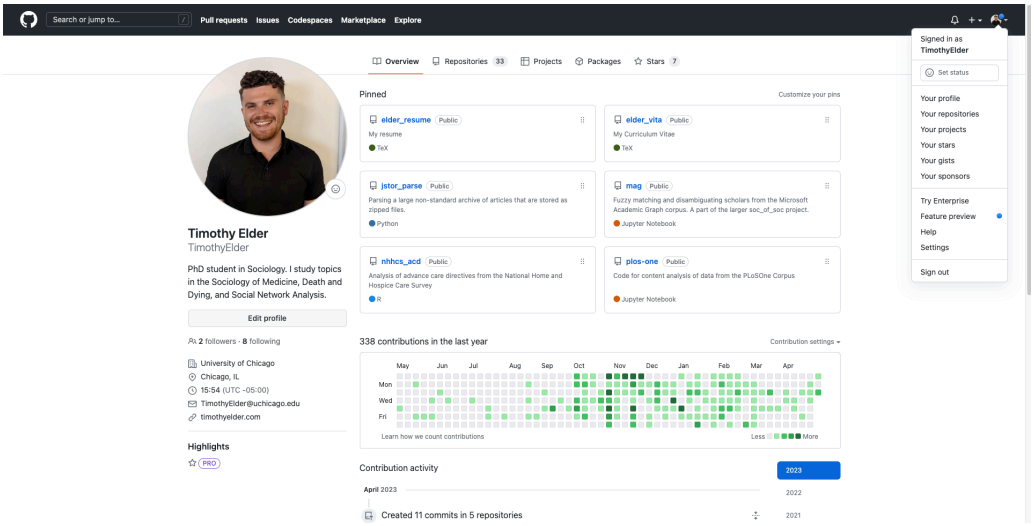
A personal access token is a little piece of cryptography I think that verifies who you are and is required for pushing and pulling to and from GitHub. It is sort of hidden but I will walk you through the steps. You can attempt to do it on your own following the instructions on this page but I will also enumerate the steps here.

Make sure that you have already installed git to your computer and signed up for a github account via the links above. To make sure that git is installed (it will not appear that in

your applications folder) open up a terminal and run the following code: `git --version`. If git is installed then it will return something like `git version 2.21.0 (Apple Git-122)`. If git is not installed then it will return something like `git: command not found`.

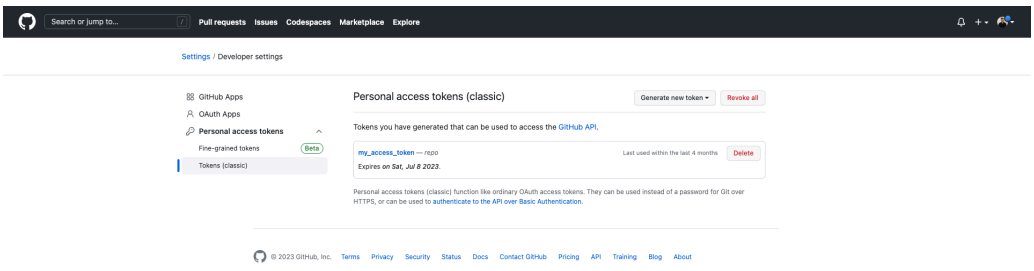
1. go to the home page of your github profile and on the upper right hand corner click the icon that has a picture, at the bottom click settings as in Figure 7.8.

Figure 7.8: A GitHub profile page with publicly available repos.



2. On the left hand sign of the screen there are a list of options, at the bottom is an entry called `< > Developer settings`. Click on it.
3. On the bottom left hand sign click on `personal access tokens` and then `Tokens (classic)`.
4. On the top right hand screen click on generate new token and pick the "classic" option. See Figure 7.9.

Figure 7.9: Settings for your personal access token.



5. Give the token a name in the "Note" field, mine is "my_access_token", set the expiration to "90 Days" and then click through all the options below that. Then generate the token and **note** the token that is returned to you. *You must note down this token* as you will not be able to see it again and you will need it. But do not fear, if you forget it, lose it or fail to write it down, you can regenerate the token. It will be redundant, but far from catastrophic.

8 A Larger World

There is more to learn about plaintext, and this book justs gets you started but hopefully it has given you a substantial command over the most important tools for adopting plain text software for your research and writing. It is as good a moment as any to reiterate that plaintext isn't for everyone; it presents complications to your work that some find to be excessive. Fiddling with the settings of your computer or installing esoteric software and establishing the appropriate workflow for working with these tools can seem to some scholars like a bridge too far, and so they settle for using Word and Google Docs, endlessly copying and pasting figures and tables into their documents and remembering the citation conventions in their fields. That is all well and good but learning plaintext for authoring papers is only the beginning of what we can use plaintext tools for in communicating our scientific findings. This larger world of plaintext tools is vast but I think there are a few next places you should visit.

Now that you have an understanding of the conventions of markup languages including LaTeX, markdown and Rmarkdown, you can take advantage of these to author your own websites. Hugo is a static site generator that allows you to use markdown files to create websites without having an extensive understanding of HTML. Hugo offers customizable templates that have beeen created for a variety of purposes, that are both beautiful and simple to use. You can even go one step further and use Rmarkdown and the `blogdown` package to use dynamic documents to create websites. This will be particularly helpful when authoring documentation for packages you write or for providing resources for reproducing your work to a wide audience. I have used both extensively to create a number of websites including my personal site, timothyelder.com as well as a site for this book, introtoplaintext.com. Using Hugo you can host your website on GitHub using a custom domain purchased from domain registrars like GoDaddy, Squarespace, or DreamHost.

Using plain text tools you can pursue a variety of different avenues to further communicate your work to a wide audience. You can author books and technical documents using RMarkdown and the bookdown package, create websites with Hugo and blogdown, or integrate code into your slides using sweave.

References

Bhattacharjee, Yudhijit. 2013. "The Mind of a Con Man." *The New York Times Magazine*.

Durkheim, Émile. [1893] 2008. *The Division of Labor in Society*. 13. [Repr.]. New York: Free Press.

Healy, Kieran. 2020. "The Plain Person's Guide to Plain Text Social Science."

Martin, John Levi. 2018. *Thinking Through Statistics*. Chicago ; London: The University of Chicago Press.

Scheiber, Noam. 2023. "Harvard Scholar Who Studies Honesty Is Accused of Fabricating Findings." *The New York Times*.

Singal, Jesse. 2015. "The Case of the Amazing Gay-Marriage Data: How a Graduate Student Reluctantly Uncovered a Huge Scientific Fraud." *New York Magazine*.

Wilkinson, Leland. 1999. *The Grammar of Graphics*. New York: Springer.

Windows Supplement

First a moment for evangelism. Windows is far and away the most popular operating system in the world for desktops and laptops, with 76% of the global market share, followed by macOS (16%) and then Linux (5%). Pretty much everyone has a working familiarity with Windows, and what more it is meant to be user friendly for the average person. But Windows offers a much more What You See Is What You Get Approach to computing. Window's user-friendliness comes at the cost of controlling every part of your operating system.

Yes, Apple products are more cost prohibitive than products that use Windows, and Linux can be intimidating as it has comparatively less support than the more popular alternatives. With those caveats, macOS and Linux (both UNIX based operating systems) are comparatively better development platforms for the kind of computing tasks we will be covering in this workshop. For one, their file structure is much more intuitive and includes fewer redundancies, so finding where to install the software we are working with is going to be achieved much quicker on macOS or Linux than on Windows. This is to say that Windows users have a few extra steps in the noble task of switching to plain text but this guide will help you every step of the way.

For installing LaTeX, pandoc and Git you can follow the steps outlined in Chapter 4 as they are the same in Windows and macOS. There are going to be slight differences for Windows when adding the document templates and ensuring that a few extra helper programs are installed.

8.1 Installing Document Templates

Now that you have LaTeX and pandoc installed you are ready to start using plain text software for your research and writing. But if you want to have nicely formatted documents that handle all the bells and whistles that you need for your scientific and social scientific writing then you need to do just a few more steps. This is the harder part of the installation process but I will take you step by step through it.

Ensure that the software you need is installed by running the following code in a terminal:

```
pandoc --version

latex --version

git --version
```

Each of these commands, if the software is installed, will print out version information. If you the software is not installed then your terminal will say something like `python: command not found` or `latex: command not found` which means something has gone wrong and you will have to attempt to install the software again. ***NOTE***: Go to Section 8.7 below for learning out how to add to your PATH variable which could solve your problem for you.

You need to make sure that you create a directory and a sub-directory the custom LaTeX templates, class and style files. Run the following code (changing the paths for your machine):

```
mkdir C:\texmf && mkdir C:\texmf\tex && mkdir C:\texmf\tex\latex
```

This creates a directory where you install the custom class and style files for LaTeX. With those directories made you can install the templates into their proper locations with the following code (make sure to change the paths to match your computer):

```
cd C:\Users\Timot\AppData\Roaming
git clone https://github.com/TimothyElder/pandoc-templates.git
ren "pandoc-templates" ".pandoc"

cd C:\texmf\tex\latex
git clone https://github.com/TimothyElder/latex-custom-te.git

cd C:\Users\Timot\Documents
git clone https://github.com/timothyelder/md-starter
```

Windows has an extra step which includes you adding a directory to your LaTeX installation so it knows where to look. To do that do the following steps:

1. Start MiKTeX Console (either find it in the Start menu or search for it in the search bar) and open the Settings page.
2. Click the Directories tab.
3. Click the Add button and browse to the texmf root directory created earlier (`C:\texmf`).

Make sure that you change all the paths to match your machine otherwise you get lots of errors and nothing will work. If you did everything above right, Congratulations! You are one step closer to making beautiful and reproducible documents in plain text.

There are a few key pieces of software that Windows does not come installed with that you will need to take the extra step of installing.

8.2 `winget` Package Manager

`winget` is a command line based package manager for Windows that helps with installing other pieces of software. It is part of App Installer which you can download and install here.

8.3 Make

Make is a tool that handles the creation of files from some source material and is super helpful when you need to repetitively run the same programs in the same order over and over again, such as when you are editing and typesetting your next paper. With `winget` installed, Make is super easy to get, just run this line of code in a command prompt on your computer:

```
winget install GnuWin32.Make
```

8.4 Python

Python is a modern general computing language that is very common in the sciences and social sciences. Though we are not going to be actively using it in the working group, some of the software that we use relies upon it. It doesn't take up much space and can be downloaded and installed here.

Alternatively, you can install python with `winget` using the following command:

```
winget install python
```

If you do use `winget` you may have to update your PATH variable manually, so the command prompt can find it (See Section 8.7 below for help adding to PATH in Windows).

8.5 pandoc-xnos

`pandoc-xnos` is for cross-referencing figures, tables, equations, sections and pretty much anything else that can be written in a Markdown file. This helps for automatically handling the labelling of different parts of your document, so you can move the position of a figure or table or equations without having to change the reference to it in the body of your text. Install with:

```
pip install pandoc-fignos pandoc-eqnos pandoc-tablenos \
            pandoc-secnos --user
```

And then use it with `--filter pandoc-xnos` as a flag for the `pandoc` command in your `Makefile` recipe. **NOTE**: Whenever using `pandoc-fignos` or `pandoc-xnos`, they need to be invoked before the `--citeproc` filter as they both use the @ symbol to identify references and they will get confused otherwise.

8.6 Installing a Text Editor

One of the advantages of plain text is that you can open your files on any computer with the native programs installed on it from the factory. Every computer will have a basic text editor (macOS has TextEdit and Windows has Notepad) and you could do everything we are going to do in this working group with the command line and one of these text editors (in fact you could do everything just with the command line). I *highly* recommend you install a more advanced text editor to do your work in.

There are a few excellent options to choose from including Sublime Text, Emacs, Atom, Notepad++ and RStudio. I use Microsoft's Visual Studio Code (or VS Code for short) which is particularly good and seems to be becoming the standard. The advantage to using a text editor regardless of which specific one you choose is that they highlight the syntax of whatever programming language you are writing in (including Markdown and LaTeX) enhancing the readability of your code to help writing and debugging. What more, you can typically run everything directly in the text editor so you never need to navigate away from a single window to get all your work done. You can run `R`, `python`, and keep your LaTeXdocument open all at the same time toggling between the different tasks as you need.

I really think that you should use VS Code and it is what I will be using throughout the working group and so go ahead and install it here. One of the nice things about VS Code is that it is *extensible* and has all sorts of extra features designed and implemented by users

to help you get things done. One of the most important extensions is a LaTeX helper that we will be using when we do have to edit or write in LaTeX. After you install VS Code go ahead and install the extension here. There are also other helpful things for writing like a spell checker, a tool for auto-completing citations, a word counter, and support for your favorite (or not so favorite) statistical software. And don't worry, installing these extensions is just a matter of clicking a button.

8.7 Adding to PATH in Windows

A lot of the work of typesetting is going to be taking place in the terminal and sometimes you will get an annoying an inexplicable error like "this software is not on the PATH" or "I can't find this software", which means that your computer is searching the location where executable programs are located (the PATH variable) but it is not finding whatever software you are telling it to run. Why this happens is a mystery. Sometimes the software has installed onto your computer, but the terminal program doesn't know where to look for where the programs are installed. I know this is a little confusing but here is a quick lesson. When you invoke a command on the terminal the first thing the computer does is check an index of the available software, which it does by examining what is called the PATH variable. If it finds whatever you are asking it to use like `pandoc` or LaTeX or R or python or whatever, it runs the command. If it doesn't find what you tell it to look for it returns an error.

Before giving up completely try updating the PATH variable so the terminal knows where the software is. Depending on what specific command line or terminal you are using the instructions are slightly different but they will be pretty portable across platforms, and operating systems. This is one of those things that you really only have to learn once and then you'll be able to improvise a lot better later. To find out where the files for a program are on macOS, run the `which` command followed by what your looking for, such as `which pandoc`, and it will print the path to where the executables of these files are installed. Sometimes the data files for a program get installed onto our computer without the PATH variable getting updated to tell the terminal that the software is installed. Check below for relevant information about how to add to the path:

8.7.1 Windows

1. The first step depends which version of Windows you're using:

- If you're using Windows 8 or 10, press the Windows key, then search for and select "System (Control Panel)".
- If you're using Windows 7, right click the "Computer" icon on the desktop and click "Properties".

2. Click "Advanced system settings".
3. Click "Environment Variables".
4. Under "System Variables", find the `PATH` variable, select it, and click "Edit". If there is no `PATH` variable, click "New".
5. Add your directory to the beginning of the variable value followed by `;` (a semicolon). For example, if the value was `C:\Windows\System32`, change it to `C:\Users\Me\bin;C:\Windows\System32`.
6. Click "OK".
7. Restart your terminal.

8.8 Installing `vim` on Windows

`vim` is a command line based text editor that we are going to be using for doing quick edits to files. Unfortunately, Windows does not come with it pre-installed like it is on macOS and Linux so Windows users should do the following steps to make sure you can use `vim` during our first session.

1. Download the following installer.
2. Open the downloaded file.
3. Accept the User Agreement.
4. On the next window ("Choose Components") make sure to select the "Custom" install type from the dropdown menu and that the "Vim Console Program", "Create .bat files" and "Create Default Config" options are selected with checkboxes.
5. Leave the next window ("Choose _vimrc settings") as it is.
6. Hit "Install" on the last window and the program will install.

The program should then be installed and ready to use for Friday's session.

Figure 8.1: Check off the proper settings as in this image.

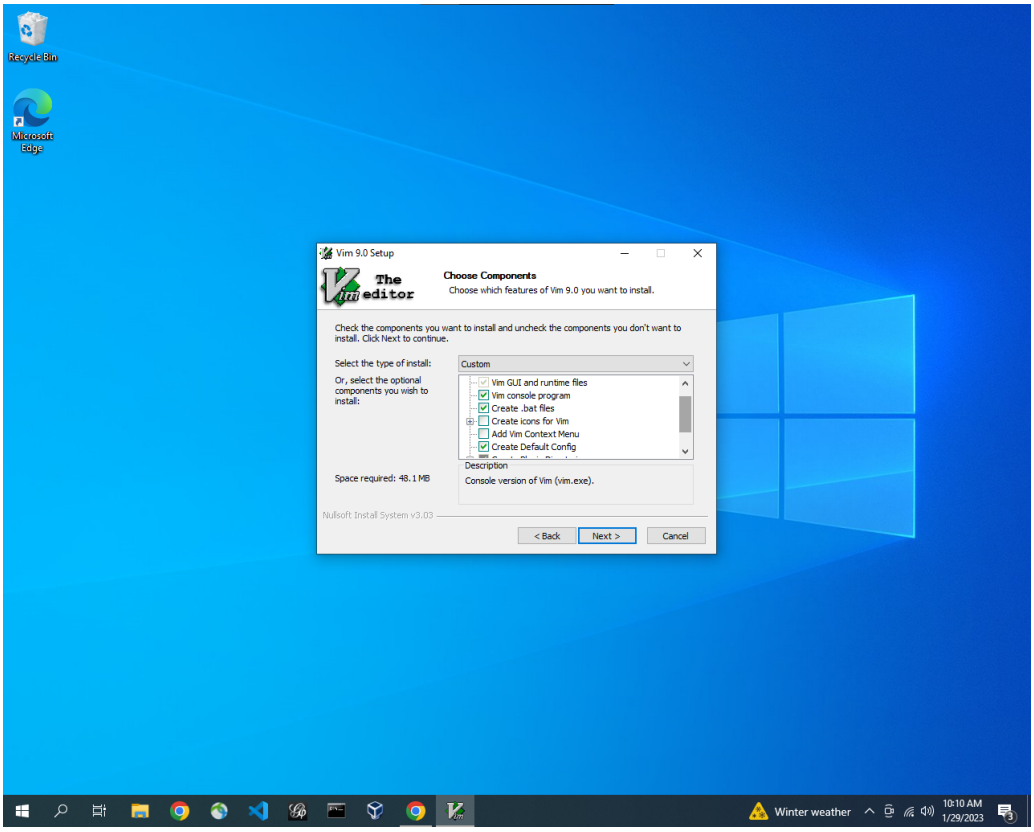


Figure 8.2: You can leave this window set to the defaults.

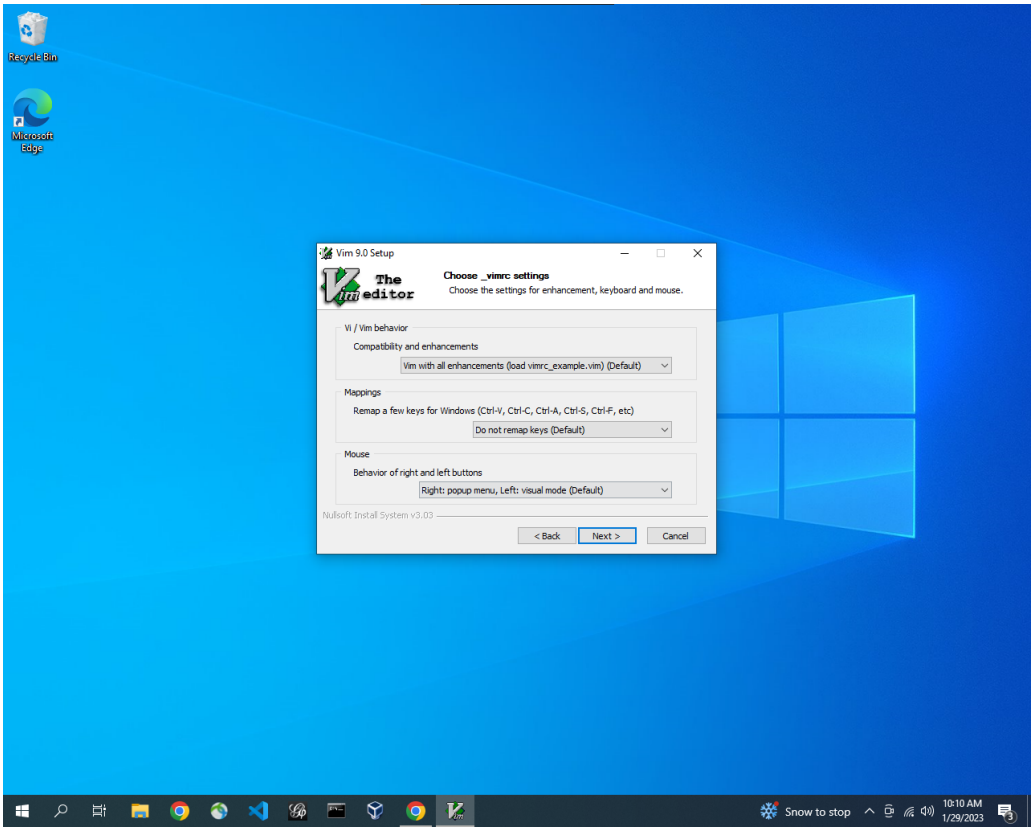


Figure 8.3: Click install on the final page and you're all set.